Computability
Some unofficial notes

Master's degree in Computer Science
A.Y. 2023/2024

These LaTeX notes originates from some handwritten material distributed for the course in Computability at the Master Degree in Computer Science at the University of Padua. I am grateful to Riccardo Borsetto, Mert Anil Hasret, Luca Zaninotto for translating the notes (originally in italian) and producing the LaTeX version.

They are unofficial sketchy notes, which are intended as a support for students for understanding what is discussed in the course. They can contain errors or inaccuracies (in case you find some, please report them).

They are not intended in any sense as a replacement for the official book (Nigel Cutland "Computability. An Introduction to Recursive Function Theory"), on which they are heavily based.

Paolo Baldan

Contents

CHAPTER 1

Introduction

In this chapter, we informally discuss the notion of *effective procedure* and *function computable* by means of an effective procedure. This will lead us to single out the main features of an algorithm/computational model. Despite being informal, these considerations will allow us to derive the existence of non-computable functions for every effective computational model. In the next lessons these notions and considerations will be formalized by setting a specific computational model, a kind of idealized computer.

1.1. Algorithm or effective procedure

Effective procedures and *algorithms*, even though we do not always call them in this way, are a part of our everyday life.

For example, at the primary school we are not only taught that given two numbers their sum exists, but we are also provided with a procedure to compute the sum of two numbers!

In general terms, an *algorithm* can be defined as the description of a sequence of *elementary steps* (where an "elementary step" is a step which can be performed "mechanically", without any intelligence) which allows one reach some objective. Typically, the aim is transforming some input into a corresponding output, suitably related to the input. This could be transforming ingredients into a cake, although normally we are interested in computational problems.

EXAMPLE 1.1. Some examples are:

(1) given $n \in \mathbb{N}$, establish whether n is prime;

(2) find the nth prime number;

(3) differentiate a polynomial;

(4) perform the square root $\sqrt{n}$;

(5) find least common multiple lcm and greatest common divisor GCD.

Therefore we can think of an algorithm as a black box

$$\text{in} \to \boxed{blackbox} \to \text{out}$$

where the transformation is performed by executing a sequence of elementary instructions.

If each step is *deterministic* (i.e., for each state of the system, the instruction to be executed next and the new state it produces are uniquely determined), then each possible input will uniquely determine the corresponding output (or the procedure might not terminate, in which case we will have no output).

In mathematical terms, the algorithm determines a (partial) function

$$f : input \to output.$$

We say that f is the function *computed* by the algorithm and that f is effectively computable. Thus, we can give the following first definition of a computable function (still informal since it refers to a generic notion of algorithm).

DEFINITION 1.2 (Computable function). A function f is *computable* if there exists an algorithm that computes f.

We stress that for f to be computable, it is not important to know which is the algorithm that computes f, but we just need to know that some algorithm that computes f exists.

EXAMPLE 1.3. According to the above definition, we informally expect the the following functions to be computable:

- GCD (greatest common divisor), e.g., exploiting Euclid's algorithm.

- the function $f : \mathbb{N} \to \mathbb{N}$ defined as

$$f(n) = \begin{cases} 1 & n \text{ prime} \\ 0 & \text{otherwise} \end{cases}$$

- $g(n) = p_n$
  where p_n is the n-th prime number.

  This is computable by generating numbers and testing for primality until the n-th prime is found. ==> get all the prime numbers

- $h(n) = \pi_n$ where π_n is the n-th digit of the decimal representation of π.

  Indeed there are

    - series that converge to π

    - techniques to estimate (by excess) the error caused by

        * truncating a series

        * rounding in the computation of the value of the truncated series

What about the function below?

$$g(n) = \begin{cases} 1 & \text{there is a sequence of exactly } n \text{ consecutive 5's in } \pi \\ 0 & \text{otherwise} \end{cases}$$

For example $g(3) = 1$ if and only if $\pi = 3.14 \ldots i555j \ldots$, with $i, j \neq 5$.

A naive algorithm could consist in generating the digits of π until a sequence of 5's of the desired length n is found. Clearly, if such a sequence exists, it will be eventually found and the answer 1 will be returned. However, at no stage of the computation, we can exclude that the desired sequence of n 5's will appear later; hence, apparently, we have no way of returning 0.

REMARK 1.4. On the basis of the considerations above, the following

- generate all digits in the decimal representation of π;
- if they include a sequence of n consecutive 5's then $g(n) = 1$;
- otherwise $g(n) = 0$.

is *not* an effective procedure.

Note that this does not mean that g is not computable, i.e., that an effective procedure for computing g does not exist, but at the moment this procedure is not known (to us)!

We do not know if g is computable, but there might be a property of π that allows us to conclude. In particular, there is a conjecture that all finite sequences of digits appear in π, which would imply that g is simply the constant 1, whence computable.

Consider now a slightly different function $h : \mathbb{N} \to \mathbb{N}$, defined by

$$h(n) = \begin{cases} 1 & \text{there is a sequence of at least } n \text{ consecutive 5's in } \pi \\ 0 & \text{otherwise} \end{cases}$$

The function seems very similar to the one considered before. However, note that if $\pi = 3.14\ldots i555j\ldots$, then we deduce, not only that $h(3) = 5$, but also $h(2) = h(1) = h(0) = 1$. More generally, whenever $h(n) = 1$ then $h(m) = 1$ for all $m < n$. This suggests that h could have a quite simple shape.

More precisely, consider $K = sup\{n \mid \pi \text{ contains } n \text{ consecutive digits 5}\}$. Then we have 2 possibilities:

(1) K is finite, and thus

$$h(n) = \begin{cases} 1 & \text{if } n \leqslant K \\ 0 & \text{otherwise} \end{cases}$$

* functions

(2) K is infinite, and thus

$$h(n) = 1 \text{ for all } n \in \mathbb{N}$$

*

This implies that h is computable because it is either a step function or a constant function, and these function can are computed by simple programs. One could object that we do not know which shape the function has and thus we do not know exactly which is the program that computes the function. This is true, but irrelevant for computability.

Trying to replicate the same argument for function g fails. In fact, one could think of defining $A = \{n \mid \pi \text{ contains exactly } n \text{ consecutive 5's}\}$. Then

$$g(n) = \begin{cases} 0 & n \in A \\ 1 & n \notin A \end{cases}$$

This does not suggest that g is computable. Set A is possibly infinite and we do not see a way of providing a finite representation of A which can be included in a program.

Bringing the argument to the extreme, one could consider the function $G : \mathbb{N} \to \mathbb{N}$ defined by

$$G(x) = \begin{cases} 1 & \text{if } P = NP \\ 0 & \text{otherwise} \end{cases}$$

Since the condition does not depend on the variable x, the function is either the constant 0 or the constant 1. Independently of which of the two cases applies, the function is computable.

Algorithms and existence of non-computable functions

2.1. Characteristics of an algorithm

We present a list of features that an algorithm should satisfy in order to capture the intuitive idea of *effective procedure*. Roughly, what we ask is that an algorithm is "implementable" on some sort of idealised machine, that we call the computational model. Hence, below, we also list some requirements that the computational model should meet to be considered effective.

An *algorithm* is a sequence of instructions with the following characteristics:

a) it is of *finite length*;

b) there exists a *computing agent* able to execute its instructions;

c) the agent has a *memory* (for storing the input, intermediate results to be used in subsequent steps and the output);

d) the computation consists of discrete steps;

e) the computation is neither non-deterministic nor probabilistic (we model a digital computer);

f) there is no limit to the size of the input data
(we want to be able to define algorithms that work on any possible input, e.g. + operating on summands of any size);

g) there is no limit to the memory that can be used.

This requirement may seem less natural, but having unbounded memory is essential to avoid the notion of computability being dependent from the available resources. In fact, for many functions the space required for intermediate results depends on the size of the input,

e.g. $f(n) = n^2$ then $(1000)^2 = 1000000$. Note that I must add a number of zeroes that depends on n and thus n must be stored (the states are finite);

h) there exist a finite limit to the number of the instructions and to their complexity.

This is intended to capture the intrinsic finiteness of the computing device (justified by Turing with the limits of the human mind/memory),

e.g. for a computer, the memory that can be accessed with a single in-struction must be finite (even if by (g), the memory is unbounded);

i) computations might

(a) terminate and return a result after a finite, but unbounded number of steps (e.g. the square function requires a number of steps proportional to the argument);   ==> output

(b) continue forever, without returning a result.   ==> no output

2.2. Existence of non-computable functions

Later on, we will focus on a concrete computational model and this will allow us to give a completely formal definition of computable function. Now we argue that, simply on the basis of the assumptions above, we can infer the existence of non-computable functions for every "effective" computational model.

2.2.1. Some mathematical notions and notation. We start by recalling some basic notions and introducing useful notation.

- We will consider the set of *natural numbers* $\mathbb{N} = \{0, 1, 2, \dots\}$;

- Given the sets A, B their *Cartesian product* is

$$A \times B = \{(a, b) \mid a \in A \ \wedge \ b \in B\}.$$

We will write A^n for $\underbrace{A \times A \times A \times \dots \times A}_{n \text{ times}}$. Thus, we have $A^1 = A$ and $A^{n+1} = A \times A^n$.

- A (binary) *relation* or *predicate* is $r \subseteq A \times B$.

- A *(partial) function* $f : A \to B$ is a special relation $f \subseteq A \times B$ such that if $(a, b_1), (a, b_2) \in f$ then $b_1 = b_2$. Following the standard convention, we will write $f(a) = b$ instead of $(a, b) \in f$

  - the *domain* of f is $dom(f) = \{a \mid \exists b \in B. \ f(a) = b\}$;

  - we write $f(a) \downarrow$ for $a \in dom(f)$ and $f(a) \uparrow$ for $a \notin dom(f)$;

- Given a set A we indicate with $|A|$ its *cardinality* (intuitively, the number of elements of A, but the notion extends to infinite sets). Given the sets A and B we have

  - $|A| = |B|$ if there exists a bijective function $f : A \to B$;

  - $|A| \leqslant |B|$ if there exists an injective function $f : A \to B$ or equivalently[1] a surjective function $g : B \to A$.

Observe that if $A \subseteq B$ then $|A| \leqslant |B|$ as witnessed by the inclusion, which is an injective function

$$i : \quad A \to B$$
$$a \mapsto a$$

injective function

surjective function

---
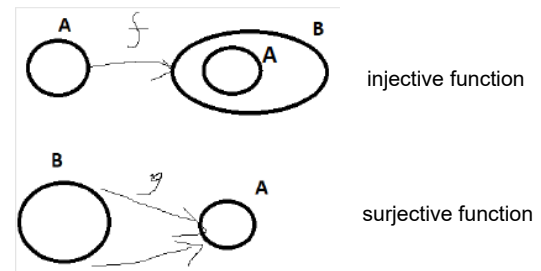
[1]Strictly speaking, the equivalence requires the axiom of choice.

- We say that A is *countable* or *denumerable* when $|A| \leqslant |\mathbb{N}|$, i.e., there is a surjective function $f : \mathbb{N} \to A$. Note that, when this is the case, we can list (enumerate, whence the name) the elements of A as

$$
\begin{array}{cccc}
f(0) & f(1) & f(2) & \dots \\
a_0 & a_1 & a_2 & \dots
\end{array}
$$

- When A, B are countable then $A \times B$ is countable.

  Idea of the proof:

  - Since A and B are countable, we can consider the corresponding enumerations

$$
\begin{array}{cccc}
A & a_0 & a_1 & a_2 \\
B & b_0 & b_1 & b_2
\end{array}
$$

  and place the elements of $A \times B$ in a matrix

	b_0	b_1	b_2
a_0	(a_0, b_0)	(a_0, b_1)	(a_0, b_2)
a_1	(a_1, b_0)	(a_1, b_1)	(a_1, b_2)
a_2	(a_2, b_0)	(a_2, b_1)	(a_2, b_2)

  so that they can be enumerated along the diagonals as follows: $(a_0, b_0), (a_0, b_1), (a_1, b_0), (a_0, b_2), (a_1, b_1), (a_2, b_0), \dots$ (this is referred to as *dove tail* enumeration)

- A countable union of countable sets is countable: if $\{A_i\}_{i \in \mathbb{N}}$ is a collection of countable sets then $\bigcup_{i \in \mathbb{N}} A_i$ is countable.

2.2.2. Existence of non-computable functions. Let us consider some fixed computational model satisfying the assumptions in §2.1. We want to show that there are functions which are not computable in such a model.

We focus on unary functions over the natural numbers. Let

$$
\mathcal{F} = \{ f \mid f : \mathbb{N} \to \mathbb{N} \}
$$

be the set of all the *(partial) unary functions* on $\mathbb{N}$.

Let $\mathcal{A}$ be the set of all algorithms in our fixed computational model. Every algorithm $A \in \mathcal{A}$ computes a function $f_A : \mathbb{N} \to \mathbb{N}$ and a function is said to be computable in our model if there exists an algorithm that computes it. Hence the set $\mathcal{F}_{\mathcal{A}}$ set of computable functions in the given computational model is

$$
\mathcal{F}_{\mathcal{A}} = \{ f_A \mid A \in \mathcal{A} \}
$$

Certainly $\mathcal{F}_{\mathcal{A}} \subseteq \mathcal{F}$. But, is the inclusion strict (i.e., is there a non-computable function)?

The answer is yes. Basically for combinatorial reasons: the algorithms are too few to compute all the functions.

In fact, an algorithm $A \in \mathcal{A}$, by assumption a) in the characteristics of an algorithm, will be a finite sequence of instructions taken from some instruction set I. Moreover,

by assumption h), I must be finite. Hence:

$$\mathcal{A} \subseteq \bigcup_{i \in \mathbb{N}} I^n$$

Since a countable union of finite (hence countable) sets is countable, we have

$$|\mathcal{A}| \leqslant |\bigcup_{n \in \mathbb{N}} I^n| \leqslant |\mathbb{N}|$$

and since the function

$$\mathcal{A} \to \mathcal{F}_{\mathcal{A}}$$
$$A \mapsto f_A$$

is surjective by definition, we have that

$$|\mathcal{F}_{\mathcal{A}}| \leqslant |\mathcal{A}| \leqslant |\mathbb{N}|$$

On the other hand the set of all functions, $\mathcal{F}$, is not countable. Let $\mathcal{T}$ the subset of $\mathcal{F}$ consisting of the total functions $\mathcal{T} = \{f \mid f \in \mathcal{F} \ \wedge \ dom(f) = \mathbb{N}\}$. We show that

$$|\mathcal{F}| \geqslant |\mathcal{T}| > |\mathbb{N}|$$

We prove that $|\mathcal{T}| > |\mathbb{N}|$ by contradiction. Let us suppose that $\mathcal{T}$ is countable. Then we can consider an enumeration $f_0, f_1, f_2, \ldots$ of $\mathcal{F}$ as in the following matrix

	f_0	f_1	f_2
0	$f_0(0)$	$f_1(0)$	$f_2(0)$
1	$f_1(0)$	$f_1(1)$	$f_1(2)$
2	$f_2(0)$	$f_2(1)$	$f_2(2)$

and build a function d, by considering and systematically changing diagonal values

$$d : \mathbb{N} \to \mathbb{N}$$
$$n \mapsto f_n(n) + 1$$

We can observe that

- d is total, by definition;

- $d \neq f_n$ for all $n \in \mathbb{N}$, since $d(n) = f_n(n) + 1 \neq f_n(n)$.

This is absurd, since $f_0, f_1, f_2, \ldots$ is an enumeration of all the total functions.

Summing up

$$\mathcal{F}_{\mathcal{A}} \subseteq \mathcal{F}$$
$$|\mathcal{F}_{\mathcal{A}}| \leqslant |\mathbb{N}| < |\mathcal{T}| = |\mathcal{F}|$$

we get $\mathcal{F}_{\mathcal{A}} \subset \mathcal{F}$, as desired.

Note that the set of non-computable functions is not countable

$$|\mathcal{F} \backslash \mathcal{F}_{\mathcal{A}}| > |\mathbb{N}|$$

In fact, $\mathcal{F} = \mathcal{F}_{\mathcal{A}} \cup (\mathcal{F} \backslash \mathcal{F}_{\mathcal{A}})$. Thus, if it were $|\mathcal{F} \backslash \mathcal{F}_{\mathcal{A}}| \leqslant |\mathbb{N}|$, we would have had $|\mathcal{F}| \leqslant |\mathbb{N}|$ because the union of countable sets is countable.

We conclude that

(1) no computational model can compute all functions;

(2) there are more non-computable than computable functions.

CHAPTER 3

URM computability

3.1. Which model?

To give a formal notion of computability we must choose a concrete model of computation that induces a class of algorithms and thus a corresponding class of computable functions. Despite the fact that we focus on an abstract ideal model, there are still a lot of possibilities. Many models have been considered in the literature:

(1) Turing machine (Turing, 1936)

(2) λ-calculus (Church, 1930)

(3) Partial recursive functions (Godel-Kleene 1930)

(4) Canonical deductive systems (Post, 1943)

(5) Markov systems (Markov, 1951)

(6) Unlimited register machine (URM) (Shepherdson - Sturgis, 1963)

(7) ...

In principle, each computational model determines a class of computable functions. We may be concerned thinking that the developed theory is valid only for a specific model chosen. Actually, it can be verified that the class of computable functions for all models cited (and for all "sufficiently expressive" models considered in literature) is always the same. This leads to the so-called Church-Turing thesis:

Church-Turing thesis: *A function is computable by an effective procedure (i.e., in a finitary computational model, obeying the conditions (a)-(e) from the chapter before) if and only if it is computable by a Turing machine.*

This means that the notion of "computable function" is robust (i.e. independent of the specific computational model), and we can choose our favorite one for developing our theory.

REMARK 3.1. The *Church-Turing thesis* is called a thesis and not a theorem due of its informal nature. It cannot be proved since it refers to an informal notion of effective procedure, but is supported only by evidence: many computational models have been considered and all respect the thesis (e.g. Yuri Gurevich, argues that it should be proved on the basis of a formal axiomatization of conditions (a) - (e)).

Sometimes we resort to the Church-Turing thesis to shorten the proof that a certain function is computable. However this should only be done when it is not strictly

necessary, i.e. when it could be replaced by a formal proof (and providing all the details could hide the intuitive idea under a bunch of technicalities).

3.2. URM (Unlimited register machine)

We will formalise the notion of **computable function** by using an **abstract machine** called **URM-machine** (Unlimited Register Machine), which is an abstraction of a computer based on the Von Neumann's model. It is characterized by

- **unbounded memory** that consists of a infinite sequence of **registers**, each of which can store a natural number

R_1	R_2	...	R_n	...
r_1	r_2	...	r_n	...

  the n-th register is indicated by R_n, its content by r_n

  the sequence $(r_1, r_2, \ldots, r_n, \ldots) \in \mathbb{N}^\omega$ is called **configuration** of the URM;

- a **computing agent** capable of executing an URM program;

- a **URM program**, i.e. a finite sequence of instructions $I_1, I_2, \ldots, I_s$ that can "locally" alter the configuration of the URM.

Program instructions can be the following

- **zero** $Z(n)$
  sets the content of the register R_n to zero: $r_n \leftarrow 0$;

- **successor** $S(n)$
  increments by 1 the content of register R_n: $r_n \leftarrow r_n + 1$;

- **transfer** $T(m, n)$
  transfers the content of the register R_m in the register R_n, R_m stays untouched: $r_n \leftarrow r_m$.

The above are often referred to as *arithmetic instructions*. They are characterised by the fact that the instruction to be executed in the next step is the one following the current instruction in the program.

Then last instruction is

- **conditional jump** $J(m, n, t)$ compares the content of the registers R_m and R_n

  - if $r_m = r_n$ it jumps to the t-th instruction;

  - otherwise, it continues with the next instruction.

EXAMPLE 3.2. An example of program is the following:

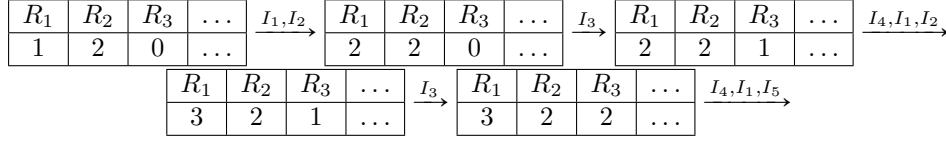I_1: J(2,3,5)
I_2: S(1)
I_3: S(3)
I_4: J(1,1,1) // unconditional jump

Computation

1) starts from:
- initial configuration of registers
- executes I1

2) terminates if the instruction to be executed next does not exist:
- last instruction
- jump out of the program

Disregard what this program computes for the moment. The computation starting from the configuration below is:

R_1	R_2	R_3	$\ldots$
1	2	0	$\ldots$

$\xrightarrow{I_1, I_2}$

R_1	R_2	R_3	$\ldots$
	2	2	0	$\ldots$

$\xrightarrow{I_3}$

R_1	R_2	R_3	$\ldots$
	2	2	1	$\ldots$

$\xrightarrow{I_4, I_1, I_2}$

R_1	R_2	R_3	$\ldots$
3	2	1	$\ldots$

$\xrightarrow{I_3}$

R_1	R_2	R_3	$\ldots$
3	2	2	$\ldots$

$\xrightarrow{I_4, I_1, I_5}$

The **state** of the URM machine in which it executes a program $P = I_1 \ldots I_s$ is given by a pair $\langle c, t \rangle$ that consists of a

- *register configuration c*
  a total function $c : \mathbb{N} \to \mathbb{N}$ such that $c(n)$ is the content of register R_n;

- *program counter t*, i.e., index of the current instruction.

An *operational semantics* can easily be defined via a set of deduction rules axiomatising the state transitions $\langle c, t \rangle \to \langle c', t' \rangle$. However we do not need this level of formality, and we will rely on an informal description of program execution.

REMARK 3.3. A computation might **not terminate**! Consider for instance the program

$$
\begin{array}{ll}
I_1: & S(1) \\
I_2: & J(1,1,1) \quad \text{LOOP}
\end{array}
$$

Then the computation will not terminate. For instance

R_1	R_2	R_3	$\ldots$
0	0	0	$\ldots$

$\xrightarrow{I_1, I_2}$

R_1	R_2	R_3	$\ldots$
1	0	0	$\ldots$

$\xrightarrow{I_1, I_2}$

R_1	R_2	R_3	$\ldots$
2	0	0	$\ldots$

$\xrightarrow{\ldots}$
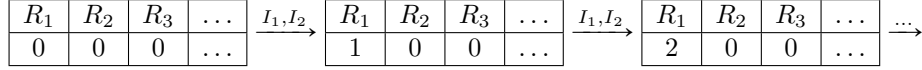
NOTATION 3.4. Let P be an URM program, and $(a_1, a_2, a_3, \ldots) \in \mathbb{N}^\omega$ a sequence of natural numbers. We indicate the computation of P starting from the initial configuration by $P(a_1, a_2, \ldots)$:

R_1	R_2	R_3	$\ldots$
a_1	a_2	a_3	$\ldots$

and

- $P(a_1, a_2, \ldots) \downarrow$ if the computation **halts**.

- $P(a_1, a_2, \ldots) \uparrow$ if the computation **never halts** (i.e., it **diverges**).

We will work on computations that start from an initial configuration where only a **finite number of registers contain a non-zero value** for the majority of the time (almost always for obvious reasons of input finiteness). Hence; given $a_1, a_2, \ldots, a_k \in \mathbb{N}$ we will write

$$P(a_1, \ldots, a_k) \text{ for } P(a_1, \ldots, a_k, 0, \ldots, 0)$$

The notation extends to $P(a_1, \ldots, a_k) \downarrow$ or $P(a_1, \ldots, a_k) \uparrow$.

3.3. URM-computable functions

Let $f : \mathbb{N}^k \to \mathbb{N}$ be a partial function. What does it mean for f to be computable by an URM machine?

Intuitively, it means that there exists a program P such that for each $(a_1, \ldots, a_k) \in \mathbb{N}^k$, $P(a_1, \ldots, a_k)$ computes the value of f, i.e. when $(a_1, \ldots, a_k) \in dom(f)$, P terminates and outputs $f(a_1, \ldots, a_k)$. Instead, P does not terminate if $(a_1, \ldots, a_k) \notin dom(f)$.

A doubt could concern where the output is stored. We conventionally decide that the output will be in the first register R_1 (hence at the end of the computation, the content of any register other than the first one will be irrelevant). For this reason we introduce the following notation

NOTATION 3.5. Let P be a program and $(a_1, \ldots, a_k) \in \mathbb{N}^k$, we write $P(a_1, \ldots, a_k) \downarrow a$ if $P(a_1, \ldots, a_k) \downarrow$ and the final configuration contains a in R_1

DEFINITION 3.6 (URM-computable function). A function $f : \mathbb{N}^k \to \mathbb{N}$ is said to be **URM-computable** if there exists a URM program P such that for all $(a_1, \ldots, a_k) \in \mathbb{N}^k$ and $a \in \mathbb{N}$, $P(a_1, \ldots, a_k) \downarrow$ if and only if $(a_1, \ldots, a_k) \in dom(f)$ and $f(a_1, \ldots, a_k) = a$.

In this case we say that P computes f.

We denote by $\mathcal{C}$ the class of all URM-computable functions and by $\mathcal{C}^{(k)}$ the class of the k-ary URM-computable functions. Therefore we have $\mathcal{C} = \bigcup_{k \geqslant 1} \mathcal{C}^{(k)}$.

3.4. Examples of URM-computable functions

We next list some URM-computable functions, providing the corresponding programs.

(1) $f : \mathbb{N}^2 \to \mathbb{N}$
   $f(x, y) = x + y$

   I_1: J(2,3,5)   --> alternative signature: J(2,3,STOP) [more readable]
   I_2: S(1)
   I_3: S(3)
   I_4: J(1,1,1)   // unconditional jump   --> J(1,1,LOOP)

R_1	R_2	R_3	...
x	y	0	...

   Idea: Increment R_1 and R_3 until R_2 and R_3 contain the same value. This results in adding to R_1 the content of R_2.

(2) $f : \mathbb{N} \to \mathbb{N}$
   $$f(x) = x \dot{-} 1 = \begin{cases} 0 & \text{(if)} \, x = 0 \\ x - 1 & x > 0 \end{cases}$$

R_1	R_2	R_3	...
x	0	0	...

Idea: if $x = 0$ it trivially terminates; if $x > 0$, it keeps a value $k - 1$ in R_2 and k in R_5, with $k > 1$ ascending until $R_3 = x$, at that point $R_2 = x - 1$.

Here's the program

I_1: J(1,3,8)   x=0? Yes, we increment it to move on
I_2: S(3)
I_3: J(1,3,7)
I_4: S(2)
I_5: S(3)
I_6: J(1,1,3)
I_7: T(2,1)

(3) $f : \mathbb{N} \to \mathbb{N}$

$$f(x) = \begin{cases} \dfrac{x}{2} & \text{if } x \text{ even} \\ \uparrow & \text{otherwise} \end{cases}$$

Idea: Store an increasing even number in R_2 and store its' half in R_3.

R_1	R_2	R_3	$\ldots$
x	$2k$	k	$\ldots$

I_1: J(1,2,6)
I_2: S(2)
I_3: S(2)
I_4: S(3)
I_5: J(1,1,1)
I_6: T(3,1)

3.5. Function computed by a program

Given a program P, for some fixed number $k \geqslant 1$ of parameters, there exists a unique **function computed by** P that we denote by $f_P^{(k)} : \mathbb{N}^k \to \mathbb{N}$ defined by:

$$f_P^{(k)}(a_1, \ldots, a_k) = \begin{cases} a & \text{if } P(a_1, \ldots, a_k) \downarrow a \\ \uparrow & \text{if } P(a_1, \ldots, a_k) \uparrow \end{cases}$$

REMARK 3.7. The same function can be computed by different programs, for the following two reasons

- we can add useless instructions to a program (dead code, $T(n,n)$, ...)

- the same function can be computed via different algorithms (e.g., for sorting we have quicksort, mergesort, heapsort, etc.)

A function can be computed either by no program or by infinitely many programs.

3.6. Exercises

EXERCISE 3.8 (Reduced URM). Let URM^- be the class of URM machines without transfer instruction. Indicate by $\mathcal{C}^-$ the class of functions that can be computed by URM^- machines. How does $\mathcal{C}^-$ compare to $\mathcal{C}$?

PROOF. We show that $\mathcal{C}^- = \mathcal{C}$

Obviously $\mathcal{C}^- \subseteq \mathcal{C}$.

Let us prove that $\mathcal{C} \subseteq \mathcal{C}^-$. Informally an instruction $T(m, n)$ at the t step can be replaced with the following subroutine

$$
\begin{aligned}
&\qquad\qquad Z(n) \\
&\text{LOOP}: \quad J(m, n, END) \\
&\qquad\qquad S(n) \\
&\qquad\qquad J(1, 1, LOOP) \\
&\text{END}:
\end{aligned}
$$

We prove it formally. Given $f \in \mathcal{C}$, $f : \mathbb{N}^k \to \mathbb{N}$, there is an URM program P such that $f_P^{(k)} = f$. We show that the program P can be transformed into a URM^- program P' of the reduced URM machine such that $f_{P'}^{(k)} = f_P^{(k)}$.

We proceed by induction on the number h of transfer instructions T in P. Observe that we can assume, without loss of generality, that when a program halts it does so at the index of the last instruction plus one.

(1) $h = 0$ trivial, we can take $P = P'$ since P is already a URM^--program.

(2) $h \to h + 1$: Assume that P contains $h + 1$ transfer instructions. Hence it has the shape

$$
\begin{aligned}
&I_1: \ldots \\
&\ldots \\
&I_t : T(m, n) \\
&\ldots \\
&I_s : \ldots
\end{aligned}
$$

We can transform it into the program P'', where the instruction T is replaced by a jump to the subroutine:

$$
\begin{aligned}
&I_1: \qquad \ldots \\
&\ldots \\
&I_t : \qquad J(1, 1, s + 2) \quad // \text{ jump to the subroutine} \\
\\
&\ldots \\
&I_s : \qquad \ldots \\
&I_{s+1} : \quad J(1,1,s+6) \qquad // \text{ jump to the end} \\
&I_{s+2}: \quad Z(n) \\
&I_{s+3}: \quad J(m, n, t + 1) \quad // \text{ back to the successor of the } T\text{-instruction} \\
&I_{s+4}: \quad S(n) \\
&I_{s+5}: \quad J(1, 1, s + 3)
\end{aligned}
$$

Note that $f_{P''}^{(h)} = f_P^{(h)}$. Moreover program P'' includes h instructions T and therefore, by inductive hypothesis, there exists a URM^- program P' such that $f_{P'}^{(h)} = f_{P''}^{(h)}$. Then $f_{P'}^{(h)} = f_{P''}^{(h)} = f_P^{(h)}$ and P' is the desired program.

$\square$

EXERCISE 3.9 (URM with swap instructions). Let URM^S be the model obtained by removing the transfer instruction and inserting a swap instruction $T_S(m, n)$ which exchanges the contents of registers m and n. Let $\mathcal{C}^S$ be the corresponding class of computable functions. How do the classes $\mathcal{C}$ and $\mathcal{C}^S$ relate?

PROOF. ($\mathcal{C} \subseteq \mathcal{C}^S$) We already know that $\mathcal{C} \subseteq \mathcal{C}^R$ by the previous exercise and therefore, since $\mathcal{C}^R \subseteq \mathcal{C}^S$, the desired inclusion follows.

($\mathcal{C}^S \subseteq \mathcal{C}$) First observe that the swap instruction $T_S(m, n)$ can be encoded in the URM machine by means of the routine:

$$T(n, i)$$
$$T(m, n)$$
$$T(i, m)$$

where i is a "new" register, i.e., a register not used by the program.

More formally, let $f \in \mathcal{C}^S$, $f : \mathbb{N} \to \mathbb{N}$. Then there exists a URM^S program P s.t. $f_P^{(k)} = f$. Let us proceed by induction on the number of swap instructions h.

- ($h = 0$) the program is already a URM program. Therefore wwe can take $P' = P$.

- ($h \to h + 1$) Assume that P contains $h + 1$ swap instructions.

  Let i be a register not used by P (observe that it can be found by just inspecting the program text). Let t be the index of a swap instructions. As in the previous exercise, replace such instruction by a jump

  $$I_t : J(1, 1, SUB)$$

  to a subroutine encoding the swap. Let P'' be the program obtained in this way. Since it has only h swap instructions, by inductive hypothesis there is P' URM such that $f_{P'}^{(k)} = f_{P''}^{(k)} = f_P^{(k)}$, and we are done.

  Observe that strictly speaking the proof above is not working properly!

  In fact the program P'' obtained from P replacing a swap instruction will indeed have h swap instructions but it possibly contains also some transfer instructions, hence it is not a URM^S program.

  We can easily solve the issue by proving the following stronger statement: given a program P that uses both URM instructions and URM^S instructions, there is a URM program P' such that $f_P^{(k)} = f_{P'}^{(k)}$.

  The proof remains essentially the same but the inductive case now works smoothly and we conclude that $\mathcal{C}^S \subseteq \mathcal{C}$.

Therefore we deduce $\mathcal{C}^S = \mathcal{C}$, as desired. □

EXERCISE 3.10 (URM without jump instructions). Consider an URM machine without jump instructions $J(m, n, t)$ and call it URM^{nj}. Let $\mathcal{C}^{nj}$ be the corresponding class of computable functions. How does this class relate to $\mathcal{C}$?

(Handwritten margin notes:)

I_1

$I_t \quad J(1,1, S+2)$

I_s

$I_{s+1} \quad J(1,1, S+5)$

$I_{s+2} \quad T(m,i)$

$I_{s+3} \quad T(m,m)$

$I_{s+4} \quad T(i,m)$

$I_{s+5} \quad J(1,1, t+1)$

PROOF. Clearly $\mathcal{C}^{nj} \subseteq \mathcal{C}$ and the inclusion is strict since, $f : \mathbb{N} \to \mathbb{N}$ with $f(x) \uparrow \forall x$ is computable in URM, but it is not computable in URM ^{nj}. In fact, all functions in $\mathcal{C}^{nj}$ are total since programs without jump instructions always terminate.

We can characterise precisely the (unary) functions in $\mathcal{C}^{nj}$. They are of the shape:

- $f(x) = c$ without the jump instruction the program can just put 0 or add +1 in register so the only 2 operations that the program P could do are:
  - operation without use the input (c) --> at certain point I can cancel the register with Z(n)
- $f(x) = x + c$ - operation using the input x --> ex: S(n) --> add content to the register (input + 1 + 1 + ...)

where c is a constant in $\mathbb{N}$.

This can be proved as follows. Denote by $r_1(h, x)$ the content of register R_1 after h steps starting from an initial configuration where R_1 is x and the other registers contain 0.

We show by induction on h that after h execution steps $r_1(h, x)$ is equal to $x + c$ or to c for some suitable constant $c \in \mathbb{N}$.

- Case $h = 0$:
  We have $r_1(0, x) = x$, which is fine, with $c = 0$.

- Case $h \to h + 1$:
  We know $r_1(h, x) = x + c$ or $r_1(h, x) = c$ by inductive hypothesis. The next instruction can be of three shapes:

  - $Z(n)$
    If $n = 1$, $r_1(h + 1, x) = 0$, otherwise $r_1(h + 1, x) = r_1(h, x)$, and we conclude by inductive hypothesis.

  - $S(n)$
    If $n = 1$ we have that $r_1(h + 1, x) = r_1(h, x) + 1$ which, by inductive hypothesis, is fine. Otherwise, $r_1(h + 1, x) = r_1(h, x)$ and, again, we conclude by induction hypothesis.

  - $T(m, n)$
    When $n > 1$ or $n = m = 1$ then $r_1(h + 1, x) = r_1(h, x)$ and we conclude by inductive hypothesis. Otherwise, if $n = 1$, $m > 1$ we do know nothing about the content of $r_1(h + 1, x)$. We are stuck . . .

  The problem can be solved by observing that register 1 has nothing special and the same result can be proved for all registers. More precisely, denote by $r_n(h, x)$ the content of register R_n after h steps starting from an initial configuration where R_1 is x and the other registers contain 0. Then one can show that $r_n(h, x)$ contains either c or $x + c$ for a suitable constant c. In this case the proof goes smoothly.

  $\square$

CHAPTER 4

Decidable Predicates

In mathematics we often want to establish **properties**. For example, consider the property "*m is a divisor of n*". We can view it as a relation

$$div \subseteq \mathbb{N} \times \mathbb{N}$$
$$div = \{(m, k \cdot m) \mid m \in \mathbb{N}, k \in \mathbb{N}\}$$

We can also view div as a function

$$div : \mathbb{N} \times \mathbb{N} \to \{true, false\}$$
$$div = \begin{cases} true & \text{if } m \text{ is a divisor of } n \\ false & \text{otherwise} \end{cases}$$

In the setting of computability theory one normally uses the term **predicates**.

Thus a **k-ary predicate** on $\mathbb{N}$ indicated with $Q(x_1, \ldots, x_k)$ is a property that can be true or false, formally we can see it as

- a function $Q : \mathbb{N}^k \to \{true, false\}$;

- a set $Q \subseteq \mathbb{N}^k$.

We write $Q(x_1, \ldots, x_k)$ to denote $(x_1, \ldots, x_k) \in Q$ or $Q(x_1, \ldots, x_k) = true$

When is Q computable? When there exists a URM such that given a k-tuple $(x_1, \ldots, x_k)$ in input, it returns $true$ if $Q(x_1, \ldots, x_k)$ and $false$ otherwise.

To represent $true$ and $false$ we conventionally use values 1 and 0.

DEFINITION 4.1 (decidable predicate). A predicate $Q \subseteq \mathbb{N}^k$ is said to be **decidable** if its **characteristic function**

$$\mathcal{X}_Q(x_1, \ldots, x_k) = \begin{cases} 1 & \text{if } Q(x_1, \ldots, x_k) \\ 0 & \text{otherwise} \end{cases}$$

is (URM) computable.

REMARK 4.2. $\mathcal{X}_Q$ is a **total** function (dealing with decidability of predicates, involves only total functions).

19

4.1. Examples of decidable predicates

(1) Equality

$Q \subseteq \mathbb{N}^2$, $Q(x,y) \equiv$ "$x = y$"

The characteristic function

$$\mathcal{X}_Q(x,y) = \begin{cases} 1 & \text{if } x = y \\ 0 & \text{otherwise} \end{cases}$$

is computed, for instance, by the program

	I_1	J(1,2,3)	// J(1,2,TRUE)
FALSE:	I_2	J(1,1,4)	// J(1,1,RES)
TRUE:	I_3	S(3)	
RES:	I_4	T(3,1)	

(2) $Q(x) \equiv$ "x is even"

EVEN:	I_1	J(1,2,6)	// J(1,2,YES)
	I_2	S(2)	
ODD:	I_3	J(1,2,7)	// J(1,2,NO)
	I_4	S(2)	
	I_5	J(1,1,1)	// J(1,1,EVEN)
YES:	I_6	S(3)	
NO:	I_7	T(3,1)	

| x | k | r | in memory where k is a growing index and r is the result.

(3) $Q(x,y) \equiv$ "$x \leqslant y$"

We can increment both x and y until either $x + k = y$ and thus $x \leqslant y$ or $y + k = x$ and thus $x > y$.

	T(1,3)	
	T(2,4)	
LOOP:	J(2,3,SI)	// x+k=y?
	J(1,4,NO)	// y+k=x?
	S(3)	
	S(4)	
	J(1,1,LOOP)	
SI:	S(5)	
NO:	T(5,1)	

Memory: | x | y | $x + k$ | $y + k$ | r | where r is the result.

Another approach is to increment a register starting from 0. If we reach x first then $x \leqslant y$, otherwise $x > y$.

LOOP:	J(1,3,SI)
	J(2,3,NO)
	S(3)
	J(1,1,LOOP)
SI:	S(4)
NO:	T(4,1)

| $x + k$ | y | k | r | where r is the result.

(4) $div(x, y)$ with $x \neq 0$

```
LOOP:   J(2,3,SI)
        Z(4)              // sum x to R₂
ADDX:   J(1,4,LOOP)
        J(2,3,NO)         // kx + h = y?
        S(3)
        S(4)
        J(1,1,ADDX)
SI:     S(5)
NO:     T(5,1)
```

| x | y | $kx + h$ | h | r | where r is the result.

Computability on other domains

Since the URM is confined to manipulate natural numbers, our definition of computability concerns only functions and predicates over $\mathbb{N}$.

The concept of computability can be extended to other domains by resorting to a notion of effective encoding.

Suppose that we are interested in computability on a domain D of objects. Can our notion of computability extend to this domain?

One of the necessary conditions is the possibility of encoding the elements of D as natural numbers. Suppose there exists $\alpha : D \to \mathbb{N}$, which is bijective and that α, α^{-1} are "effective". We don't have a formal notion of effectiveness.

The domain D must be countable. For example, take the strings over some alphabet Σ, $D = \Sigma^*$. The set of rational numbers $\mathbb{Q}$ is also countable, and so is the set of integers $\mathbb{Z}$, while D cannot be $\mathbb{R}$ or A^ω (streams).

Once an encoding is fixed we can use it for defining URM-computability computability on the domain D.

DEFINITION 5.1 (Computable function on generic domain). Given $f : D \to D$, we say that it is **computable** if $f^* = \alpha \circ f \circ \alpha^{-1}$

$$
\begin{array}{ccc}
D & \xrightarrow{\ f\ } & D \\
{\scriptstyle \alpha^{-1}}\big\uparrow & & \big\downarrow{\scriptstyle \alpha} \\
\mathbb{N} & \xrightarrow[\ f^*\ ]{} & \mathbb{N}
\end{array}
$$

is URM-computable.

We will see that if α is effective, its inverse is also effective.

EXAMPLE 5.2 (Computability on the integers). Assume we want to define computability over the integers $\mathbb{Z}$. We need an encoding $\alpha : \mathbb{Z} \to \mathbb{N}$. It can be defined in several ways. One possibility is

$$
\alpha(z) = \begin{cases} 2z & z \geqslant 0 \\ -2z - 1 & z < 0 \end{cases}
$$

$\mathbb{N}$	0	1	2	3	4	5	6	....
	0	-1	1	-2	2	-3	3	

which is an effective function with inverse

$$
\alpha^{-1}(n) = \begin{cases} \dfrac{n}{2} & n \text{ is even} \\ -\dfrac{(n+1)}{2} & n \text{ is odd} \end{cases}
$$

Consider the function

$$f(z) = |z|.$$

It is computable if $f^* = \alpha \circ f \circ \alpha^{-1}$ is URM-computable. We have

$$f^*(n) = (\alpha \circ f \circ \alpha^{-1})(n)$$

$$= \begin{cases} (\alpha \circ f)\left(\dfrac{n}{2}\right) & n \text{ even} \\ (\alpha \circ f)\left(-\dfrac{n+1}{2}\right) & \text{otherwise} \end{cases}$$

$$= \begin{cases} \alpha\left(\dfrac{n}{2}\right) & n \text{ even} \\ \alpha\left(\dfrac{n+1}{2}\right) & \text{otherwise} \end{cases}$$

$$= \begin{cases} n & n \text{ even} \\ n+1 & \text{otherwise} \end{cases}$$

that is URM-computable, so f is computable.
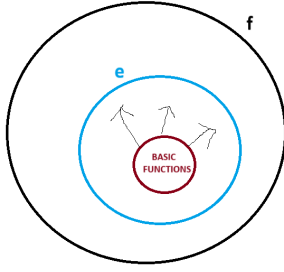
Generation of computable functions

The aim here is to provide a way of proving that certain functions are computable by arguing that they are combinations of simpler functions that are known to be computable.

This amount to showing that there are operations op that take functions f_1, f_2 and compose them producing $op(f_1, f_2)$ in a way that if $f_1, f_2 \in \mathcal{C}$ then $op(f_1, f_2)$ is still in $\mathcal{C}$.

More precisely we will prove that the $\mathcal{C}$ class is closed with respect to the following operations:

- (generalized) composition
- primitive recursion
- (unbounded) minimization

After this, in order to prove that a function $f : \mathbb{N}^k \to \mathbb{N}$ is computable we have two techniques: write a URM program P that computes f (i.e., such that $f_P^{(k)} = f$), or use the closure theorems of $\mathcal{C}$.

Actually the three operations above are chosen carefully. The long term objective is to show that $\mathcal{C}$ coincides with the class of functions which can be obtained through composition, primitive recursion and minimization, starting from a restricted core of basic functions (**partial recursive functions** of Godel-Kleene).

6.1. Basic computable functions

The following basic functions are URM-computable:

(1) constant zero
$$z : \mathbb{N}^k \to \mathbb{N}$$
$$(x_1, \ldots, x_k) \mapsto 0$$

(2) successor
$$s : \mathbb{N} \to \mathbb{N}$$
$$x \mapsto x + 1$$

(3) projection
$$U_i^k : \mathbb{N}^k \to \mathbb{N}$$
$$(x_1, \ldots, x_k) \mapsto x_i$$

In fact, one immediately sees that these basic functions are computed by simple programs consisting of one arithmetic instruction:

(1) z computed by $Z(1)$;

(2) s computed by $S(1)$;

(3) U_i^k computed by $T(i,1)$.

REMARK 6.1. The identity is just a special projection.

To prove the closure properties we will need to "combine" programs so we need some notation.

NOTATION 6.2. Given a URM program P

- $\rho(P)$ is the **largest register index** used in P

- $l(P)$ is the **number of instructions** in P;

- P is in **standard form** if, for each $J(m,n,t)$ instruction, $t \leqslant l(P) + 1$ (if the program stops it will do so at the instruction $l(P) + 1$).

Considering only programs in standard form is not restrictive, as stated by the following lemma:

LEMMA 6.3. *For each URM program P there exists an equivalent program P' in standard form, i.e. for all k, $f_P^{(k)} = f_{P'}^{(k)}$*

PROOF. It is enough to replace every instruction $J(m,n,t)$ in P such that $t > l(P) + 1$ with $J(m,n,l(P)+1)$ □

Often we will have to **concatenate** programs. Given programs P, Q, their concatenation is obtained by considering P followed by the instructions of Q. Only observe that jump instructions in Q need to be updated (each instruction $J(m,n,t)$ in Q is replaced with $J(m,n,t+l(P))$).

REMARK 6.4. If P and Q are in standard form then PQ is in standard form; moreover $(PQ)R = P(QR)$. We will assume every program is in standard form and we will use concatenation freely.

It will be useful to consider programs which take the input and give the output in arbitrary registers.

Given a program P, we want a program $P[i_1, \ldots, i_k \to h]$ that takes input from $R_{i1}, \ldots, R_{ik}$, without assuming that the remaining the registers are set to 0, and gives back the output in R_h This is easily obtainable with transfer and reset operations to move the contents of registers from $i_1, \ldots, i_k$ to $1, \ldots, k$ and the output from h to 1.

More precisely $P[i_1, \ldots, i_k \to h]$ is as follows:

$T(i_1, 1)$

$\ldots$

$T(i_k, k)$

$Z(k+1)$ // clean the memory only from k+1 register

$\ldots$

$Z(\rho(P))$

P // concatenation

$T(1, l)$

problem: P[2,1 --> 1]

T(2,1)
T(1,2)
P
.
.
.

you want

| x | y | | | → | y | x | |

you get

| y | y | |

if we execute the code x and y overlap each other and
I get y and y value at first and second register
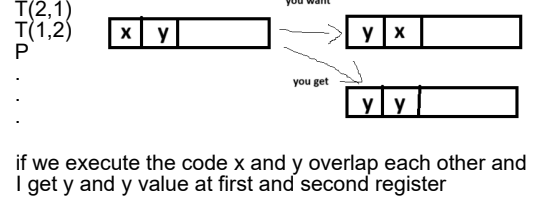
6.2. Generalized composition

DEFINITION 6.5. Given a function $f : \mathbb{N}^k \to \mathbb{N}$ and functions $g_1, \ldots, g_k : \mathbb{N}^n \to \mathbb{N}$ we define the **composition** $h : \mathbb{N}^n \to \mathbb{N}$ by

$$h(\vec{x}) = \begin{cases} f(g_1(\vec{x}), \ldots, g_k(\vec{x})) & \text{if } g_1(\vec{x})\downarrow, \ldots, g_k(\vec{x})\downarrow \text{ and } f(g_1(\vec{x}), \ldots, g_k(\vec{x}))\downarrow \\ \uparrow & \text{otherwise} \end{cases}$$

EXAMPLE 6.6. Consider

undefined function
$=$
$\vee$

$$z(x) = 0 \quad \forall x \qquad \varnothing(x)\uparrow \quad \forall x$$

then

$$z(\varnothing(x))\uparrow \quad \forall x \qquad \text{==> the composition of undefined functions in an undefined function}$$

EXAMPLE 6.7. Consider $\varnothing$ and U_1^2, then

$$U_1^2(x_1, x_2) = x_1 \quad \text{but} \quad U_1^2(x_1, \varnothing(x_2))\uparrow$$

PROPOSITION 6.8. $\mathcal{C}$ *is closed under generalised composition*

PROOF. Let

$$f : \mathbb{N}^k \to \mathbb{N}$$
$$g_1, \ldots, g_k : \mathbb{N}^n \to \mathbb{N}$$

in $\mathcal{C}$, consider the composition

$$h : \mathbb{N}^k \to \mathbb{N}$$
$$\vec{x} \mapsto f(g_1(\vec{x}), \ldots, g_k(\vec{x}))$$

Since $f, g_1, \ldots, g_k \in \mathcal{C}$, we can take $F, G_1, \ldots, G_k$ programs in standard form for them.

Let us consider the largest register index possibly used by the involved programs i.e., $m = max\{\rho(F), \rho(G_1), \ldots \rho(G_k), k, n\}$. Then the registers from $m+1$ on can be used freely without the risk of interferences. The program for the composition can be

1	$\ldots$	m	$m+1$	$\ldots$	$m+n$	$m+n+1$	$\ldots$	$m+n+k$
	$\ldots$		x_1	$\ldots$	x_n	$g_1(\vec{x})$	$\ldots$	$g_k(\vec{x})$

$$T(1, m+1)$$
$$\ldots$$
$$T(n, m+n)$$
$$G_1[m+1, \ldots, m+n \rightarrow m+n+1]$$
$$\ldots$$
$$G_k[m+1, \ldots, m+n \rightarrow m+n+k]$$
$$F[m+n+1, \ldots, m+n+k \rightarrow 1]$$

This allows us to conclude that $h \in \mathcal{C}$.   □

EXAMPLE 6.9. If $f : \mathbb{N}^2 \rightarrow \mathbb{N}$ is computable, then the following are also computable

- $f_1(x, y) = f(y, x);$ // x+y known to be in C

- $f_2(x) = f(x, x);$

- $f_3(x, y, z) = f(x, y).$ // x+y+z = f(f(x,y)+z) known to be in C

REMARK 6.10. On the basis of the results above we can use generalized composition when the g_i are not functions of all the variables or are functions with repetitions.

EXAMPLE 6.11. We know that $f : \mathbb{N}^2 \rightarrow \mathbb{N}$ where $f(x_1, x_2) = x_1 + x_2$ is computable. Using this fact and the closure of $\mathcal{C}$ under generalise composition we can derive that $g : \mathbb{N}^3 \rightarrow \mathbb{N}$ where $g(x_1, x_2, x_3) = x_1 + x_2 + x_3$ is also computable. In fact $g(x_1, x_2, x_3) = f(f(x_1, x_2), x_3) = f(f(U_1^3(\vec{x}), U_2^3(\vec{x})), U_3^3(\vec{x}))$, that is computable.

EXAMPLE 6.12. The following functions are computable

- **constant** $m(\vec{x}) = m$
  $m(\vec{x}) = s(s(\ldots s(z(\vec{x}))))$, i.e., s applied m times;

- **addition** $g(x_1, \ldots, x_k) = x_1 + \cdots + x_k$, as seen before;

- **product by a constant** $f(x) = k \cdot x = g(\underbrace{x, \ldots, x}_{k \text{ times}})$, where g is the function
  at the previous step;

- if $f(x, y)$ is computable, then also $f'(x) = f(x, m)$ is computable. In fact
  $f'(x) = f(x, m) = f(U_1^1(x), m(x))$, that is computable;

- if $f : \mathbb{N} \rightarrow \mathbb{N}$ is total computable, the predicate $Q(x, y) \equiv \text{“} f(x) = y \text{”}$ is
  decidable.

  In fact, we know that

$$\mathcal{X}_{Eq}(x, y) = \begin{cases} 1 & x = y \\ 0 & \text{otherwise} \end{cases}$$

  is computable.

  Therefore $\mathcal{X}_Q(x, y) = \mathcal{X}_{Eq}(f(x), y) = \mathcal{X}_{Eq}(f(U_1^2(x, y)), U_2^2(x, y))$, and
  thus $\mathcal{X}_Q$ is computable.

6.3. Primitive recursion

Recursion is a familiar concept; it allows to define a function specifying its values in terms of other values of the same function (and possibly using other functions already defined).

EXAMPLE 6.13 (Factorial).

$$\begin{cases} 0! = 1 \\ (n+1)! = n! \cdot (n+1) \end{cases}$$

EXAMPLE 6.14 (Fibonacci).

$$\begin{cases} f(0) = 1 \\ f(1) = 1 \\ f(n+2) = f(n) + f(n+1) \end{cases}$$

There are many types of recursion, here we use a very "controlled" version of recursion.

DEFINITION 6.15 (Primitive recursion). Given $f : \mathbb{N}^k \to \mathbb{N}$ and $g : \mathbb{N}^{k+2} \to \mathbb{N}$ functions, we define $h : \mathbb{N}^{k+1} \to \mathbb{N}$ by **primitive recursion** as follows

$$\begin{cases} h(\vec{x}, 0) = f(\vec{x}) \quad \text{{\scriptsize // the second argument of h is used for iteration}} \\ h(\vec{x}, y+1) = g(\vec{x}, y, h(\vec{x}, y)) \quad \text{{\scriptsize // I defined h using h (left and right side of =) --> recursion}} \end{cases}$$

REMARK 6.16. The function h is defined in an equational manner, with h that appears on both sides: it is an implicit definition and it is not obvious that such h exists or that it is unique, but actually it does exist and it is unique. However, a general theory that supports this observation is not trivial.

The argument proceeds as follows

(1) let $\mathbb{N}^n \to \mathbb{N}$ the set of functions on natural numbers with n arguments

(2) we define an operator

$$T : (\mathbb{N}^{k+1} \to \mathbb{N}) \to (\mathbb{N}^{k+1} \to \mathbb{N})$$
$$T(h)(\vec{x}, 0) = f(\vec{x})$$
$$T(h)(\vec{x}, y+1) = g(\vec{x}, y, h(\vec{x}, y))$$

(3) the desired function is a fixed points of T, i.e. h such that $T(h) = h$;

(4) the existence of the fixed point follows from these properties

- $\mathbb{N}^{k+1} \to \mathbb{N}$ is a CPO;

- T is continuous;

- continuous functions over a CPO have a least fixed point.

(5) uniqueness can be proved inductively, showing that if h, h' are fixed points then $h = h'$.

EXAMPLE 6.17. Consider the sum function $h(x, y) = x + y$. It can be defined by primitive recursione as

$$\begin{cases} h(x, 0) = x = f(x) \\ h(x, y + 1) = h(x, y) + 1 = g(h(x, y)) \end{cases}$$

where f is the identity and g is the successor. Both are computable, so the sum is computable by primitive recursion.

PROPOSITION 6.18. *Functions obtained from total functions by*

(1) *generalized composition*

(2) *primitive recursion*

are total.

PROOF.   (1) obvious by definition;

(2) Let $f : \mathbb{N}^k \to \mathbb{N}, g : \mathbb{N}^{k+2} \to \mathbb{N}$ be total functions and define h by primitive recursion.

It can be proved by induction on y that

$$\forall \vec{x} \in \mathbb{N}^k \ (\vec{x}, y) \in dom(h)$$

- ($y = 0$): for all $\vec{x} \in \mathbb{N}^k$, $h(\vec{x}, 0) = f(\vec{x}) \downarrow$;

- ($y \to y+1$): for all $\vec{x} \in \mathbb{N}^k$, $h(\vec{x}, y+1) = g(\vec{x}, y, h(\vec{x}, y)) \downarrow$ by inductive hypothesis.

$\square$

EXAMPLE 6.19. We observe that some functions can be defined by primitive recursion:

- **sum** $x + y$
  $x + 0 = x$
  $x + (y + 1) = (x + y) + 1$

  $h(x, 0) = x$
  $h(x, y + 1) = h(x, y) + 1$

  $f(x) = x$
  $g(x, y, z) = z + 1$

- **product** $x \cdot y$
  $x \cdot 0 = 0$
  $x \cdot (y + 1) = (x \cdot y) + x$

  $h(x, 0) = 0$
  $h(x, y + 1) = h(x, y) + x$

  $f(x) = 0$
  $g(x, y, z) = z + y$

- **factorial** $y!$
  $0! = 1$
  $(y+1)! = y! \cdot (y+1)$

  $h(0) = 1$
  $h(y+1) = h(y) \cdot (y+1)$

  $f(0) = 1$
  $g(y,z) = z \cdot (y+1)$

PROPOSITION 6.20. $\mathcal{C}$ *is closed under primitive recursion.*

PROOF. Let $f : \mathbb{N}^k \to \mathbb{N}$ and $g : \mathbb{N}^{k+2} \to \mathbb{N}$ be computable functions. We want to prove that $h : \mathbb{N}^{k+1} \to \mathbb{N}$ defined through primitive recursion

$$\begin{cases} h(\vec{x}, 0) = f(\vec{x}) \\ h(\vec{x}, y+1) = g(\vec{x}, y, h(\vec{x}, y)) \end{cases}$$

is computable.

Let F, G programs in standard form for f, g. We want a program H for h. We proceed as suggested by the definition.

We start from

| x_1 | $\ldots$ | x_k | y | 0 | $\ldots$ |

we save the parameters and we start to compute $h(\vec{x}, 0)$ using F.

If $y = 0$ we are done, otherwise we save $h(\vec{x}, 0)$ and compute $h(\vec{x}, 1) = g(\vec{x}, 0, h(\vec{x}, 0))$ with G. We do the same for $h(\vec{x}, i)$ until we arrive at $i = y$.

As usual we need registers not used by F and G, $m = max\{\rho(F), \rho(G), k+2\}$ and we build the program for h as follows:

1	$\ldots$	$m+1$	$\ldots$	$m+k$	$m+k+1$	$\ldots$	$m+k+3$	
$\ldots$	$\ldots$	$\ldots$	$\vec{x}$	$\ldots$	i	$h(\vec{x},2)$	y	0

$$T(1, m+1)$$
$$\ldots$$
$$T(k, m+k)$$
$$T(k+1, m+k+3)$$
$$F[m+1, \ldots, m+k \to m+k+2] \qquad \text{// compute } h(\vec{x},0)$$
$$LOOP: \quad J(m+k+1, m+k+3, END) \qquad \text{// i=y?}$$
$$G[m+1, \ldots, m+k+2 \to m+k+2]$$
$$S(m+k+1) \qquad \text{// i = i+1}$$
$$J(1, 1, LOOP)$$
$$END: \quad T(m+k+2, 1)$$

$\square$

OBSERVATION 6.21. We do nothing more than implementing recursion through iteration.

OBSERVATION 6.22. The following functions are computable.

(1) **sum** $x + y$, see above;

(2) **product** $x \cdot y$ see above;

(3) **exponential** x^y

$$
\begin{array}{lll}
x^0 = 1 & h(x,0) = 1 & f(x) = 1 \\
x^{y+1} = x^y \cdot x & h(x, y+1) = h(x,y) \cdot x & g(x,y,z) = z \cdot x
\end{array}
$$

(4) **predecessor** $x \dot{-} 1$

$$
\begin{array}{lll}
0 \dot{-} 1 = 0 & h(0) = 0 & f \equiv \underline{0} \\
(x+1) \dot{-} 1 = x & h(x+1) = x & g(y,z) = y
\end{array}
$$

(5) **subtraction** $x \dot{-} y = \begin{cases} x - y & x \geqslant y \\ 0 & \text{otherwise} \end{cases}$

$$
\begin{array}{ll}
x \dot{-} 0 = x & f(x) = x \\
x \dot{-} (y+1) = (x \dot{-} y) \dot{-} 1 & g(x,y,z) = z \dot{-} 1
\end{array}
$$

(6) **sign** $sg(x) = \begin{cases} 0 & x = 0 \\ 1 & x > 0 \end{cases}$

$$
\begin{array}{ll}
sg(0) = 0 & f \equiv \underline{0} \\
sg(x+1) = 1 & g(y,z) = 1
\end{array}
$$

(7) **complement sign** $\bar{sg}(x) = \begin{cases} 1 & x = 0 \\ 0 & x > 0 \end{cases}$

$\bar{sg}(x) = 1 \dot{-} sg(x)$, composition and (6);

(8) $|x - y| = \begin{cases} x - y & x \geqslant y \\ y - x & x < y \end{cases}$

$|x - y| = (x \dot{-} y) + (y \dot{-} x)$ from (1), (6) and composition;

(9) **factorial** $y!$
$$0! = 1 \quad f \equiv (y+1)! = y! \cdot (y+1) \quad g(y,z) = (y+1) \cdot z$$

(10) **minimum** $min(x,y) = x \dot{-} (x \dot{-} y)$;

(11) **maximum** $max(x,y) = (x \dot{-} y) + y$;

(12) **remainder** $rm(x,y) = \begin{cases} y \mod x & x \neq 0 \\ y & x = 0 \end{cases}$

remainder of the integer division of y by x

$$rm(x,0) = 0$$

$$rm(x, y+1) = \begin{cases} rm(x,y) + 1 & rm(x,y) + 1 \neq x \\ 0 & \text{otherwise} \end{cases}$$

$$= (rm(x,y) + 1) \cdot sg((x \dot{-} 1) \dot{-} rm(x,y))$$

$$f(x) = 0 \quad g(x,y,z) = z * sg(x \dot{-} 1 \dot{-} z)$$

(13) **quotient** $qt(x, y) = y$ div x (convention $qt(0, y) = y$), we define:

$$qt(x, 0) = 0$$

$$qt(x, y + 1) = \begin{cases} qt(x, y) + 1 & rm(x, y) + 1 = x \\ qt(x, y) & \text{otherwise} \end{cases}$$

$$= qt(x, y) + sg((x \div 1) \div rm(x, y))$$

(14)

$$div(x, y) = \begin{cases} 1 & rm(x, y) = 0 \\ 0 & \text{otherwise} \end{cases}$$

$$= \bar{sg}(rm(x, y))$$

COROLLARY 6.23 (Definition by cases). *Given $f_1, \ldots, f_n : \mathbb{N}^k \to \mathbb{N}$ total, computable and $Q_1, \ldots, Q_n \subseteq \mathbb{N}^k$ decidable and mutually exclusive predicates (for each $\vec{x} \in \mathbb{N}^k$, **exactly one** of $Q_1, \ldots, Q_n$ holds) then $f : \mathbb{N}^k \to \mathbb{N}$ is total computable where*

$$f(\vec{x}) = \begin{cases} f_1(\vec{x}) & Q_1(\vec{x}) \\ f_2(\vec{x}) & Q_2(\vec{x}) \\ \ldots \\ f_n(\vec{x}) & Q_n(\vec{x}) \end{cases}$$

PROOF. $f(\vec{x}) = f_1(\vec{x}) \cdot \mathcal{X}_{Q1}(\vec{x}) + \cdots + f_n(\vec{x}) \cdot \mathcal{X}_{Qn}(\vec{x})$

We conclude using the computability of sum and product and the fact that composition preserves computability.   □

6.4. Algebra of decidability

LEMMA 6.24. *Let Q, Q' be decidable predicates. Then also $\neg Q, Q \wedge Q', Q \vee Q'$ are decidable.*

PROOF. It is enough to observe that:

(1) $\mathcal{X}_{\neg Q}(\vec{x}) = \bar{sg}(\mathcal{X}_Q(\vec{x}))$

(2) $\mathcal{X}_{Q \vee Q'}(\vec{x}) = \mathcal{X}_Q(\vec{x}) \cdot \mathcal{X}_{Q'}(\vec{x})$

(3) observe that $Q \wedge Q' = \neg(\neg Q \vee \neg Q')$

□

Recall that $\{\neg, \wedge, \vee\}$ ($\{\neg, \vee\}$ is enough) is a functionally complete set of connectives (it allows to express any function $\{0, 1\}^n \to \{0, 1\}$). We deduce that:

COROLLARY 6.25. *Let $Q_1, \ldots, Q_n \subseteq \mathbb{N}^k$ decidable predicates and let $f : \{0, 1\}^n \to \{0, 1\}$ a function. Let us consider:*

$$\mathcal{X} : \mathbb{N}^k \to \{0, 1\}$$
$$\mathcal{X}(\vec{x}) = f(\mathcal{X}_{Q_1}(\vec{x}), \ldots, \mathcal{X}_{Q_n}(\vec{x}))$$

Then the predicate Q which corresponds to $\mathcal{X}$ is decidable, and therefore $\mathcal{X}$ is computable.

6.5. Bounded sum, product and quantification

DEFINITION 6.26 (Bounded sum and product). Let $f : \mathbb{N}^{k+1} \to \mathbb{N}$ be a total function. Then

- $\sum_{z<y} f(\vec{x}, z)$ is defined by

$$\sum_{z<0} f(\vec{x}, z) = 0$$
$$\sum_{z<y+1} f(\vec{x}, z) = \sum_{z<y} f(\vec{x}, z) + f(\vec{x}, y)$$

- $\prod_{z<y} f(\vec{x}, z)$ is defined by:

$$\prod_{z<1} f(\vec{x}, z) = 1$$
$$\prod_{z<y+1} f(\vec{x}, z) = \prod_{z<y} f(\vec{x}, z) \cdot f(\vec{x}, y)$$

LEMMA 6.27. *If $f : \mathbb{N}^{k+1} \to \mathbb{N}$ is total computable then*

(1) $g(\vec{x}, y) = \sum_{z<y} f(\vec{x}, y)$

(2) $h(\vec{x}, y) = \prod_{z<y} f(\vec{x}, y)$

are total computable.

PROOF. Just note that they are defined by primitive recursion!

$$g(\vec{x}, 0) = 0$$
$$g(\vec{x}, y + 1) = g(\vec{x}, y) + f(\vec{x}, y)$$

and $+, f$ are computable.

Same for 2.   □

Obviously, by closure under composition, the bound can be a total computable function.

Another immediate consequence concerns the decidability of the bounded quantification on the predicates.

LEMMA 6.28. *Let $Q \subseteq \mathbb{N}^{k+1}$ be a decidable predicate, then:*

(1) $Q_1(\vec{x}, y) \equiv \forall z < y.Q(\vec{x}, z)$

(2) $Q_2(\vec{x}, y) \equiv \exists z < y.Q(\vec{x}, z)$

are decidable.

PROOF.   (1) observe that $\mathcal{X}_{Q_1}(\vec{x}, y) = \prod_{z<y} \mathcal{X}_Q(\vec{x}, z)$

(2) observe that $\mathcal{X}_{Q_2}(\vec{x}, y) = sg(\sum_{z<y} \mathcal{X}_Q(\vec{x}, z))$

$\square$

6.6. Bounded minimalisation

Given a total function $f : \mathbb{N}^{k+1} \to \mathbb{N}$, we define a function $h : \mathbb{N}^{k+1} \to \mathbb{N}$ as follows:

$$h(\vec{x}, y) = \mu z < y.f(\vec{x}, z) = \begin{cases} \text{miniumum } z < y \text{ such that } f(\vec{x}, z) = 0 & \text{if it exists} \\ y & \text{otherwise} \end{cases}$$

LEMMA 6.29. *Let* $f : \mathbb{N}^{k+1} \to \mathbb{N}$ *total computable. Then also* $h : \mathbb{N}^k \to \mathbb{N}$ *defined by* $h(\vec{x}, y) = \mu z < y.f(\vec{x}, z)$ *is (total) computable.*

PROOF. We observe that h can be defined as:

$$h(\vec{x}, y) = \sum_{z<y} \prod_{w \leqslant z} sg(f(\vec{x}, w))$$

The product value is 1 on the intervals $[0, z]$ in which $f \neq 0$, i.e. if z_0 is the min $z < y$ where f is null, they're equal to z_0, therefore the external sum counts them.

Alternatively h can be defined directly through primitive recursion:

$$\begin{cases} h(\vec{x}, 0) = 0 \\ h(\vec{x}, y+1) \quad = \begin{cases} h(\vec{x}, y) & h(\vec{x}, y) \neq y \\ \begin{cases} y & f(\vec{x}, y) = 0 \\ y+1 & \text{otherwise} \end{cases} & \text{otherwise} \end{cases} \\ \qquad\qquad = sg(y - h(\vec{x}, y)) \cdot h(\vec{x}, y) + \bar{sg}(y - h(\vec{x}, y))(y + sg(f(\vec{x}, y))) \end{cases}$$

$\square$

LEMMA 6.30. *The following functions are computable:*

a) $D(x) = $ *number of divisors of* x

b) $Pr(x) = \begin{cases} 1 & x \text{ is prime} \\ 0 & \text{otherwise} \end{cases}$ *(x prime is decidable)*

c) $p_x = x$-*th prime number (convention:* $p_0 = 0, p_1 = 2, p_2 = 3 \ldots$*)*

d) $(x)_y = \begin{cases} \text{exponent of } p_y \text{ in the factorization of } x & x, y > 0 \\ 0 & x = 0 \vee y = 0 \end{cases}$

e.g. $72 = 2^3 \cdot 3^2, (72)_1 = 3, (72)_2 = 2, (72)_3 = 0$

PROOF. a) $D(x) = \sum_{y \leqslant x} div(y, x)$

b) $Pr(x)$ is 1 if $x > 1$ and is divided only by 1 and itself

$$Pr(x) = \begin{cases} 1 & D(x) = 2 \\ 0 & \text{otherwise} \end{cases}$$
$$= \bar{sg}(|D(x) - 2|)$$

c) P_x can be defined by primitive recursion

$$P_0 = 0$$
$$P_{x+1} = \mu z \leqslant (P_x! + 1).\bar{sg}(P_z(z) \cdot \mathcal{X}_{z > P_x}(z))$$

Certainly $P_{x+1} \leqslant P_x! + 1$, in fact, call p a prime in the decomposition of $p_x! + 1$, therefore $p \mid p_x! + 1$, so $p > p_x$, otherwise $p \mid p_x!$ and therefore $p \mid 1$. Thus $p_x < p_{x+1} \leqslant p$.

d) Note that

$$(x)_y = \max\ z.p_y^z \mid x =$$
$$= \min\ z.p_y^{z+2} \nmid x$$
$$= \mu z \leqslant x.\neg div((p_y)^{z+1}, x)$$

$\square$

6.6.1. Exercises. Prove that the following functions are computable:

- $\lfloor \sqrt{x} \rfloor$

$$\lfloor \sqrt{x} \rfloor = max\ y \leqslant x.y^2 \leqslant x$$
$$= min\ y \leqslant x.(y+1)^2 > x$$
$$\mu y \leqslant x.((x+1) - (y+1)^2)$$

- $lcm(x, y)$

$$lcm(x, y) = \mu z \leqslant x \cdot y.(x \mid z \wedge y \mid z)$$
$$= \mu z \leqslant x \cdot y \cdot \bar{sg}(div(x, z) \cdot div(y, z))$$

- $GCD(x, y)$

$GCD(x, y) \leqslant min\{x, y\}$ and it can be characterized using the minimum number that can be subtracted to $min\{x, y\}$ to obtain the divisor of x, y

$$GCD(x, y) \leqslant min(x, y) - \mu z$$
$$\leqslant min(x, y).(1 \dot{-} div(min(x, y) - z, x) \cdot div(min(x, y) - z, y))$$

- number of prime divisors of x

$\sum_{z \leqslant x} pr(z) \cdot div(z, x)$

6.7. Encoding of pairs (and n-tuples)

Let's see an encoding in $\mathbb{N}$ of pairs (and n-tuples) of natural numbers that will be useful for some considerations on recursion. Define a **pair encoding** as

$$\pi : \mathbb{N}^2 \to \mathbb{N}$$
$$\pi(x, y) = 2^x(2y + 1) - 1$$

Notice that π is bijective and effective (computable).

The inverse can be characterized in terms of two computable functions that give the first and second component of a natural number n seen as pair:

$$\pi^{-1} : \mathbb{N} \to \mathbb{N}^2$$
$$\pi^{-1}(n) = (\pi_1(n), \pi_2(n))$$

where $\pi_1(n) = (n+1)_1$ and $\pi_2(n) = (\frac{n+1}{2\pi_1(n)} - 1)/2$.

It can be generalized to an encoding of n-tuples:

$$\pi^n : \mathbb{N}^n \to \mathbb{N}$$

defining

$$\pi^2 = \pi$$
$$\pi^{n+1}(\vec{x}, y) = \pi(\pi^n(\vec{x}, y)) \quad \vec{x} \in \mathbb{N}^n, y \in \mathbb{N}$$

and then we can define the projections $\pi_j^n : \mathbb{N} \to \mathbb{N}^n$.

6.7.1. Considerations on recursion. The Fibonacci function is defined by:

$$\begin{cases} fib(0) = fib(1) = 1 \\ fib(n+2) = fib(n) + fib(n+1) \end{cases}$$

This is not exactly a definition by primitive recursion. Given that $f(y+2)$ is defined in terms of $f(y)$ and $f(y+1)$, it does not completely adhere to the primitive recursion schema.

We can show that f is computable by resorting to the encoding of pairs. Define:

$$g : \mathbb{N} \to \mathbb{N}$$
$$g(y) = \pi(f(y), f(y+1))$$

therefore g can be defined by primitive recursion:

$$\begin{cases} g(0) = \pi(f(0), f(1)) = \pi(1, 1) \\ g(y+1) = \pi(f(y+1), f(y+2)) = \pi(\pi_2(g(y)), \pi_1(g(y)) + \pi_2(g(y))) \end{cases}$$

so g is computable, by primitive recursion. Finally, $f(y) = \pi_1(g(y))$ is computable by composition.

In general we could have a function f defined using k previous values

$$\begin{cases} f(0) = c_0 \\ f(k-1) = c_k \\ f(y+k) = h(f(y), \dots, f(y+k-1)) \end{cases}$$

with $h : \mathbb{N}^k \to \mathbb{N}$ computable.

One can proceed like before and define

$$g : \mathbb{N} \to \mathbb{N}$$
$$g(y) = \pi^k(f(y), \dots, f(y+k-1))$$

Then function g can be defined by primitive recursion

$$\begin{cases} g(0) = \pi^k(c_0, \ldots, c_{k-1}) \\ g(y+1) = \pi^k(f(y+1), \ldots, f(y+k-1), f(y+k)) \end{cases}$$

where

$$f(y+1) = \pi_2^k(g(y))$$
$$f(y+k-1) = \pi_k^k(g(y))$$
$$f(y+k) = h(f(y), \ldots, f(y+k-1))$$
$$= h(\pi_1^k(g(y)), \ldots, \pi_k^k(g(y)))$$
$$= \pi^k(\pi_2^k(g(y)), \ldots, \pi_k^k(g(y)), h(\pi_1^k(g(y)), \ldots, \pi_k^k(g(y))))$$

g is computable, so $f(y) = \pi_1(g(y))$ is computable.

6.8. Unbounded minimalisation

Generalized composition and primitive recursion produce total functions when starting from total functions. Another essential operator, which instead allows to construct partial functions is the **unbounded minimalisation** operator.

It is similar to bounded minimalisation, but the search is not bounded and $f(\vec{x}, y)$ not necessarily total. It defines, informally, the following function:

$$\mu y.f(\vec{x}, y) = \text{ minimum } y \text{ s.t. } f(\vec{x}, y) = 0.$$

But there are two cases in which the definition has to be clarified:

  (1) if there is no y s.t. $f(\vec{x}, y) = 0$

  (2) if before finding a y s.t. $f(\vec{x}, y) = 0$, it happens that $f(\vec{x}, z) \uparrow$

In both cases the result of the minimalisation is undefined.

This is intuitive if we think about the obvious algorithm to compute the minimalisation: start from 0, $f(\vec{x}, 0) = 0$? if yes then $out(0)$, otherwise $f(\vec{x}, 1) = 0$? until $f(\vec{x}, y) = 0$.

DEFINITION 6.31. Let $f : \mathbb{N}^{k+1} \to \mathbb{N}$ be a function. Then the function $h : \mathbb{N}^k \to \mathbb{N}$ defined through **unbounded minimalisation** is:

$$h(\vec{x}) = \mu y.f(\vec{x}, y) = \begin{cases} \text{least } z \text{ s.t.} & \begin{cases} f(\vec{x}, z) = 0 \\ f(\vec{x}, z) \downarrow \quad f(\vec{x}, z') \neq 0 \quad \text{for } z < z' \end{cases} \\ \uparrow & \text{otherwise, if such a } z \text{ does not exist} \end{cases}$$

THEOREM 6.32 (Closure under minimalisation). *Let* $f : \mathbb{N}^{k+1} \to \mathbb{N}$ *a computable function (not necessarily total). Then* $h : \mathbb{N}^k \to \mathbb{N}$ *defined by* $h(\vec{x}) = \mu y.f(\vec{x}, y)$ *is computable.*

PROOF. Let F be a program in standard form for f.

Idea: for $z = 0, 1, 2, \ldots$ we compute $f(\vec{x}, z)$ until we find zero.

We need to save the argument $\vec{x}$ in a register R_m ($m = max\{\rho(F), k+1\}$) such that it is not used by the program F.

So the program for h is obtained as follows:

1	...	k	...	$m+1$	...	$m+k$	$m+k+1$
	$\vec{x}$				$\vec{x}$		z

$$
\begin{array}{lll}
& T(1, m+1) & \\
& \ldots & \\
& T(k, m+k) & \\
LOOP: & F[m+1, \ldots, m+k+1 \to 1] & // \; f(\vec{x}, z) \to R_1 \\
& J(1, m+k+2, END) & // f(\vec{x}, z) = 0? \\
& S(m+k+1) & // \; z = z+1 \\
& J(1, 1, LOOP) & \\
END: & T(m+k+1, 1) &
\end{array}
$$

$\square$

OBSERVATION 6.33. Observe that F may not terminate, this is correct! The entire program does not terminate and μ is undefined!

OBSERVATION 6.34. The unbounded minimalisation is nothing more than a **while** loop implemented with **goto**.

OBSERVATION 6.35. The μ operator allows us to obtain **non total** functions starting from total functions.

EXAMPLE 6.36. Given $f(x, y) = |x - y^2|$, we have that

$$
\mu y. f(x, y) = \begin{cases} \sqrt{x} & x \text{ is a perfect square} \\ \uparrow & \text{otherwise} \end{cases}
$$

EXERCISE 6.37. Let $f : \mathbb{N} \to \mathbb{N}$ be computable, total and injective. The the **inverse**

$$
f^{-1} = \begin{cases} y & f(y) = x \\ \uparrow & \nexists y. f(y) = x \end{cases}
$$

is computable. In fact, in our hypothesis $f^{-1}(x) = \mu y. |f(y) - x|$.

OBSERVATION 6.38. Intuitively, when f is not total, to find $f^{-1}(x)$ we consider a program P for f and execute it as follows:

- 0 steps of the program on argument 0

- 1 step on 0

- 0 steps on 1

- 2 steps on 0

    $\ldots$

in a dove-tail execution pattern.

Every time the program **terminates** in a certain number of steps k on argument y, we check the output $f(y)$, if $f(y) = x$ we stop, otherwise we continue.

EXERCISE 6.39. Prove that the following function is computable.
$$f(x, y) = \begin{cases} \frac{x}{y} & y \neq 0 \wedge y \mid z \\ \uparrow & \text{otherwise} \end{cases}$$

PROOF.
$$f(x, y) = \mu z.(|yz - x| + \mathcal{X}_{x=0 \wedge y=0}(x, y))$$
$\square$

LEMMA 6.40. *All* finite functions *(functions with finite domain) are computable.*

PROOF. Let $\theta : \mathbb{N} \to \mathbb{N}$ a finite domain function
$$\theta = \{(x_1, y_1), \ldots, (x_n, y_n)\}$$
i.e.

$$\theta(x) = \begin{cases} y_1 & x = x_1 \\ \ldots \\ y_n & x = x_n \\ \uparrow & \text{otherwise} \end{cases}$$

then
$$\theta(x) = \sum_{i=1}^{n} y_i \cdot \bar{sg}(|x - x_i|) + \mu z.(\prod_{i=1}^{n} |x - x_i|)$$
The minimalisation is needed only to make the function $\uparrow$ when $x \neq x_1, \ldots, x_n$, it is 0 otherwise. $\square$

CHAPTER 7

Other approaches to computability

We already observed that the URM machine is just one of the many possible computational models that allow us to formalize the notion of computable functions.

We could have used:

- Turing machine

- Canonical deduction systems of Post

- λ-calculus of Church

- Partial recursive functions of Gödel-Kleene

All of these approaches define the **same class of computable functions**, leading to the

Church-Turing thesis: a function is computable through an effective procedure if and only if it is URM-computable

Now, we introduce another formalism for the definition of computable functions, the set $\mathcal{R}$ of **partial recursive functions** of Gödel-Kleene and prove that it is equivalent to the URM, meaning it defines the same class of functions: $\mathcal{R} = \mathcal{C}$.

7.1. Partially recursive functions

DEFINITION 7.1 (Partially recursive functions). The class $\mathcal{R}$ of **partially recursive functions** is the least class of partial functions on the natural numbers which contains

(a) zero function;

(b) successor;

(c) projections

and **closed** under

(1) composition;

(2) primitive recursion;

(3) minimalisation.

We argue that the above is a well given definition.

DEFINITION 7.2 (Rich class). A class of functions $\mathcal{A}$ is said to be **rich** if it includes (a),(b) and (c) and it is closed under (1), (2) and (3).

$\mathcal{R}$ is rich and for all $\mathcal{A}$, we have $\mathcal{R} \subseteq \mathcal{A}$

REMARK 7.3. The property of being a rich class is **closed under intersection**: let $\{\mathcal{A}_i\}_{i \in I}$ a family of rich classes, then $\bigcap_{i \in I} \mathcal{A}_i$ rich.

Finally we observe that

PROPOSITION 7.4. *The set of the partially recursive functions can be characterised as*

$$\mathcal{R} = \bigcap_{\mathcal{A} \ rich} \mathcal{A}$$

We can now prove the main result, showing that the class of URM-computable functions coincides with the class of partial recursive functions.

THEOREM 7.5. $\mathcal{R} = \mathcal{C}$

PROOF.
$(\mathcal{R} \subseteq \mathcal{C})$
Just observe that $\mathcal{C}$ is a rich class, $\mathcal{R}$ is the smallest rich class, so this inclusion trivially follows.

$(\mathcal{C} \subseteq \mathcal{R})$
Let $f : \mathbb{N}^k \to \mathbb{N} \in \mathcal{C}$ be a computable function. We have to show that $f \in \mathcal{R}$.

We know that there exists a URM program P such that $f_P^{(k)} = f$.

Consider the following functions

- $c_P^1 : \mathbb{N}^{k+1} \to \mathbb{N}$ with $c_P^1(\vec{x}, t)$ be the content of R_1 after t steps of $P(\vec{x})$. If $P(\vec{x})$ terminates in less than t steps, $c_P^1(\vec{x}, t)$ gives the content of R_1 in the final configuration, i.e. the output of the function f;

- $j_P : \mathbb{N}^{k+1} \to \mathbb{N}$ with $j_P(\vec{x}, t)$ be the instruction to be executed after t steps of $P(\vec{x})$. If the program has already ended, then $j_P(\vec{x}, t) = 0$.

Clearly c_p^1 and j_p are total functions.

Given $\vec{x} \in \mathbb{N}^k$

- if $f(\vec{x}) \downarrow$ then $P(\vec{x}) \downarrow$ in a number of steps $t_0 = \mu t. j_P(\vec{x}, t)$, so
$$f(\vec{x}) = c_P^1(\vec{x}, t_0) = c_P^1(\vec{x}, \mu t. j_P(\vec{x}, t))$$

- otherwise, if $f(\vec{x}) \uparrow$ then $P(\vec{x}) \uparrow$ and $\mu t. j_P(\vec{x}, t) \uparrow$, and thus
$$f(\vec{x}) = c_P^1(\vec{x}, \mu t. j_P(\vec{x}, t)) \uparrow$$

therefore
$$f(\vec{x}) = c_P^1(\vec{x}, \mu t. j_P(\vec{x}, t)) \quad \forall \vec{x} \in \mathbb{N}^k$$

If we knew that $c_P^1, j_P \in \mathcal{R}$ then we could argue that $f \in \mathcal{R}$.

The idea of the proof is the following

- work on sequences encodings that represent the registers and program counter configurations

- manipulate such sequences with functions such as $(p_x, qt, \mathrm{div}, \dots)$ that were built by:

  - composition

  - primitive recursion

in this way, we obtain c_P^1, j_P through primitive recursion.

A register configuration in which a finite number of registers contains a value other than 0 can be encoded in the following way:

$$c = \prod_{i \geqslant 1} p_i^{r_i} = \prod_{i=1}^{k} p_i^{r_i}$$

such that

$$r_i = (x)_i$$

Thus, using this encoding, we can consider a function $c_P : \mathbb{N}^{k+1} \to \mathbb{N}$, that provides the (encoding of the) registers' configuration after t steps of $P(\vec{x})$.

We define c_P, j_P by primitive recursion:

- base cases

$$c_P(\vec{x}, 0) = \prod_{i=1}^{k} p_i^{x_i}$$
$$j_P(\vec{x}, 0) = 1$$

- recursive cases
  In order to simplify the notation, below let $c = c_P(\vec{x}, t)$ and $j = j_P(\vec{x}, t)$.

$$c_P(\vec{x}, t+1) = \begin{cases} qt(p_n^{(c)_n}, c) & \text{if } 1 \leqslant j \leqslant l(P) \ \& \ I_j = Z(n) \\ p_n \cdot c & \text{if } 1 \leqslant j \leqslant l(P) \ \& \ I_j = S(n) \\ qt(p_n^{(c)_n}, c) \cdot p_n^{(c)_m} & \text{if } 1 \leqslant j \leqslant l(P) \ \& \ I_j = T(m, n) \\ c & \text{otherwise} \end{cases}$$

$$j_P(\vec{x}, t+1) = \begin{cases} j+1 & \text{if } 1 \leqslant j < l(P) \ \& \ I_j = Z(n), S(n), T(m,n) \\ & \quad \text{or } J(m,n,t) \text{ with } (c)_m \neq (c)_n \\ u & \text{if } 1 \leqslant j \leqslant l(P) \ \& \ I_j = J(m,n,u) \\ & \quad \& \ (c)_m = (c)_n \ \& \ \text{if } 1 \leqslant u \leqslant l(P) \\ 0 & \text{otherwise} \end{cases}$$

thus c_P, j_P are in $\mathcal{R}$. Hence also c_P^1 is, since $c_P^1(\vec{x}, t) = (c_P(\vec{x}, t))_1$ for all $\vec{x} \in \mathbb{N}^k$ and $t \in \mathbb{N}$. Therefore f, defined by composition and minimalisation of c_P^1 and j_P is in $\mathcal{R}$, as desired. $\qquad\square$

Primitive recursive functions

We define the primitive recursive functions as follows

DEFINITION 8.1 (Primitive recursive functions). The class of *primitive recursive functions* is the smallest class of functions $\mathcal{PR}$ containing

- (a) zero function
- (b) successor
- (c) projections

and closed under

- (1) composition
- (2) primitive recursion

One reason of interest for $\mathcal{PR}$ is that primitive recursion intuitively corresponds to bounded iteration, i.e., `for` loops constructs, while minimalisation corresponds to unbounded iteration, i.e., `while` loops. This fact can be formalized by considerng variant on the URM machine, with *structured programs*, where the jump instruction is replaced by `for` and `while` loops. We'll call this machine URM$_{\text{for,while}}$.

We can prove that this model has the same expressive power as the URM model, i.e., the class $\mathcal{C}_{\text{for,while}}$ coincides with $\mathcal{C} = \mathcal{R}$. Instead the class $\mathcal{C}_{\text{for}}$ of functions computable using only the `for` construct coincides with $\mathcal{PR}$.

Thus, studying the relation between $\mathcal{R}$ and $\mathcal{PR}$ corresponds to studying the relation between the expressive power of `for` and `while` constructs. We know that many "arithmetic" functions, like $Pr(x), (x)_y, qt, mcm(x,y), x^y$ are in $\mathcal{PR}$ and $\mathcal{PR}$ is closed under sum, product and minimalisation. This class is very ample, but it does not contain all computable functions, in other words $\mathcal{PR} \subsetneq \mathcal{R}$, because $\mathcal{PR}$ functions are always total, since $\mathcal{PR}$ functions are obtainable from base total functions by composition and primitive recursion.

One could still suppose that $\mathcal{PR}$ includes all the total recursive functions, in other words if Tot is the set of all total functions: $\mathcal{PR} = \mathcal{R} \cap Tot$ [Hilbert, 1926].

This is false, i.e.,

$$\mathcal{PR} \subsetneq \mathcal{R} \cap \text{Tot}$$

i.e., even if we restrain ourselves to total functions (programs that always terminates), the `while` construct is essential.

8.1. Ackermann's function

A function which witnesses the strict inclusion $\mathcal{PR} \subsetneq \mathcal{R} \cap \text{Tot}$ is the Ackermann function which can be proved to be total computable and not primitive recursive.

The Ackermann's function is $\psi : \mathbb{N}^2 \to \mathbb{N}$ defined as

$$\begin{cases} \psi(0, y) = y + 1 \\ \psi(x + 1, 0) = \psi(x, 1) \\ \psi(x + 1, y + 1) = \psi(x, \psi(x + 1, y)) \end{cases}$$

This scheme uniquely determine a function, because the value $\psi(x, y)$ is always defined based on *smaller* values of ψ itself. But what does *smaller* mean?

- in $\psi(x + 1, 0) = \psi(x, 1)$ the first argument diminishes.

- in $\psi(x + 1, y + 1) = \psi(x, \psi(x + 1, y))$ at first we compute $\psi(x + 1, y)$ where the second argument diminishes and then $\psi(x, u)$ in which the first argument u is, the first argument is smaller.

We can see that the arguments diminish in a *lexicographical* order on $\mathbb{N}^2$, i.e., in $(\mathbb{N}^2, \leqslant_{lex})$ with $(x, y) \leqslant (x', y')$ if $(x < x') \wedge (x = x' \text{ and } y \leqslant y')$ and we can show that $(\mathbb{N}^2, \leqslant_{lex})$ does not allow for infinite descending sequences

DEFINITION 8.2 (Partially ordered set). A set D with a binary relation $\leqslant$ is a partially ordered set (poset) $(D, \leqslant)$ if $\leqslant$ is a partial order, i.e., for all $x, y, z \in D$, it is

(1) reflexive: $x \leqslant x$;

(2) antisymmetric: if $x \leqslant y$ and $y \leqslant x$, then $x = y$;

(3) transitive: if $x \leqslant y$ and $y \leqslant z$, then $x \leqslant z$.

DEFINITION 8.3 (Well-founded poset). $(D, \leqslant)$ is well-founded if every non-empty $X \subseteq D$ has a minimal element d, i.e.

$$\forall d' \in X \quad d' \leqslant d \Rightarrow d' = d$$

OBSERVATION 8.4. $(D, \leqslant)$ is well-founded iff it does not allow for infinite descending chains

$$d_0 > d_1 > d_2 > \cdots > d_n > d_n + 1 \ldots$$

This fact can be useful when dealing with termination problems. If we can conclude that the set of configurations is well-founded, we simply need to prove that for each step $\text{conf}_i \to \text{conf}_{i+1}$ and $\text{conf}_{i+1} < \text{conf}_i$ to end our proof. This way our computation descends a decreasing sequence of values, which is necessarily finite.

Looking back at the Ackermann function, the computation of ψ is based on the computation of ψ with smaller values, at some point it will for sure reach the case $\psi(0, y) = y + 1$, terminating.

EXAMPLE 8.5. $(\mathbb{N}^2, \leqslant_{lex})$ is *well-founded*. Let $\varnothing \neq X \subseteq \mathbb{N}^2$ and define

$$x_0 = \min\{x \mid \exists y \in \mathbb{N}.(x, y) \in X\}$$
$$y_0 = \min\{y \mid (x_0, y) \in X\}$$

then we can see that $\min X = (x_0, y_0)$. (Actually, in this way we can prove that the product of two well-ordered sets is well-ordered.)

OBSERVATION 8.6. $\leqslant_{lex}$ is total.

Over the natural numbers we can prove the so-called *complete induction principle*: if $\forall n' < n$. $P(n')$ implies $P(n)$ then we can deduce that $\forall n \in \mathbb{N}$. $P(n)$. The principle can be actually generalised to each well-founded poset D:

DEFINITION 8.7 (Well-founded induction). Let $(D, \leqslant)$ be a well-founded poset and let $P(x)$ a property on elements of D. If for all $d \in D$, assuming $P(d')$ for $d' < d$, we can conclude that $P(d)$ holds, then

$$\forall d \in D.P(d)$$

THEOREM 8.8. *The Ackermann's function ψ is total, i.e.*

$$\forall (x, y) \in \mathbb{N}^2 \quad \psi(x, y) \downarrow$$

PROOF. We proceed by well-founded induction on $(\mathbb{N}^2, \leqslant_{lex})$. Let $(x, y) \in \mathbb{N}^2$, assume

$$\forall (x', y') \leqslant_{lex} (x, y) . \psi(x', y') \downarrow$$

we want to prove $\psi(x', y') \downarrow$. We have 3 cases:

- $(x = 0)$
  $\psi(0, y) = y + 1 \downarrow$

- $(x > 0, y = 0)$
  $\psi(x, 0) = \psi(x - 1, 1) \downarrow$ for inductive hypothesis, since $(x - 1, 1) \leqslant_{lex} (x, 0)$

- $(x > 0, y > 0)$
  $\psi(x, y) = \psi(x - 1, \psi(x, y - 1))$ where $\psi(x, y - 1) \downarrow$ by inductive hypothesis. Let $u = \psi(x, y - 1)$, so $\psi(x, y) = \psi(x - 1, u) \downarrow$ by inductive hypothesis.

$\square$

EXERCISE 8.9. Given a box with an arbitrary number of balls in it, each one with a number in $\mathbb{N}$, do the following:

- extract a ball;

- substitute the extracted ball with an arbitrary number of balls, each one with a label lower than the extracted one.

Prove that this process always terminates.

THEOREM 8.10. *The Ackermann's function ψ is computable, i.e.*

$$\psi \in \mathcal{C} = \mathcal{R}$$

One could argue by using the Church-Turing thesis: the computation of $\psi(x, y)$ is always reduced to the computation of ψ on smaller input values until we reach a base case where the successor is used.

The above is unsatisfactory. A formal proof can be based on the notion of a valid set. Intuitively a set $S \subseteq \mathbb{N}^3$ is considered valid if, for all $(x, y, z) \in S$, we have

- $z = \psi(x, y)$

- S contains all the triples needed to compute $\psi(x, y)$

EXAMPLE 8.11. $\psi(1, 1) = \psi(0, \psi(1, 0)) = \psi(0, \psi(0, 1)) = \psi(0, 2) = 3$

$\Rightarrow S = (1, 1, 3), (0, 2, 3), (1, 0, 2), (0, 1, 2)$

Formally:

DEFINITION 8.12 (Valid set). Let S be a set of triples such that $S \subseteq \mathbb{N}^3$. We say that S is *valid* if:

(1) $(0, y, z) \in S \quad \Rightarrow \quad z = y + 1$

(2) $(x + 1, 0, z) \in S \quad \Rightarrow \quad (x, 1, z) \in S$

(3) $(x + 1, y + 1, z) \in S \quad \Rightarrow \quad \exists u \, . \, (x + 1, y, u) \in S \wedge (x, u, z) \in S$

We can prove that for every $(x, y, z) \in \mathbb{N}^3$ we have $\psi(x, y) = z$ if and only if there exists a valid **finite** set of triples $S \subseteq \mathbb{N}^3$ such that $(x, y, z) \in S$ by complete induction on (x, y), knowing that the validity of a set is preserved under union (left as an exercise).

A triple (x, y, z) can be encoded into an integer using the encoding function

$$\pi^3 : \mathbb{N}^3 \to \mathbb{N} \qquad (\pi_i^3 : \mathbb{N} \to \mathbb{N} \text{ are the projections})$$

In this way a set of triples becomes a set of natural numbers $\{x_1, \ldots, x_n\}$ that we can encode injectively as the product

$$\{x_1, \ldots, x_n\} \mapsto p_{x_1} \cdot \cdots \cdot p_{x_n}$$

Now given $\nu \in \mathbb{N}$ which represents a set of triples S_ν we have that

$$(x, y, z) \in S_\nu \iff div(p_{\pi(x, y, z)}, \nu)$$

and the predicate $\text{Val}(\nu) =$ "ν encodes a set of valid tuples" is decidable.

In fact $\text{Val}(\nu)$ is true if and only if:

- $\forall i \leqslant \nu \quad (\nu)_i \leqslant 1$

- $\forall \omega \leqslant \nu \; div(p_\omega, \nu)$

$$\Rightarrow \begin{cases} \pi_1(\omega) = 0 \quad \Rightarrow \quad \pi_3(\omega) = \pi_2(\omega) + 1 \\ \\ \pi_1(\omega) > 0 \quad \Rightarrow \quad \begin{cases} \pi_2(\omega) = 0 \quad \Rightarrow \pi(\pi_1(\omega), 0, \pi_3(\omega)) \in S_\nu \\ \pi_2(\omega) > 0 \quad \Rightarrow \exists u \leqslant \omega \text{ s.t.} \\ \qquad \pi(\pi_1(\omega), \pi_2(\omega) - 1, u) \in S_\omega \\ \qquad \pi(\pi_1(\omega) - 1, u, z) \in S_\omega \end{cases} \end{cases}$$

with associated characteristic function

$$\chi_{\mathrm{Val}} \in \mathcal{PR}$$

We can also verify that

$$R(x, y, z) = \begin{cases} \chi_{\mathrm{Val}(\omega)} & \text{if } \omega \text{ encodes some valid } S \text{ that contains } (x, y, z) \text{ for some } z \\ 0 & \text{otherwise} \end{cases}$$

$$= \chi_{\mathrm{Val}(\omega)} \cdot \mathrm{sg}(\omega + 1 \dot{-} \mu z \leqslant \omega.\mathrm{div}(P_{\pi(x,y,z)}, \omega)$$

Thus we can write the Ackermann function as

$$\psi(x, y) = \mu(z, y) \cdot \overline{\mathrm{sg}}(R(x, y, \omega) \cdot \mathrm{div}(p_{\pi(x,y,z)}, \omega))$$

Since it is computable,

$$\psi \in \mathcal{R} = \mathcal{C}$$

THEOREM 8.13. *The Ackermann's function is not primitive recursive:*

$$\psi \notin \mathcal{PR}$$

INFORMAL IDEA OF THE PROOF. The proof of the fact that ψ is not a primitive recursive function is done by showing that ψ *grows* faster than every function in $\mathcal{PR}$. We already saw how we obtain

- sum from successor
- product from sum
- exponential from product

each one by nested primitive recursion.

The idea of the Ackermann function is that it won't be possible to compute it with a finite number of nested primitive recursions.

In fact, by calling

$$\psi_x(y) = \psi(x, y)$$

we have that

$$\psi_{x+1}(y) = \psi_x(\psi_{x+1}(y - 1)) = \psi_x^2(\psi_{x+1}(y - 2)) = \cdots = \psi_x^{y+1}(1)$$

$$\psi_0(y) = y + 1 = succ(x)$$
$$\psi_1(y) = \psi_0^{y+1}(1) = y + 2$$
$$\psi_2(y) = \psi_1^{y+1}(1) = 2y + 3$$
$$\psi_3(y) = \psi_2^{y+1}(1) = 2^{y+3} - 3$$

e.g.

$$\psi_0(1) = 2, \quad \psi_1(1) = 3, \quad \psi_2(1) = 5, \quad \psi_3(1) = 13, \quad \ldots$$

Intuitively, if x grows so does the level of nesting in the functions, which is equivalent to say that we need more nested `for` loops. Since x can grow to infinity and `for` loops cannot be nested to infinity, a `while` loop is needed. More precisely, given

a function $f : \mathbb{N}^n \to \mathbb{N} \in \mathcal{PR}$ and a program P computing f using only *for-loops* (primitive recursion), if j is the maximum level of nesting of `for`-loops, then

$$f(\vec{x}) < \psi_{j+1}(\max\{x_1, \ldots, x_k\})$$

Now, assume $\psi \in \mathcal{PR}$, let j be the level of nesting of `for`-loops for computing ψ, so

$$\forall (x, y) \; . \; \psi(x, y) < \psi_{j+1}(\max\{x, y\})$$

Let $x = y = j + 1$ big enough we have that

$$\psi(j + 1, j + 1) < \psi_{j+1}(j + 1) = \psi(j + 1, j + 1)$$

which is absurd, so $\psi \notin \mathcal{PR}$. $\hfill\square$

OBSERVATION 8.14. Initially, Gödel and Kleene studied a class of functions $\mathcal{R}_0$ called μ-recursive. This class contained

    a zero function

    b successor

    c projections

and was closed under

    (1) composition;

    (2) primitive recursion;

    (3) minimization, restricted to the case in which the function that produces is *total*.

$\mathcal{R}_0 \subset \mathcal{R}$ trivially holds, since:

- functions in $\mathcal{R}_0$ are total;

- some functions in $\mathcal{R}$ are partial.

Also

$$\mathcal{R}_0 \subseteq \mathcal{R} \cap \text{Tot}$$

but is not obvious that the equality holds. In fact, a function $f \in \mathcal{R} \cap \text{Tot}$ can be total, but obtained through minimization of partial functions. For example:

$$f(x, y) = \begin{cases} x + 1 & x < y \\ 0 & x = y \\ \uparrow & x > y \end{cases}$$

$$\mu y.f(x, y) = \lambda x.x$$

thus, $f(x, y)$ is partial and $\mu y.f(x, y)$ is total, then

$$\mu y.f(x, y) \in \mathcal{R}_0$$

THEOREM 8.15.

$$\mathcal{R}_0 = \mathcal{R} \cap \text{Tot}$$

PROOF.   ($\subseteq$) trivial.

($\supseteq$) Let $f \in \mathcal{R} \cap \text{Tot}$, then $f \in \mathcal{C}$. We can observe that

$$f(\vec{x}) = c_P^1(\vec{x}, \mu t.j_P(\vec{x}, t))$$

but c_P^1, j_P are total, so f is total.

$\square$

Enumeration of programs

The objective here is to define an *effective enumeration* of URM programs and URM-computable functions. These results will be fundamental for our theory, and in particular to

- prove the existence of non computable functions

- the *smn* theorem

- the universal function/machine.

DEFINITION 9.1 (Countable set). A is **countable** if $|A| \leqslant |\mathbb{N}|$, i.e. we have $f : \mathbb{N} \to A$ surjective. We say that f is an enumeration of X, because we can enumerate all elements in X as

$$f(0), f(1), f(2), \ldots$$

An enumeration is **without repetitions** if it is injective (and thus bijective).

We will call **effective** those enumerations which are "intuitively"computable, but the type does not allow to talk formally about their computability. Note that we will argue about effectiveness by showing that they are built using components which are formally computable and we will only use in proofs the computability of these components.

LEMMA 9.2. *There are effective bijective enumerations of*

 (1) $\mathbb{N}^2$

 (2) $\mathbb{N}^3$

 (3) $\bigcup_{k \geqslant 1} \mathbb{N}^k$

PROOF.   (1) we already saw that

$$\pi : \mathbb{N}^2 \to \mathbb{N}$$
$$\pi(x, y) = 2^x(2y + 1) - 1$$

is computable with inverse

$$\pi^{-1} : \mathbb{N} \to \mathbb{N}^2$$
$$\pi^{-1}(x) = (\pi_1(x), \pi_2(x))$$

where $\pi_1, \pi_2 : \mathbb{N} \to \mathbb{N}$

$$\pi_1(n) = (n+1)_1$$

$$\pi_2(n) = \left(\left(\frac{n+1}{2^{\pi_1(n)}}\right) - 1\right)$$

are computable.

(2) consider

$$\nu : \mathbb{N}^3 \to \mathbb{N}$$

$$\nu(x, y, z) = \pi(\pi(x, y), z)$$

with inverse built upon projections

$$\nu^{-1} : \mathbb{N} \to \mathbb{N}^3$$

$$\nu^{-1}(x) = (\nu_1(x), \nu_2(x), \nu_3(x))$$

with ν_1, ν_2, ν_3 are computable.

(3) The following tuple encoding

$$\tau : \bigcup_{k \geqslant 1} \mathbb{N}^k \to \mathbb{N}$$

$$\tau(x_1, \ldots, x_k) = \prod_{i=1}^{k} p_i^{x_i} - 1$$

does not work, since it is not injective. The idea is that we can *increment* the last component, in this way

$$\tau(x_1, \ldots, x_k) = \left(\prod_{i=1}^{k-1} p_i^{x_i}\right) \cdot p_k^{x_k+1} - 2$$

with inverse $\tau^{-1} : \mathbb{N} \to \bigcup_{k \geqslant 1} \mathbb{N}^k$ defined out of the following functions:

- $l : \mathbb{N} \to \mathbb{N}$

$$l(n) = \max\{k : div(p_k, (x+2)) = 1\} = x - \mu(h \leqslant x) \cdot \overline{sg}(div(p_{x-h}, (x+2)))$$

- $a : \mathbb{N}^2 \to \mathbb{N}$

$$a(n, i) = \begin{cases} (n+2)_i & i = 1, \ldots, \ell(x) - 1 \\ (n+2)_i - 1 & i = \ell(x) \end{cases}$$

An alternative encoding is the following

- $\tau(x_1, \ldots, x_k) = \pi(\prod_{i=1}^{k} p_i^{a_i}, k)$
- $l(n) = \pi_2(n)$
- $a(n, i) = (\pi_1(n))_i$

$\square$

THEOREM 9.3. *Let $\mathcal{P}$ the set of all URM programs. Then there exists an effective bijective enumeration of $\mathcal{P}$.*

$$\gamma : \mathcal{P} \to \mathbb{N}$$

PROOF. Let $\mathcal{F}$ the set of all URM instructions. First, we'll prove that there exists
$$\beta : \mathcal{F} \to \mathbb{N}$$
a bijective effective correspondence. The idea is to use the enumeration of pairs and triples, sending

- $Z(n)$ instructions to multiples of 4

- $S(n)$ instructions to numbers congruent 1 $\mod 4$

- $T(m,n)$ instructions to numbers congruent 2 $\mod 4$

- $J(m,n,t)$ instructions to numbers congruent 3 $\mod 4$

Concretely
$$\begin{cases} \beta(Z(n)) = 4 * (n-1) \\ \beta(S(n)) = 4 * (n-1) + 1 \\ \beta(T(m,n)) = 4 * \pi(m-1, n-1) + 2 \\ \beta(J(m,n,t)) = 4 * \nu(m-1, n-1, t-1) + 3 \end{cases}$$
with inverse $\beta^{-1} : \mathbb{N} \to \mathcal{F}$ such that, let $r = rm(4, x)$ and $q = qt(4, x)$,

$$\beta^{-1}(x) = \begin{cases} Z(q+1) & \text{if } r = 0 \\ S(q+1) & \text{if } r = 1 \\ T(\pi_1(q)+1, \pi_2(q)+1) & \text{if } r = 2 \\ J(\nu_1(q)+1, \nu_2(q)+1, \nu_3(q)+1) & \text{if } r = 3 \end{cases}$$

so both β and β^{-1} are effective. Now $\gamma : \mathcal{P} \to \mathbb{N}$ can be defined as follows: if $P = I_1 \ldots I_s$, then
$$\gamma(P) = \tau(\beta(I_1), \ldots, \beta(I_s))$$
with inverse $\gamma^{-1}(x) = P = I_1 \ldots I_{l(x)}$, where $I_i = \beta^{-1}(a(n, i))$. Thus, γ is bijective because is composition of bijective functions. Since γ, γ^{-1} are effective, $\mathcal{P}$ is effectively denumerable. $\square$

DEFINITION 9.4 (Gödel number). Given $P \in \mathcal{P}$ the value $\gamma(P)$ is called code (or Gödel number) of P. Usually we'll write P_n to represent $\gamma^{-1}(n)$, the nth program of the enumeration.

OBSERVATION 9.5. From now on we will consider a fixed enumeration γ of programs, which determines the meaning of P_n. This fixed enumeration can be defined in various ways, but we need to fix one, in a way that:

- given a program P we can compute in an effective way its code $\gamma(P)$;

- given a number it is possible to find the nth program $P_n = \gamma^{-1}(n)$.

EXAMPLE 9.6. Let us consider the program P

$T(1, 2)$

$S(2)$

$T(2, 1)$

encoded by

$$\beta(T(1,2)) = 4 * \pi(1-1, 2-1) + 2 = 4 * \pi(0,1) + 2 = 10$$
$$\beta(S(2)) = 4 * (2-1) + 1 = 5$$
$$\beta(T(2,1)) = 4 * \pi(2-1, 1-1) + 2 = 4 * \pi(1,0) + 2 = 6$$

then

$$\gamma(P) = \tau(10, 5, 6)$$
$$= p_1^{10} \cdot p_2^5 \cdot p_3^{6+1} - 2$$
$$= 2^{10} \cdot 3^5 \cdot 5^7 - 2$$
$$= 19439999998$$

What does this program compute? $\lambda x.x + 1$.

The program $P' : S(1)$ computes the same function. In this case the encoding is

$$\beta(S(1)) = 4 * (1-1) + 1 = 1$$

and so

$$\gamma(P') = \tau(1) = 2^{1+1} - 2 = 2$$

EXAMPLE 9.7. Show what $P_{100} = \gamma^{-1}(100)$ is.

We observe that

$$100 + 2 = 2^1 * 3^1 * 17^1 = p_1^1 \cdot p_2^1 \cdot p_3^1 \cdot p_4^0 \cdot p_5^0 \cdot p_6^0 \cdot p_7^1$$

hence the program contains 7 instructions:

$$\beta^{-1}(1) \to \quad S(1)$$
$$\beta^{-1}(1) \to \quad S(1)$$
$$\beta^{-1}(0) \to \quad Z(1)$$
$$\beta^{-1}(0) \to \quad Z(1)$$
$$\beta^{-1}(0) \to \quad Z(1)$$
$$\beta^{-1}(0) \to \quad Z(1)$$
$$\beta^{-1}(0) \to \quad Z(1)$$

Clearly, an enumeration of URM programs induces an enumeration of computable functions

DEFINITION 9.8. For a fixed an effective enumeration $\gamma : \mathcal{P} \to \mathbb{N}$ we define:

1. $\varphi_n^{(k)}$: the function of k arguments (k-ary function) computed by the program $P_n = \gamma^{-1}(n)$ (with the notation presently introduced: $\varphi_n^{(k)} = f_{P_n}^{(k)}$)

2. $W_n^{(k)} = \mathrm{dom}(\varphi_n^{(k)}) \subseteq \mathbb{N}^k$

3. $E_n^{(k)} = \mathrm{cod}(\varphi_n^{(k)}) \subseteq \mathbb{N}$

usually if $k = 1$, it is omitted. $\varphi_n = \varphi_n^{(1)}$

OBSERVATION 9.9. The function
$$\varphi^{(k)} : \quad \mathbb{N} \to \mathcal{C}^{(k)}$$
$$n \mapsto \varphi_n^{(k)}$$

is obviously surjective (each computable function is computed by a program!), and so $\mathcal{C}^{(k)}$ is countable:
$$|\mathcal{C}^{(k)}| = |\mathbb{N}|$$

Actually, from the existence of a surjective function $\mathbb{N} \to \mathcal{C}$ it follows that $|\mathcal{C}^{(k)}| \leqslant |\mathbb{N}|$. Equality $|\mathcal{C}^{(k)}| \geqslant |\mathbb{N}|$ derives from the observation that there are infinitely many computable functions, for example constants $\lambda x_1 \ldots x_k.c$.

Clearly $\varphi^{(k)} : \mathbb{N} \to \mathcal{C}^{(k)}$ is not injective. In fact, for each computable function there are infinitely many programs that compute it
$$\forall f \in \mathcal{C} \quad |(\varphi^{(k)})^{-1}(f)| = |\mathbb{N}|$$
which means $\varphi_0^{(k)}, \varphi_1^{(k)}, \varphi_2^{(k)}, \ldots$ is an enumeration of $\mathcal{C}$ with infinitely many repetitions. An enumeration without repetitions can be defined as:
$$\chi(0) = 0$$
$$\chi(n+1) = \mu z \,.\, (\varphi_z \notin \{\varphi_{\chi(0)}, \ldots, \varphi_{\chi(n)}\})$$
which raises the enumeration $\varphi_{\chi(0)}, \varphi_{\chi(1)}, \varphi_{\chi(2)}, \ldots$ but this enumeration is highly ineffective.

It can be proved that there exists $h : \mathbb{N} \to \mathbb{N}$ total and computable such that $\varphi_{h(0)}, \varphi_{h(1)}, \varphi_{h(2)}, \ldots$ is an enumeration without repetitions [Fri58]. However, enumerations with repetitions are sufficient for us.

THEOREM 9.10 ($|\mathcal{C}| = |\mathbb{N}|$). *The class $\mathcal{C}$ of computable functions is countable.*

PROOF.
$$\mathcal{C} = \bigcup_{k \geqslant 1} \mathcal{C}^{(k)}$$
Since the union of countable sets is countable, $\mathcal{C}$ is countable. □

OBSERVATION 9.11. From now on we will implicitly use the enumeration of programs γ. The meaning of $\varphi_n^{(k)}, W_n^{(k)}, E_n^{(k)}$ is fixed and determined starting from γ.

CHAPTER 10

Cantor diagonalization technique

Roughly speaking, the diagonalization technique allows one to build an object that differs from a (countable) infinity of similar objects. The idea behind is: given an countable set of objects $\{x_1, x_2, x_3, \dots\}$ we can build another object x of the same nature of the x_n's, but different from all of them by making it "differ from x_n on n".

This is the original method used by Cantor, one of the founding fathers of set theory, to prove that there are various "degrees of infinity" (observing that the powerset 2^X of a set A always has cardinality strictly larger than the cardinality of X).

We provide a proof in the specific case of the natural numbers.

PROPOSITION 10.1. $|\mathbb{N}| < |2^{\mathbb{N}}|$

PROOF. By contradiction $|\mathbb{N}| \geqslant |2^{\mathbb{N}}|$, i.e. $|2^{\mathbb{N}}|$ countable. This means that there exists an enumeration of $2^{\mathbb{N}}$: $x_0, x_1, x_2, \dots$

Consider

	X_0	X_1	X_2	$\dots$
0	?	NO	$\dots$	
1	NO	?	YES	
2	YES	NO	?	
$\vdots$				

We can define $D = \{i \mid i \notin X_i\} \subseteq \mathbb{N}$ which "differs from X_i on i" element. Obviously $D \in 2^{\mathbb{N}}$ which means that there exists k such that $D = X_k$. But is k in D?

$$k \in D \quad \Rightarrow \quad k \notin X_k = D$$
$$k \notin D \quad \Rightarrow \quad k \in X_k = D$$

which is absurd. Therefore $|\mathbb{N}| < |2^{\mathbb{N}}|$. $\qquad\square$

EXAMPLE 10.2. Consider $\mathbb{N} \to \mathbb{N} = \{f \mid f : \mathbb{N} \to \mathbb{N}\}$, we have

$$|\mathbb{N} \to \mathbb{N}| > |\mathbb{N}|$$

PROOF. There are two approaches to proceed

(1) Define

$$\mathbb{N} \to 2 = \{f \mid f : \mathbb{N} \to \mathbb{N} \text{ total }, \forall x \ f(x) \in \{0,1\}\} \subseteq \mathbb{N} \to \mathbb{N}$$

Note that there is a bijection between $\mathbb{N} \to 2$ and $2^{\mathbb{N}}$ and thus

$$|\mathbb{N} \to \mathbb{N}| \geqslant |\mathbb{N} \to 2| > |\mathbb{N}|$$

(2) Let $f_1, f_2, f_3, \ldots$ be an enumeration of elements in $\mathbb{N} \to \mathbb{N}$ and consider

	f_0	f_1	f_2	$\ldots$
0	$f_0(0)$	$\ldots$	$\ldots$	
1	$\ldots$	$f_1(1)$	$\ldots$	
2	$\ldots$	$\ldots$	$f_2(2)$	
$\vdots$				

We can define a function f that differs from every other function by considering the diagonal and systematically changimg it:

$$f(i) = \begin{cases} 0 & \text{if } f_i(i) \uparrow \\ \uparrow & \text{if } f_i(i) \downarrow \end{cases}$$

In this way

$$\forall i \quad f \neq f_i \quad \text{since } f(i) \neq f_i(i)$$

Hence no enumeration of functions in $\mathbb{N} \to \mathbb{N}$ can include the whole $\mathbb{N} \to \mathbb{N}$, which is this not countable.

$\square$

COROLLARY 10.3. *The set $\bar{\mathcal{C}} = \{f : \mathbb{N} \to \mathbb{N} \mid f \text{ not computable}\}$ is not countable.*

PROOF. We know that $|\mathcal{C}| = |\mathbb{N}|$. If $\bar{\mathcal{C}}$ were countable, then $\mathbb{N} \to \mathbb{N} = \mathcal{C} \cup \bar{\mathcal{C}}$ would be countable, which is absurd for the previous corollary. $\square$

OBSERVATION 10.4. There exists a total non-computable function $f : \mathbb{N} \to \mathbb{N}$ defined by

$$f(n) = \begin{cases} \varphi_n(n) + 1 & \text{if } \varphi_n(n) \downarrow \\ 0 & \text{if } \varphi_n(n) \uparrow \end{cases}$$

f is not computable because it differs from all computable functions. In fact

- if $\varphi_n(n) \downarrow$, then $f(n) = \varphi_n(n) + 1 \neq \varphi_n(n)$

- if $\varphi_n(n) \uparrow$, then $f(n) = 0 \neq \varphi_n(n)$

so

$$\forall n \ f \neq \varphi_n$$

OBSERVATION 10.5. There are infinitely many total non-computable functions of the following shape

$$f(n) = \begin{cases} \varphi_n(n) + k & n \in W_n \\ k & n \notin W_n \end{cases}$$

EXERCISE 10.6. Let $f : \mathbb{N} \to \mathbb{N}$, $m \in \mathbb{N}$. Show that there exists a non-computable function $g : \mathbb{N} \to \mathbb{N}$ such that

$$g(x) = f(x) \quad \forall x < m$$

Idea: use a "translated diagonal":

$$g(x) = \begin{cases} f(x) & x < m \\ \varphi_{x-m}(x) + 1 & x \geqslant m \text{ and } x \in W_{x-m} \\ 0 & x \geqslant m \text{ and } x \notin W_{x-m} \end{cases}$$

g is not computable since $g(x + m) \neq \varphi_x(x + m)$ for all x, so

$$\forall x \ g \neq \varphi_x$$

Another approach is to define g in the following way

$$g(x) = \begin{cases} f(x) & x < m \\ \varphi_x(x) + 1 & x \geqslant m \text{ and } x \in W_x \\ 0 & x \geqslant m \text{ and } x \notin W_x \end{cases}$$

because each function appears infinitely many times in the enumeration, and skipping the first $m - 1$ steps does not create any problem. Formally

$$\forall x \geqslant m \quad g \neq \varphi_x$$

so for all y

$$\forall y \ \exists x \geqslant m \ \varphi_y = \varphi_x$$

thus

$$\forall y \ \varphi_y \neq g$$

then g is not computable.

EXERCISE 10.7. Given a family of functions $\{f_i\}_{i \in \mathbb{N}}$ with $f_i : \mathbb{N} \to \mathbb{N}$, construct $g : \mathbb{N} \to \mathbb{N}$ such that $dom(g) \neq dom(f_i)$ for all $i \in N$

Idea:

$$g(n) = \begin{cases} 0 & \text{if } n \notin dom(f_n) \\ \uparrow & \text{if } n \in dom(f_n) \end{cases}$$

In this way

$$\forall n \ n \in dom(g) \Leftrightarrow n \notin dom(f_n)$$

EXERCISE 10.8. Define a non-computable total function that returns 0 when the input is even

Idea:

$$f(x) = \begin{cases} 0 & x \text{ is even} \\ \varphi_{\frac{x-1}{2}}(x) + 1 & x \text{ is odd, and } x \in W_{\frac{x-1}{2}} \\ 0 & x \text{ is odd, and } x \notin W_{\frac{x-1}{2}} \end{cases}$$

it is total not computable. In fact

- if $2n + 1 \in W_n \Rightarrow f(2n + 1) = \varphi_n(2n + 1) + 1 \neq \varphi_n(2n + 1)$

- if $2n + 1 \notin W_n \Rightarrow f(2n + 1) = 0 \neq \varphi_n(2n + 1) \uparrow$

so

$$\forall n \ f(2n + 1) \neq \varphi_n(2n + 1)$$

CHAPTER 11

Parametrisation theorem

We start by giving an intuition on what the theorem is about. Let $f : \mathbb{N}^2 \to \mathbb{N}$ be a computable function. Then there exists $e \in \mathbb{N}$ such that

$$f(x,y) = \varphi_e^{(2)}(x,y)$$

Now, if we fix the first argument to some value $x \in \mathbb{N}$, we obtain a function of a single argument $f_x : \mathbb{N} \to \mathbb{N}$

$$f_x(y) = f(x,y) = \varphi_e^{(2)}(x,y)$$

and for all $x \in \mathbb{N}$, f_x is computable (since it is obtained as composition of computable functions). This means that there exists a $d \in \mathbb{N}$ such that

$$f_x = \varphi_d$$

in other words, for all $y \in \mathbb{N}$

$$f_x(y) = \varphi_e^{(2)}(x,y) = \varphi_d(y)$$

Clearly d depends on e and x. Thus there is a total function $s : \mathbb{N}^2 \to \mathbb{N}$ such that

$$s(e,x) = d$$

i.e., for all $e, x, y \in \mathbb{N}$ it holds $\varphi_e^{(2)}(x,y) = \varphi_{s(e,x)}(y)$.

The *smn* theorem additionally tells us that s is computable.

Intuitively, how can we compute $s(e,x)$?

- get the program $P_e = \gamma^{-1}(e)$ that computes $\varphi_e^{(2)}(x,y)$
- get the program that computes $f_x = \lambda y \,.\, f(x,y)$ with fixed x, from P_e:
  - move y to R_2;
  - write x on R_1;
  - execute P_e
- take the code of the obtained program

Functions on indices, like s, are functions that transform programs. The *smn* theorem states that the operation of fixing an argument of a program is effective.

EXAMPLE 11.1. Consider the computable function

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

We know that there is an index such that $\varphi_d = f_x$, i.e.,

$$\varphi_d(x, y) = f(x, y) = x^y$$

So, when x varies we obtain computable functions

$$f_0(y) = y^0 = 1 \quad \text{with index} \quad s(e, 0)$$

$$f_1(y) = y^1 = y \quad \text{with index} \quad s(e, 1)$$

$$f_2(y) = y^2 \quad \text{with index} \quad s(e, 2)$$

$$\ldots$$

by *smn* theorem we can determine those indices in an effective way.

In its general form, the theorem works for functions of the form $f(\vec{x}, \vec{y}) : \mathbb{N}^{m+n} \to \mathbb{N}$ whence the name.

11.1. *smn* Theorem

THEOREM 11.2 (*smn* theorem). *Given $m, n \geqslant 1$ there is a computable total function*

$$s_{m,n} : \mathbb{N}^{m+1} \to \mathbb{N}$$

such that $\forall e \in \mathbb{N}, \ \vec{x} \in \mathbb{N}^m, \vec{y} \in \mathbb{N}^n$

$$\varphi_e^{(m+n)}(\vec{x}, \vec{y}) = \varphi_{s_{m,n}(e, \vec{x})}^{(n)}(\vec{y})$$

PROOF. Intuitively, given $e \in \mathbb{N}, \vec{x} \in \mathbb{N}^m$

- we get the program $P_e = \gamma^{-1}(e)$ in standard form that computes $\varphi_e^{(m+n)}$, so starting from

  | $\vec{x}$ | $\vec{y}$ | 0 | 0 | $\ldots$ |

  it computes $\varphi_e^{(m+n)}(\vec{x}, \vec{y})$

- from P_e we can build a new program P'. Starting from

  | $\vec{y}$ | 0 | 0 | $\ldots$ |

  it computes $\varphi_e^{(m+n)}(\vec{x}, \vec{y})$

In fact, it is sufficient to

- move $\vec{y}$ forward of m registers
- load $\vec{x}$ in the free m registers
- execute P_e

The program P' can be

$$T(n, m + n)$$
$$\dots$$
$$T(1, m + 1)$$
$$z(1)$$
$$s(1)$$

...   `// ` x_1 ` times`

$$s(1)$$

$$\vdots$$

$$z(m)$$
$$s(m)$$

...   `// ` x_m ` times`

$$s(m)$$

$$P_e$$

where concatenation has to update all the jump instructions in P_e, $J(m', n', t) \rightsquigarrow J(m', n', t + m + n + \sum_{i=1}^{m} x_i)$

Once P has been built, we have

$$s(e, \vec{x}) = \gamma(P')$$

Each function and construction method used are effective (so are γ, γ^{-1}). Thus, the existence, totality and computability of s are informally proven.

The formal proof of computability is long, but not difficult. We next provide just some hints. We first discuss how to define some auxiliary functions and then we use them to construct the smn-function.

Update function. Consider

$$upd : \mathbb{N}^2 \to \mathbb{N}$$

where $upd(e, h)$ is the code of a program obtained from $P_e = \gamma^{-1}(e)$ by updating each jump instruction $J(m, n, t)$ to $J(m, n, t + h)$.

It is useful to define an auxiliary function that works on each single instruction encoded with β

$$\widetilde{upd} : \mathbb{N}^2 \to \mathbb{N}$$

where $\widetilde{upd}(i, h)$ is the code of the instruction $\beta^{-1}(i)$, updated when it is a jump instruction.

Given $i, h \in \mathbb{N}$ and $q = qt(4, i), r = rm(4, i)$ it is formally defined in this way

$$\widetilde{upd}(i, h) = \begin{cases} 4 * \nu(\nu_1(q), \nu_2(q), \nu_3(q) + h) + 3 & r = 3 \\ i & r \neq 3 \end{cases}$$
$$= sg(r - 3) \cdot i + \bar{s}g(r - 3) \cdot (4 * \nu(\nu_1(q), \nu_2(q), \nu_3(q) + h) + 3)$$

Now

$$upd(e,t) = \tau(\widetilde{upd}(a(e,1),h), \ldots, \widetilde{upd}(a(e,l(e)),h))$$

$$= \left(\prod_{i=1}^{l(e)-1} p_i^{\widetilde{upd}(a(e,i),h)} \right) \cdot p_{l(e)}^{\widetilde{upd}(a(e,l(e)),h)+1} - 2$$

Concatenation of sequences. We will need a function

$$c : \mathbb{N}^2 \to \mathbb{N}$$

to concatenate sequences

$$c(e_1,e_2) = \tau(a(e_1,1), \ldots, a(e_1,l(e_1)), a(e_2,1), \ldots, a(e_2,e(e_2))) =$$

$$= \prod_{i=1}^{l(e_1)} p_i^{a(e_1,i)} \cdot \prod_{j=1}^{l(e_2)-1} p_{l(e_1)+j}^{a(e_1,j)} \cdot p_{l(e_1)+l(e_2)}^{a(e_2,l(e_2)+1)} - 2$$

Concatenation of programs.

$$seq : \mathbb{N}^2 \to \mathbb{N}$$

where

$$seq(e_1,e_2) = \gamma \left(\begin{array}{c} P_{e_1} \\ P_{e_2} \end{array} \right) = c(e_1, upd(e_2, l(e_2)))$$

Transfer. Shift registers $R_1, \ldots R_m$ of n positions forward

$$transf : \mathbb{N}^2 \to \mathbb{N}$$

where

$$transf(m,n) = \gamma(T([1,n],[m+1,m+n]))$$

$$= \tau(\beta(T(1,m+1)), \ldots, \beta(T(n,n+m)))$$

$$= \prod_{i=1}^{n-1} p_i^{\widetilde{\beta(T(i,m+i))}} \cdot p_n^{\beta(T(n,m+n))+1} - 2$$

$$= \prod_{i=1}^{n-1} p_i^{4*\pi(i-1,m+i-1)} \cdot p_n^{\pi(n-1,m+n-1)+1} - 2$$

Set. Set a register R_i to a value x

$$set : \mathbb{N}^2 \to \mathbb{N}$$

$$set(i,x) = \gamma \begin{pmatrix} z(i) \\ s(i) \\ \vdots \\ s(i) \end{pmatrix}$$

$$= \tau(\beta(z(i)), \beta(s(i)), \ldots, \beta(s(i)))$$

$$= p_1^{\beta(Z(1))} \cdot \prod_{i=1}^{x-1} p_{i+1}^{\beta(S(i))} \cdot p_{x+1}^{\beta(S(i))+1} - 2$$

$$= p_1^{4*(i-1)} \cdot \prod_{i=1}^{x-1} p_{i+1}^{4*(i-1)+1} \cdot p_{x+1}^{4*(i-1)+2} - 2$$

Proof of the fact that the smn function is computable We can now conclude that the smn-function is computable by composition. Just define:

$$pref_{m,n} : \mathbb{N}^m \to \mathbb{N}$$

where

$$pref_{m,n}(\vec{x}) = seq(transf(m,n), seq(set(1,x_1), \ldots, seq(\ldots, set(m,x_m))) \ldots)$$

Then we have that

$$s_{m,n} : \mathbb{N}^{m+1} \to \mathbb{N}$$

$$s_{m,n}(e, \vec{x}) = seq(pref_{m,n}(\vec{x}), e)$$

which is in $\mathcal{PR}$ □

OBSERVATION 11.3. The proof above proves that the *smn*-function is not only computable and total, but also primitive recursive.

The theorem is usually presented in the following simpler shape.

COROLLARY 11.4 (Simplified *smn* theorem). *Let* $f : \mathbb{N}^{m+n} \to \mathbb{N}$ *be a computable function. There exists a total computable function* $s : \mathbb{N}^m \to \mathbb{N}$ *such that*

$$f(\vec{x}, \vec{y}) = \varphi_{s(\vec{x})}^{(n)}(\vec{y}) \qquad \forall x \in \mathbb{N}^m \quad \forall y \in \mathbb{N}^n$$

PROOF. Since f is computable, given $e \in \mathbb{N}$ and $s(\vec{x}) = s_{m,n}(e, \vec{x})$

$$f(\vec{x}, \vec{y}) = \varphi_e^{(m+n)}(\vec{x}, \vec{y})$$

$$= \varphi_{s_{m,n}(e,\vec{x})}^{(n)}(\vec{y})$$

$$= \varphi_{s(\vec{x})}^{(n)}(\vec{y})$$

□

11.1.1. The *smn* theorem at work.

EXAMPLE 11.5. Prove that there exists a total computable function $k : \mathbb{N} \to \mathbb{N}$ such that for all $n, x \in \mathbb{N}$

$$\varphi_{k(n)}(x) = \lfloor \sqrt[n]{x} \rfloor$$

This means that φ_k is an enumeration of functions of the form $\lfloor \sqrt[n]{x} \rfloor$. To put it differently, φ_k is a function that given n, it returns the program that computes $\lfloor \sqrt[n]{x} \rfloor$.

PROOF. We define $f : \mathbb{N}^2 \to \mathbb{N}$

$$f(n, x) = \lfloor \sqrt[n]{x} \rfloor = \mu y \leqslant x \quad \text{``} (y+1)^n > x \text{''}$$
$$= \mu y \leqslant x \, . \, (x + 1 \dot{-} (y+1)^n)$$

The function f is computable because it is a bounded minimalisation of a composition of known computable functions. By the smn-theorem (Corollary 11.4), there exists $k : \mathbb{N} \to \mathbb{N}$ total computable such that for all $n, x \in \mathbb{N}$

$$\varphi_{k(n)}(x) = f(n, x) = \lfloor \sqrt[n]{x} \rfloor$$

$\square$

EXAMPLE 11.6. There exists $k : \mathbb{N} \to \mathbb{N}$ computable and total such that for all $n \in \mathbb{N}$, $\varphi_{k(n)}$ is defined only on n^{th} powers, i.e.

$$W_{k(n)} = \{ \, x \mid \exists y \in \mathbb{N} \, . \, x = y^n \, \}$$

PROOF. We define $f : \mathbb{N}^2 \to \mathbb{N}$ as

$$f(n, x) = \begin{cases} \sqrt[n]{x} & \text{if } \exists y \in \mathbb{N} \, . \, x = y^n \\ \uparrow & \text{otherwise} \end{cases}$$
$$= \mu y \, . \, |y^n - x|$$

It is computable. By the smn-theorem (Corollary 11.4), there exists $k : \mathbb{N} \to \mathbb{N}$ total computable such that for all $n, x \in \mathbb{N}$

$$\varphi_{k(n)}(x) = f(n, x)$$

We claim

$$W_{k(n)} = \{ \, x \mid \exists y \in \mathbb{N} \, . \, x = y^n \, \}$$

in fact, $x \in W_{k(n)}$ iff $\varphi_{k(n)}(x) \downarrow$ iff $f(n, x) \downarrow$ iff x is a nth power. $\square$

EXERCISE 11.7. Prove that there exists a function $s : \mathbb{N} \to \mathbb{N}$ which is total and computable such that

$$W^{(k)}_{s(x)} = \{(y_1, \ldots, y_k) \mid \sum_{i=1}^{k} y_i = x)\}$$

Idea: Define

$$f(x, \vec{y}) = \begin{cases} 0 & \sum_{i=1}^{k} y_i = x \\ \uparrow & \text{otherwise} \end{cases}$$

$$= \mu z \cdot \left| \left(\sum_{i=1}^{k} y_i \right) - x \right|$$

and then use the smn theorem to conclude.

CHAPTER 12

Universal Function

We now discuss how the theory developed up to now allows us to prove the computability of a universal function, i.e., a function which, roughly speaking, embodies every computable function of a given arity.

For instance, for arity 1, the universal function is $\Psi_U : \mathbb{N}^2 \to \mathbb{N}$

$$\Psi_U(x, y) = \varphi_e(y)$$

It captures all unary computable functions $\varphi_1, \varphi_2, \dots$. In fact, for all $e \in \mathbb{N}$

$$g(y) = \Psi_U(e, y) = \varphi_e(y) \quad \rightsquigarrow \quad g = \varphi_e$$

so Ψ_U represents all the computable functions of the form $\mathbb{N} \to \mathbb{N}$.

More generally, we have the following defition.

DEFINITION 12.1. The universal function for k-ary functions (with $k \geqslant 1$) is defined as

$$\Psi_U^{(k)} : \mathbb{N}^{k+1} \to \mathbb{N}$$
$$\Psi_U(e, \vec{x}) = \varphi_e^{(k)}(\vec{x})$$

The fact that it is computable means that there is a program $P_\mathcal{U}$ which is able to reproduce the behaviour of all programs of a fixed arity k (the Universal Computer [Dav11]). While on the one hand this could seem strange and surprising, if we look at it closely, such a program receives in input

- e (the index of the program, a *description* of the program P_e to run)

- $\vec{x}$ the arguments

Hence it is an interpreter.

THEOREM 12.2. *The universal function* $\Psi_U^{(k)}$ *is computable.*

PROOF. Fixed $k \geqslant 1$, $e \in \mathbb{N}$ and $\vec{x} \in \mathbb{N}^k$ we want $\Psi_U(e, \vec{x}) = \varphi_e^{(k)}(\vec{x})$.

Idea:

- get the program $P_e = \gamma^{-1}(e)$;

- execute P_e on input $\vec{x}$;

- if $P_e(\vec{x}) \downarrow$, the value $\Psi_U(e, \vec{x})$ is in R_1, otherwise the program correctly diverges.

All operations involved are effective, hence, by Church-Turing thesis, the function $\Psi_U^{(k)}$ is computable.

The above argument is too informal and vague to be satisfactory. We next hint at the formal proof.

We need to encode the content of the memory. Consider

r_1	r_2	r_3	0	$\ldots$

the *configuration of registers* is given by

$$c = \prod_{i \geqslant 1} p_i^{r_i}$$

From the encoding we can obtain the value of each register as $r_i = (c)_i$.

Then we show that we can simulate the execution steps of a program by using only computable functions. More precisely we define the following functions:

- $c_k : \mathbb{N}^{k+2} \to \mathbb{N}$
  $c_k(e, \vec{x}, t) = $ configuration after t steps of computation of $P_e(\vec{x})$, if P_e does not stop on $\vec{x}$ in t or fewer steps; instead, it is the final configuration if $P_e(\vec{x})$ stops in t or fewer steps.

- $j_k : \mathbb{N}^{k+2} \to \mathbb{N}$
  $j_k(e, \vec{x}, t) = $ number of instructions to be executed after t steps of $P_e(\vec{x})$; instead it is 0 if $P_e(\vec{x})$ stops in t or fewer steps.

Now observe that

- if $P_e(\vec{x}) \downarrow$, then it stops in $\mu t. j_k(e, \vec{x}, t)$ steps, so

$$\varphi_e^{(k)}(\vec{x}) = (c_k(e, \vec{x}, \mu t. j_k(e, \vec{x}, t)))_1$$

- if $P_e(\vec{x}) \uparrow$, then $\mu t \,.\, j_k(e, \vec{x}, t) \uparrow$, hence

$$\varphi_e^{(k)}(\vec{x}) \uparrow = (c_k(e, \vec{x}, \mu t. j_k(e, \vec{x}, t)))_1$$

Hence in all cases

$$\Psi_U(e, \vec{x}) = \varphi_e^{(k)}(\vec{x}) = (c_k(e, \vec{x}, \mu t. j_k(e, \vec{x}, t)))_1$$

Therefore, if we prove that c_k, j_k are computable, we can conclude that $\Psi_U^{(k)}$ is also computable.

We proceed in the same way we did in the proof of Theorem 7.5, by proving that $c_k, j_k \in \mathcal{PR}$ (in fact, computability of c_k, j_k proved here implies the computatbility of c_P, j_P with a *fixed* program P, as needed in the proof of Theorem 7.5).

We build these function out of smaller components:

(a) Arguments of an instruction

Given $i \in \mathbb{N}$ instruction code ($i = \beta(\text{Instruction})$)

$$Z_{arg}(i) = qt(4, i) + 1$$
$$S_{arg}(i) = qt(4, i) + 1$$
$$T_{arg_h}(i) = \pi_h(qt(4, i)) + 1 \quad h \in \{1, 2\}$$
$$J_{arg_h}(i) = \nu_h(qt(4, i)) + 1 \quad h \in \{1, 2, 3\}$$

(b) Effect of executing an algebraic instruction on a configuration

$$zero(c, n) = qt(p_n^{(c)_n}, c)$$
$$succ(c, n) = p_n \cdot c$$
$$transf(c, m, n) = p_n^{(c)_m} \cdot zero(c, n)$$

(c) Effect on the configuration of registers of the execution of the instruction with code i

$$change(c, i) = \begin{cases} zero(c, Z_{arg}(i)) & rm(4, i) = 0 \\ succ(c, S_{arg}(i)) & rm(4, i) = 1 \\ transf(c, T_{arg_1}(i), T_{arg_2}(i)) & rm(4, i) = 2 \\ c & rm(4, i) = 3 \end{cases}$$

(d) Configuration of the registers starting from c, after executing instruction t of program P_e

$$nextconf(e, c, t) = \begin{cases} change(c, a(e, t)) & 1 \leqslant t \leqslant \ell(e) \\ c & \text{otherwise} \end{cases}$$

(e) Number of next instruction if we execute $i = \beta(\text{Instruction})$ and this is in position t of the program

$$ni(c, i, t) = \begin{cases} t + 1 & rm(4, i) \neq 3 \vee (rm(4, i) = 3 \wedge (c)_{J_{arg_1}(i)} \neq (c)_{J_{arg_2}(i)}) \\ J_{arg_3}(i) & \text{otherwise} \end{cases}$$

(f) next instruction, if we execute instruction t in a program P_e starting from configuration c

$$nextinstr(e, c, t) = \begin{cases} ni(c, a(e, t), t) & 1 \leqslant t \leqslant \ell(e) \wedge 1 \leqslant ni(c, a(e, t), t) \leqslant \ell(e) \\ 0 & \text{otherwise} \end{cases}$$

Now

$$c_k(e, \vec{x}, 0) = \prod_{i=1}^{k} p_i^{x_i}$$
$$j_k(e, \vec{x}, 0) = 1$$
$$c_k(e, \vec{x}, t + 1) = nextconf(e, c_k(e, \vec{x}, t), j_k(e, \vec{x}, t))$$
$$j_k(e, \vec{x}, t + 1) = nextinstr(e, c_k(e, \vec{x}, t), j_k(e, \vec{x}, t))$$

they are defined by primitive recursion of computable functions, therefore c_k, j_j are computable (actually, since all the involved functions are $\mathcal{PR}$ they also are in $\mathcal{PR}$). Thus,

$$\Psi_U(e, \vec{x}) = (c_k(e, \vec{x}, \mu t \,.\, j_k(e, \vec{x}, t)))_1$$

is computable. $\square$

As a corollary, we obtain the decidability of two predicates that will be really useful in the next chapters.

COROLLARY 12.3. *The following predicates are decidable:*

  (a) $H_k(e, \vec{x}, t) \equiv$ *"$P_e(\vec{x}) \downarrow$ in t or less steps"*

  (b) $S_k(e, \vec{x}, y, t) \equiv$ *"$P_e(\vec{x}) \downarrow y$ in t or less steps"*

PROOF.   (a) The characteristic function

$$\chi_{H_k}(e, \vec{x}, t) = \begin{cases} 1 & \text{if } H_k(e, \vec{x}, t) \\ 0 & \text{otherwise} \end{cases}$$
$$= \overline{sg}(j_k(e, \vec{x}, t))$$

  it is computable by composition.

  (b) The characteristic function

$$\chi_{S_k}(e, \vec{x}, y, t) = \chi_{H_k}(e, \vec{x}, t) \cdot \overline{sg}(|(c_k(e, \vec{x}, t))_1 - y|)$$

  it is computable by composition.

                                                                                                          $\square$

If $k = 1$ we will usually omit it.

Also, from the theorem we deduce the possibility to express every computable function in Kleene Normal Form (KNF).

COROLLARY 12.4 (Kleene Normal Form). *For every $e, k \in \mathbb{N}$ and $x \in \mathbb{N}^k$*

$$\varphi_e^{(k)}(x) = (\mu z \,.\, |\chi_{S_k}(e, \vec{x}, (z)_1, (z)_2) - 1|)_1$$

OBSERVATION 12.5.   i. This corollary highlights that each computable function can be obtained from primitive recursion functions using minimimalisation at most once (we need to use `while` statements, but one is sufficient).

     ii. Minimixmalisation allows us to "search" a single value that has a certain property. The one we used is a technique to search pairs of values generalizable to tuples.

12.1. Applications

We observed that if $f : \mathbb{N} \to \mathbb{N}$ is a total computable injective function, then

$$f^{-1}(y) = \begin{cases} x & \text{if exists } y \text{ s.t. } f(x) = y \\ \uparrow & \text{otherwise} \end{cases}$$

is computable since $f^{-1} = \mu x \ . \ |f(x) - y|$. The hypothesis of *totality* can be omitted.

EXERCISE 12.6. Let $f : \mathbb{N} \to \mathbb{N}$ computable and injective. Then $f^{-1} : \mathbb{N} \to \mathbb{N}$ is computable.

PROOF. Since f is computable, there exists $e \in \mathbb{N}$ such that $\varphi_e = f$. Now it is sufficient to observe that

$$f^{-1}(y) = (\mu w \ . \ |\chi_S(e, (w)_1, x, (w)_2) - 1|)_1$$

$\square$

We can also identify other non-computable functions and undecidable predicates:

EXERCISE 12.7. The statement "φ_x is total" is undecidable

PROOF. Let $Tot(x)$ be the predicate

$$Tot(x) \equiv \text{``}\varphi_x \text{ is total''}$$

and assume that it is decidable. Define

$$f(x) = \begin{cases} \varphi_x(x) + 1 & \varphi_x \text{ total} \\ 0 & \text{otherwise} \end{cases}$$

it is total. For every x if φ_x is total, then $\varphi_x \neq f$, since

$$f(x) = \varphi_x(x) + 1 \neq \varphi_x(x)$$

so f it is not computable. But we can write $f(x)$ as

$$f(x) = (\mu w.(S(x, x, (w)_1, (w)_2)) \wedge Tot(x) \wedge (w)_3 = (w)_2 + 1)$$
$$\vee ((w)_3 = 0 \wedge \neg Tot(x))$$

i.e., as the minimalisation and composition of computable functions, which would imply that it is computable. Absurd. $\square$

OBSERVATION 12.8 (Halting problem). The same technique applies to prove that the following predicates are undecidable:

- $P_1(x) \equiv \text{``}x \in W_x\text{''} \equiv \text{``}\varphi_x(x) \downarrow \text{''}$

- $P_2(x, y) \equiv \text{``}y \in W_x\text{''} \equiv \text{``}\varphi_x(y) \downarrow \text{''}$

12.2. Effective operations on computable functions

The existence of the universal function, together with the *smn* theorem allows us to formalise operations that manipulate programs and derive their effectiveness.

PROPOSITION 12.9 (Effectiveness of product). *There exists a function* $s : \mathbb{N}^2 \to \mathbb{N}$ *total and computable such that for every* $x, y \in \mathbb{N}$

$$\varphi_{s(x,y)} = \varphi_x \cdot \varphi_y$$

PROOF. We define a function $g : \mathbb{N}^3 \to \mathbb{N}$

$$\begin{aligned} g(x, y, z) &= \varphi_x(z) \cdot \varphi_y(z) \\ &= \Psi_U(x, z) \cdot \Psi_U(y, z) \end{aligned}$$

it is computable since it arises as composition of computable functions. By the *smn* theorem there exists $s : \mathbb{N}^2 \to \mathbb{N}$ total computable such that for every x, y, z

$$\varphi_{s(x,y)}(z) = g(x, y, z) = \varphi_x(z) \cdot \varphi_y(z)$$

thus

$$\varphi_{s(x,y)} = \varphi_x \cdot \varphi_y$$

□

PROPOSITION 12.10 (Effectiveness of squaring). *There exists* $k : \mathbb{N} \to \mathbb{N}$ *total and computable such that, for every* $x \in \mathbb{N}$,

$$\varphi_{k(x)} = \varphi_x^2$$

PROOF. $k(x) = s(x, x)$ □

PROPOSITION 12.11 (Effectiveness of primitive recursion). *Recall the notion of primitive recursion*

$$h(\vec{x}, 0) = f(\vec{x})$$
$$h(\vec{x}, y + 1) = g(\vec{x}, y, f(\vec{x}, y))$$

We know that if f, g *are computable then* h *is computable. We can derive that there exists* $r : \mathbb{N}^2 \to \mathbb{N}$ *total computable such that, if* $f = \varphi_{e_1}^{(k)}$ *and* $g = \varphi_{e_2}^{(k+2)}$, *then*

$$h = \varphi_{r(e_1, e_2)}^{(k+1)}$$

PROPOSITION 12.12 (Effectiveness of the inverse function). *There exists* $k : \mathbb{N} \to \mathbb{N}$ *total and computable such that*

$$\forall x \in \mathbb{N} \quad \text{if } \varphi_x \text{ is injective } \Rightarrow \varphi_{k(x)} = (\varphi_x)^{-1}$$

PROOF. We define a function $g : \mathbb{N}^2 \to \mathbb{N}$

$$\begin{aligned} g(x, y) &= (\varphi_x)^{-1}(y) \\ &= \begin{cases} z & \exists z \text{ s.t. } \varphi_x(z) = y \\ \uparrow & \text{otherwise} \end{cases} \\ &= (\mu\omega \, . \, |\chi_{S(x,(\omega)_1, y,(\omega)_2)} - 1|)_1 \end{aligned}$$

it is computable by minimalisation. Hence, by *smn* theorem, there is a $k : \mathbb{N} \to \mathbb{N}$ total and computable such that for every x, y

$$\varphi_{k(x)}(y) = g(x, y) = (\varphi_x)^{-1}(y)$$

$\square$

PROPOSITION 12.13. *There is a total computable function $s : \mathbb{N}^2 \to \mathbb{N}$ such that, for every x, y*

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

PROOF. We want $\varphi_{S(x,y)}(z) \downarrow$ iff $\varphi_x(z) \downarrow$ or $\varphi_y(z) \downarrow$. We define a function $g : \mathbb{N}^3 \to \mathbb{N}$

$$g(x, y, z) = \begin{cases} 1 & z \in W_x \vee z \in W_y \\ \uparrow & \text{otherwise} \end{cases}$$

which is computable:

$$g(x, y, z) = \mathbf{1}(\mu\omega \ . \ |\chi_{H(x,z,\omega) \wedge H(y,z,\omega)} - 1|)$$

Hence by *smn* theorem exists $s : \mathbb{N}^2 \to \mathbb{N}$ computable and total such that

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

$\square$

PROPOSITION 12.14. *There exists a $s : \mathbb{N}^2 \to \mathbb{N}$ computable and total such that*

$$\forall x, y \quad E_{s(x,y)} = E_x \cup E_y$$

PROOF. We want the value of $\varphi_{S(x,y)}$ to be the same of the functions $\varphi_x and \varphi_y$. In order to do this, we can simulate φ_x on even numbers and φ_y on odd numbers. We define a function $g : \mathbb{N}^3 \to \mathbb{N}$

$$g(x, y, z) = \begin{cases} \varphi_x(\frac{z}{2}) & \text{if } z \text{ even} \\ \varphi_y(\frac{z-1}{2}) & \text{if } z \text{ odd} \end{cases}$$

computable since

$$\begin{aligned} g(x, y, z) = \quad & (\mu\omega \ . \quad (S(x, z/2, (\omega)_1, (\omega)_2) \ \wedge \ z \text{ even}) \vee \\ & \qquad (S(y, (z-1)/2, (\omega)_1, (\omega)_2) \ \wedge \ z \text{ odd}))_1 = \\ & (\mu\omega \ . \quad |\max\{\chi_S(x, qt(2, z), (\omega)_1, (\omega)_2) \ \cdot \ \overline{sg}(rm(2, z)), \\ & \qquad \chi_S(y, qt(2, z), (\omega)_1, (\omega)_2) \ \cdot \ sg(rm(2, z))\} - 1|)_1 \end{aligned}$$

By *smn* theorem there exists $s : \mathbb{N}^2 \to \mathbb{N}$ computable and total such that

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

for every x, y, z. So

$$v \in E_{s(x,y)} \Leftrightarrow \exists z \ . \ \varphi_{S(x,y)}(z) = g(x, y, z) = v$$

$$\Leftrightarrow \exists z \ . \ \begin{cases} z \text{ even and} & \varphi_x(\frac{z}{2}) = v \\ z \text{ odd and} & \varphi_y(\frac{z-1}{2}) = v \end{cases}$$

$$\Leftrightarrow \exists z \ . \ \varphi_x(z) = v \wedge \varphi_y(z) = v \Leftrightarrow \omega \in E_x \cup E_y$$

$\square$

PROPOSITION 12.15. *There is $k : \mathbb{N} \to \mathbb{N}$ computable and total such that $E_{k(x)} = W_x$*

PROOF. Define

$$g(x,y) = \begin{cases} y & y \in W_x \\ \uparrow & \text{otherwise} \end{cases}$$

$$= \mathbb{1}(\Psi_U(x,y)) \cdot y$$

it is computable by composition, so by *smn* theorem there exists $k : \mathbb{N} \to \mathbb{N}$ computable and total such that, for every x, y

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

In other words

$$y \in E_{k(x)} \Leftrightarrow \varphi_{k(x)}(y) = y \Leftrightarrow g(x,y) = y \Leftrightarrow y \in W_x$$

$\square$

PROPOSITION 12.16. *Given $f : \mathbb{N} \to \mathbb{N}$ computable, there exists $k : \mathbb{N} \to \mathbb{N}$ computable and total such that, for every x, $W_{k(x)} = f^{-1}(W_x)$*

PROOF. Define
$$g(x,y) = \varphi_x(f(y)) = \Psi_U(x, f(y))$$
computable by definition. By the *smn* theorem, there exists $k : \mathbb{N} \to \mathbb{N}$ computable and total such that $\varphi_{k(x)}(y) = g(x,y)$. So

$$y \in W_{k(x)} \Leftrightarrow \varphi_{k(x)}(y) = g(x,y) = \varphi_x(f(y)) \downarrow$$
$$\Leftrightarrow f(y) \downarrow \text{ and } f(y) \in W_x$$
$$\Leftrightarrow y \in f^{-1}(W_x)$$

$\square$

PROPOSITION 12.17. *There exists $k : \mathbb{N} \to \mathbb{N}$ computable and total such that if $\varphi_x = \chi_Q$ is the characteristic function of a decidable predicate Q, then $\varphi_{k(x)} = \chi_{\neg Q}$*

PROOF. Define
$$g(x,y) = 1 \div \varphi_x(y) = 1 - \Psi_U(x,y)$$
which is computable by definition. By the *smn* theorem, there exists k computable and total such that
$$g(x,y) = \varphi_{k(x)}$$
In this way, if $\varphi_x = \chi_Q$
$$g(x,y) = 1 - \varphi_x(y) = \varphi_{k(x)}(y) = 1 \Leftrightarrow \varphi_x(y) = 0 \Leftrightarrow \chi_Q(y) = 0$$
therefore
$$\varphi_{k(x)} = \chi_{\neg Q}$$

$\square$

CHAPTER 13

Recursive sets

In previous chapters we spent most of our effort in identifying computable functions and decidable properties, and for devising tools and techniques for proving computability. Only in few cases we provided examples in the large classes of non-computable functions and undecidable predicates.

From now on we start a mathematical study of

- classes of undecidable predicates/non computable functions

- techniques to prove the undecidability of predicates/non-computability of functions

This will allow us to give a structure to the class of non-computable functions and single out general classes of problems which do not admit an algorithmic solution.

We will focus on *sets of numbers* $X \subseteq \mathbb{N}$ and on the corresponding membership problem "$x \in X$?". In most cases X will be seen as a set of program codes and thus it can be seen as a program property, e.g.

- $X = \{x \mid \varphi_x = fact\}$: the program is a correct implementation of the factorial function;

- $X = \{x \mid W_x = \mathbb{N}\}$: the program is defined on all inputs.

- $X = \{x \mid P_x \text{ has linear complexity}\}$: the program has a linear complexity.

- ...

We will distinguish between

- *recursive sets/decidable properties*: It is possible to answer "yes" when the property holds, "no" when the property does not hold.

- *recursively enumerable sets/semi-decidable properties*: It is possible to answer "yes" when the property holds, but no answer when the property does not hold.

13.1. Recursive sets

DEFINITION 13.1. A set $A \subseteq \mathbb{N}$ is *recursive* if its characteristic function

$$\chi_A : \mathbb{N} \to \mathbb{N}$$

$$\chi_A(x) = \begin{cases} 1 & x \in A \\ 0 & x \notin A \end{cases}$$

is computable.

In other words, if the predicate "$x \in A$" is decidable.

OBSERVATION 13.2. Note that

- if $\chi_A \in \mathcal{PR}$ we will say that A is *primitively* recursive.

- the notion can be extended to subsets of $\mathbb{N}^k$, but we will stick to subsets of $\mathbb{N}$, since every subset of $\mathbb{N}^k$ can be encoded into a subset of $\mathbb{N}$.

EXAMPLE 13.3. The following sets are recursive:

(a) $\mathbb{N}$, since $\chi_{\mathbb{N}} = \mathbf{1}$ is computable;

(b) $\varnothing$, because $\chi_{\varnothing} = \mathbf{0}$ is computable;

(c) prime numbers $\mathbb{P}$, since

$$Pr(x) = \begin{cases} 1 & \text{if } x \text{ is prime} \\ 0 & \text{otherwise} \end{cases}$$

is computable;

(d) All *finite sets*. In fact, given $A \subset \mathbb{N}$ with $|A| < \infty$, $A = \{x_1, x_2, \ldots, x_n\}$, we have that

$$\chi_A(x) = \overline{sg}\left(\prod_{i=1}^{n} |x - x_i|\right)$$

is computable.

On the other hand, the following sets are not recursive:

(a) $K = \{x \mid x \in W_x\}$, since

$$\chi_K(x) = \begin{cases} 1 & x \in W_x \\ 0 & x \notin W_x \end{cases}$$

is not computable;

(b) $\{x \mid \varphi_x \text{ total}\}$

OBSERVATION 13.4. If $A, B \subseteq \mathbb{N}$ are recursive, then

1) $\overline{A} = \mathbb{N} - A$

2) $A \cap B$

3) $A \cup B$

are recursive.

13.1.1. Reduction. Reduction is a simple but powerful tool when studying the decidability status of problems. It formalizes the intuition of a problem $\mathcal{A}$ being "easier" than another one, $\mathcal{B}$.

DEFINITION 13.5. Let $A, B \subseteq \mathbb{N}$. We say that the problem $x \in A$ *reduces* to the problem $x \in B$ (or simply that A reduces to B), written $A \leqslant_m B$ if there exists a function $f : \mathbb{N} \to \mathbb{N}$ computable and total such that, for every $x \in \mathbb{N}$

$$x \in A \quad \Leftrightarrow \quad f(x) \in B$$

In this case, we say that f is the *reduction function*.

OBSERVATION 13.6. Let $A, B \subseteq \mathbb{N}$ such that $A \leqslant_m B$ then

     1 if B is recursive, then A is recursive

     2 of A is not recursive, then B is not recursive

    PROOF. Simply observe that $\chi_A = \chi_B \circ f$.   □

We know that $K = \{x \mid x \in W_x\}$ is not recursive. We next observe see how the non-recursiveness of other sets can be proven by reduction to K.

EXAMPLE 13.7. $K \leqslant_m T = \{x \mid \varphi_x \text{ total}\}$

    PROOF. We prove that there exists $s : \mathbb{N} \to \mathbb{N}$ computable and total such that $x \in k \Leftrightarrow s(x) \in T$. In other words

$$x \in W_x \Leftrightarrow \varphi_{f(x)} \text{ is total}$$

To do so, we can define

$$g(x, y) = \begin{cases} 1 & x \in W_x \\ \uparrow & \text{otherwise} \end{cases}$$

which is computable, since

$$g(x, y) = \mathbf{1}(\varphi_x(x)) = \mathbf{1}(\Psi_U(x, x))$$

Then, by the *smn*-theorem we have that there exists $s : \mathbb{N} \to \mathbb{N}$ computable and total such that

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

and

$$x \in K \Rightarrow x \in W_x \Rightarrow \forall y \; \varphi_{s(x)}(y) = g(x, y) = 1 \Rightarrow \varphi_{s(x)} \text{ total} \Rightarrow s(x) \in T$$

$$x \notin K \Rightarrow x \notin W_x \Rightarrow \forall y \; \varphi_{s(x)}(y) = g(x, y) \uparrow \Rightarrow \varphi_{s(x)} \text{ not total} \Rightarrow s(x) \notin T$$

          □

EXAMPLE 13.8 (Input problem). For every $n \in \mathbb{N}$

$$A_n = \{x \mid \varphi_x(n) \downarrow\}$$

is not recursive.

    PROOF. We will prove that $K \leqslant A_n$. We have to define a function f s.t.

$$x \in K \Leftrightarrow f(x) \in A_n$$

i.e., $x \in W_x \Leftrightarrow \varphi_{f(x)}(n) \downarrow$.

Define

$$g(x, y) = \begin{cases} 1 & x \in W_x \\ \uparrow & \text{otherwise} \end{cases}$$
$$= \mathbf{1}(\Psi_U(x, x))$$

Function g is computable, and thus by the *smn*-theorem, there exists $f : \mathbb{N} \to \mathbb{N}$ computable and total such that $g(x, y) = \varphi_{f(x)}(y)$. It is now easy to show that s is the reduction function, i.e.,

$$x \in K \Rightarrow f(x) \in A_n$$
$$x \notin K \Rightarrow f(x) \notin A_n$$

$\square$

EXAMPLE 13.9 (The output problem). For every $n \in \mathbb{N}$, $B_n = \{x \mid n \in E_x\}$ is not recursive

PROOF. We show that $K \leqslant_m B_n$. Define the function

$$g(x, y) = \begin{cases} n & x \in W_x \\ \uparrow & \text{otherwise} \end{cases}$$
$$= n \cdot \mathbf{1}(\Psi_U(x, x))$$

Observe that g is computable. Hence by the *smn*-theorem there exists a function $s : \mathbb{N} \to \mathbb{N}$ such that

$$\forall x, y \quad g(x, y) = \varphi_{s(x)}(y)$$

It is now easy to show that s is the reduction function, i.e.,

$$x \in K \Rightarrow s(x) \in B_n$$
$$x \notin K \Rightarrow s(x) \notin B_n$$

$\square$

OBSERVATION 13.10. Let $A, B \subseteq \mathbb{N}$ with $A \leqslant_m B$ through an injective reduction function $f : \mathbb{N} \to \mathbb{N}$ (total and computable). One could think that, since f^{-1} is computable, then also $B \leqslant_m A$. This is clearly not the case since f^{-1} is not total and thus it reduces A to a "subproblem" of B (which typically have no clear relation with B).

Rice theorem

Rice's theorem gives a general undecidability result. It roughly states that *no property* of the behaviour of programs which is related to the input/output (besides the obvious ones) is decidable or, in other words, that no non-trivial property of computable functions is decidable.

Formally, we will need the notion of *saturated* set.

14.1. Saturated sets

DEFINITION 14.1 (Saturated set). A subset $A \subseteq \mathbb{N}$ is *saturated* (or *extensional*) if for all $x, y \in \mathbb{N}$

$$x \in A \wedge \varphi_x = \varphi_y \Rightarrow y \in A$$

In other words, A is saturated if it expresses a property of functions, independently from indices

$$A = \{x \mid P(\varphi_x)\}$$

or, again, if there exists $\mathcal{A} \subseteq \mathcal{C}$ such that

$$A = \{x \mid \varphi_x \in \mathcal{A}\}$$

EXAMPLE 14.2. The following set is saturated

$$T = \{n \mid P_n \text{ always terminate}\}$$
$$= \{n \mid \phi_n \in \mathcal{T}\}$$

where

$$\mathcal{T} = \{f \mid f \text{ is total}\}$$

EXAMPLE 14.3. The following set is saturated

$$ONE = \{n \mid P_n \text{ computes } \mathbf{1}\}$$
$$= \{n \mid \phi_n = \mathbf{1}\}$$
$$= \{n \mid \phi_n \in \{\mathbf{1}\}\}$$

EXAMPLE 14.4. Consider

$$T_2 = \{e \mid P_e(e) \downarrow \text{ in two steps }\}$$
$$= \{e \mid \phi_e \in \mathcal{T}_2\}$$

two programs can compute the same function, one terminates in less than 2 steps and the other in more than 2. Thus, the set is not saturated.

EXAMPLE 14.5. Consider

$$K = \{e \mid e \in W_e\}$$
$$= \{e \mid \phi_e \in \mathcal{K}\}$$

where we would like

$$\mathcal{K} = \{f \mid ?\}$$

It is not saturated. We cannot give a formal proof yet. The proof will rely on the fact that one can show the existence of a program e such that

$$\phi_e(x) = \begin{cases} 0 & x = e \\ \uparrow & \text{otherwise} \end{cases}$$

Then $e \in K$. Moreover, since there are infinitely many programs for the same function, there is $e' \neq e$ such that $varphi_e = \varphi_{e'}$. Note that $\varphi'(e') = \varphi_e(e') \uparrow$, hence $e' \notin K$ and we conclude.

14.2. Rice's theorem

THEOREM 14.6 (Rice's theorem). *Let $A \in \mathbb{N}, A \neq \emptyset, A \neq \mathbb{N}$ be saturated. Then it is not recursive.*

PROOF. We show that $K \leqslant_m A$. Let e_0 such that $\phi_{e_0}(x) \uparrow \forall x$. We distinguish two cases depending on whether $e \in A$ or not.

($e_0 \notin A$) Suppose $e_0 \notin A$ and let $e_1 \in A(\neq \emptyset)$. Now define

$$g(x, y) = \begin{cases} \phi_{e_1}(y) & x \in K \\ \phi_{e_0}(y) & x \notin K \end{cases}$$
$$= \begin{cases} \phi_{e_1}(y) & x \in K \\ \uparrow & x \notin K \end{cases}$$
$$= \phi_{e_1}(y) \cdot \mathbf{1}(\Psi_U(x, x))$$

it is computable. By *smn* theorem there is $s\mathbb{N} \to \mathbb{N}$ such that $\phi_{s(x)}(y) = g(x, y)$.

Now observe that s is a reduction function for $K \leqslant_m A$

$- \ x \in K \Rightarrow \forall y \ \varphi_{s(x)}(y) = \varphi_{e_1}(y) \Rightarrow s(x) \in A$

$- \ x \notin K \Rightarrow \forall y \ \varphi_{s(x)}(y) = \varphi_{e_0}(y) \uparrow \Rightarrow s(x) \notin A$

Hence $K \leqslant_m A$, K not recursive, thus A.

($e_0 \in A$) If $e_0 \in A$ then $e_0 \notin \bar{A}$. Then $\bar{A} \subseteq \mathbb{N}, \bar{A} \neq \emptyset, \bar{A} \neq \mathbb{N}$, and $\bar{A}$ is saturated since A is. Therefore, by the first part,o $\bar{A}$ is not recursive, and therefore A is not recursive either.

$\square$

EXAMPLE 14.7 (Output problem). We proved that

$$B_n = \{e \mid n \in E_e\}$$

is not recursive by showing that $K \leqslant_m B_n$. We can conclude the same by observing

- B_n is saturated;

- $B_n \neq \varnothing$;

- $B_n \neq \mathbb{N}$.

By Rice's theorem B_n is not recursive.

Recursively enumerable sets

DEFINITION 15.1 (Recursively enumerable set). We say that $A \subseteq \mathbb{N}$ is *recursively enumerable* if the semi-characteristic function

$$sc_A(x) = \begin{cases} 1 & x \in A \\ \uparrow & \text{otherwise} \end{cases}$$

is computable.

DEFINITION 15.2 (Semi-decidable predicate). A predicate $Q(x) \subseteq \mathbb{N}$ is semi-decidable if $\{x \in \mathbb{N} \mid Q(x)\}$ is r.e.

Thus, saying that A is r.e. is like saying that the predicate $Q(x) = \text{"}x \in A\text{"}$ is semi-decidable. This notion is also easily generalisable to

- subsets of $\mathbb{N}^k$
- k-ary predicates

OBSERVATION 15.3. Let $A]subseteq\mathbb{N}$ be a set.

$$A \text{ recursive } \Leftrightarrow A, \bar{A} \text{ are r.e.}$$

PROOF. ($\Rightarrow$) If A recursive,

$$\mathcal{X}_A(x) = \begin{cases} 1 & x \in A \\ 0 & \text{otherwise} \end{cases}$$

is computable. Then $sc_A(x) = \mathbf{1}(\mu z.|\chi_A(x) - 1|)$ is computable, therefore A is r.e. Since A is recursive, then $\bar{A}$ is recursive, thus, r.e.

($\Leftarrow$) Let $A, \bar{A}$ be r.e., then by definition sc_A and $sc_{\bar{A}}$ are computable, and we can define

$$\mathbf{1} - sc_{\bar{A}}(x) = \begin{cases} 0 & x \in \bar{A} \\ \uparrow & \text{otherwise} \end{cases}$$

that is computable. This means that $\exists e_0, e_1 \in \mathbb{N}$ such that

$$\varphi_{e_0} = sc_A \quad \varphi_{e_1} = \mathbf{1} - sc_{\bar{A}}$$

therefore we can "combine two machines" and wait until one of the two terminates. Since either $x \in A$ or $x \in \bar{A}$, then the process will terminate for sure. We can build the characteristic function of A as

$$\chi_A(x) = (\mu\omega.|S(e_0, x, (\omega)_1, (\omega)_2 \wedge S(e_1, x, (\omega)_1, (\omega)_2)) - 1|)_1$$
$$(\mu\omega.|\chi_{S(e_0,x,(\omega)_1,(\omega)_2 \wedge S(e_1,x,(\omega)_1,(\omega)_2))} - 1|)_1$$

which is computable, therefore A is recursive.

$\square$

OBSERVATION 15.4. The set $K = \{x \mid x \in W_x\}$ is r.e. In fact

$$sc_K(x) = \begin{cases} 1 & x \in K \\ \uparrow & \text{otherwise} \end{cases} = \mathbf{1}(\varphi_x(x)) = \mathbf{1}(\Psi_U(x,x))$$

is computable by definition and by 15.3

$$\bar{K} = \{x \mid x \notin W_x\}$$

is *not* r.e, otherwise $K, \bar{K}$ would have been both r.e., and therefore K would have been recursive, which is a contradiction.

THEOREM 15.5 (Structure of semi-decidable predicates). *Let* $P(\vec{x}) \subseteq \mathbb{N}^k$ *be a predicate. Then* $P(\vec{x})$ *is decidable if and only if there is a decidable predicate* $Q(t, \vec{x}) \subseteq \mathbb{N}^{k+1}$ *such that* $P(\vec{x}) = \exists t. Q(t, \vec{x})$.

PROOF.   ($\Rightarrow$) Let $P(\vec{x})$ be semi-decidable. It has a computable semi characteristic function sc_P so

$$P(\vec{x}) \equiv \exists t. H(e, \vec{x}, t)$$

therefore if we can rewrite H as $Q(t, \vec{x}) = H(e, \vec{x}, t)$, in this way Q is decidable as we wanted and

$$P(\vec{x}) \equiv \exists t. Q(t, \vec{x})$$

($\Leftarrow$) Let $P(\vec{x}) \equiv \exists t. Q(t, \vec{x})$ with $Q(t, \vec{x})$ decidable. Observe that

$$sc_P(\vec{x}) = \mathbf{1}(\mu t. |\chi_Q(t, \vec{x}) - 1|)$$

which is computable by definition, and therefore $P(\vec{x})$ is semi-decidable.

$\square$

15.1. Projection theorem

From the last theorem we had a hint about the fact that the class of semi-decidable predicates is closed under *existential quantification*. The projection theorem states this:

THEOREM 15.6 (Projection theorem). *Let* $P(x, \vec{y})$ *be semi-decidable; then*

$$\exists x. P(x, \vec{y}) = P'(\vec{y})$$

is semi-decidable.

PROOF. Let $P(x, \vec{y})$ be semi-decidable. The by Theorem 15.5, there exists $Q(t, x, \vec{y})$ decidable such that

$$P(x, \vec{y}) \equiv \exists t. Q(t, x, \vec{y})$$

Thus

$$P'(\vec{y}) \equiv \exists x. P(x, \vec{y})$$
$$\equiv \exists x. \exists t. Q(t, x, \vec{y})$$
$$\equiv \exists \omega. Q((\omega)_1, (\omega)_2, \vec{y})$$

since $Q((\omega)_1, (\omega)_2, \vec{y})$ is decidable, by Theorem 15.5 $P'(\vec{y})$ is semi-decidable. $\qquad \square$

THEOREM 15.7 (Closure under conjuction and disjunction). *Let $P_1(\vec{x}), P_2(\vec{x})$ be semi-decidable predicates. Then*

- $P_1(\vec{x}) \vee P_2(\vec{x})$;

- $P_1(\vec{x}) \wedge P_2(\vec{x})$

are semi-decidable.

PROOF. Let $P_1(\vec{x}), P_2(\vec{x})$ be semi-decidable predicates. Then by Theorem 15.5 there are two decidable predicates $Q_1(t, \vec{x}), Q_2(t, \vec{x})$ such that

$$P_1(\vec{x}) \equiv \exists t. Q_1(t, \vec{x})$$
$$P_2(\vec{x}) \equiv \exists t. Q_2(t, \vec{x})$$

Hence

(1)

$$P_1(\vec{x}) \vee P_2(\vec{x}) \equiv \exists t. Q_1(t, \vec{x}) \vee \exists t. Q_2(t, \vec{x})$$
$$\equiv \exists \omega. (Q_1((\omega)_1, \vec{x}) \vee Q_2((\omega)_2, \vec{x}))$$

This means that by Theorem 15.5, $P_1(\vec{x}) \vee P_2(\vec{x})$ is semi-decidable.

(2) Analogously

$$P_1(\vec{x}) \wedge P_2(\vec{x}) \equiv \exists t. (Q_1(t, \vec{x}) \wedge Q_2(t, \vec{x}))$$

$\qquad \square$

OBSERVATION 15.8. The set of semi-decidable predicates is closed under $\wedge, \vee$ and $\exists$, but it is not closed under $\forall$ and $\neg$. For instance $P(x) = "x \in K"$ is semi-decidable, while $\neg P(x) = "x \notin K"$ is not. Moreover $Q(x, t) = \neg H(x, x, t)$ is decidable, while $\forall t. Q(x) = "x \notin K"$ is not semi-decidable.

EXERCISE 15.9. Prove that if $P(\vec{x})$ is semi-decidable and is not decidable then $\neg P(\vec{x})$ is not semi-decidable.

OBSERVATION 15.10.   (1) $A \subseteq \mathbb{N}$ is recursive if and only if $A, \bar{A}$ are r.e.

(2) if $A \subseteq \mathbb{N}$ r.e. and $f : \mathbb{N} \to \mathbb{N}$ computable $\Rightarrow f^{-1}(A)$ is r.e. (projection)

(3) $A, B \subseteq \mathbb{N}$ r.e. $\Rightarrow A \cup B, A \cap B$ are r.e.

15.1.1. r.e. sets and reducibility. Reduction can be used as a tool for comparing sets with respect to recursive enumerability as we already did for recursiveness.

OBSERVATION 15.11. Given $A, B \subseteq \mathbb{N}, A \leqslant_m B$, then

(1) B is r.e. $\Rightarrow$ A is r.e.

(2) A is not r.e. $\Rightarrow$ B not r.e.

PROOF.   (1) If B r.e., then

$$sc_B(x) = \begin{cases} 1 & x \in B \\ \uparrow & \text{otherwise} \end{cases}$$

is computable. Let $f : \mathbb{N} \to \mathbb{N}$ be a total computable reduction function for $A \leqslant_m B$. Then $sc_A(x) = sc_B(f(x))$, therefore sc_A is computable by composition and A is r.e.

(2) equivalent.

$\square$

CHAPTER 16

Rice-Shapiro theorem

Rice-Shapiro states that a property of the functions computed by programs can be semi-decidable **only if** it depends on a finite part of the function (I/O behavior on a finite number of inputs).

In order to properly state the theorem, we need some more tools.

DEFINITION 16.1 (Finite function). A finite function is a function $\theta : \mathbb{N} \to \mathbb{N}$ such that $dom(\theta)$ is finite.

The fact that a function is finite means that the set of input-output pairs is finite, i.e.,

$$\theta(x) = \begin{cases} y_1 & \text{if } x = x_1 \\ y_2 & \text{if } x = x_2 \\ \dots \\ y_n & \text{if } x = x_m \\ \uparrow & \text{otherwise} \end{cases}$$

In other words $\theta = \{(x_1, y_1), \dots (x_n, y_n)\}$, i.e., seen a set the function is finite.

DEFINITION 16.2. Given $f : \mathbb{N} \to \mathbb{N}$, θ is a sub-function of f if $\theta \subseteq f$

NOTATION 16.3. We recall some notation:

- W_e is the domain of the function φ_e;
- $E_e = \{\varphi_e(x) \mid x \in W_e\}$;
- $H(x, y, t) = $ "$P_x(y) \downarrow$ in t steps or less";
- $s(x, y, z, t) = $ "$P_x(y) \downarrow z$ in t steps or less";
- $K = \{x \mid x \in W_x\} = \{x \mid \varphi_x(x) \downarrow\} = \{x \mid P_x(x) \text{ terminates}\}$

THEOREM 16.4 (Rice-shapiro theorem). *Let $\mathcal{A} \subseteq \mathcal{C}$ be a set of computable functions. If the set $A = \{x \mid \varphi_x \in \mathcal{A}\}$ is r.e., then*

$$\forall f (f \in \mathcal{A} \Leftrightarrow \exists \theta \text{ finite function}, \theta \subseteq f \wedge \theta \in \mathcal{A})$$

PROOF. We will prove the following

(1) $\exists f \in \mathcal{C}.f \notin \mathcal{A} \wedge \exists \theta \subseteq f$ finite, $\theta \in \mathcal{A} \Rightarrow A$ not r.e

(2) $\exists f \in \mathcal{C}. f \in \mathcal{A} \wedge \forall \theta \subseteq f$ finite, $\theta \notin \mathcal{A} \Rightarrow A$ not r.e.

Here are the proofs:

(1) Let $f \notin \mathcal{A}$ and $\theta \subseteq f$ finite with $\theta \in \mathcal{A}$. We show that $\bar{K} \leqslant_m A$.

Define

$$g(x,y) = \begin{cases} \theta(y) & x \in \bar{K} \\ f(y) & x \in K \end{cases}$$

$$= \begin{cases} \uparrow & x \in \bar{K} \wedge x \notin dom(\theta) \\ \theta(y) = f(y) & x \in \bar{K} \wedge x \in dom(\theta) \\ f(y) & x \in K \end{cases}$$

$$= \begin{cases} f(y) & x \in K \vee y \in dom(\theta) \\ \uparrow & \text{otherwise} \end{cases}$$

Since $x \in K \vee y \in dom(\theta) = Q(x,y)$ predicate, $x \in K$ semi-decidable and $y \in dom(\theta)$ decidable, then $Q(x,y)$ semi-decidable. Then, since

$$sc_Q(x,y) = \begin{cases} 1 & Q(x,y) \\ 0 & \text{otherwise} \end{cases}$$

is computable, we have $g(x,y) = f(y) \cdot sc_Q(x,y)$ computable.

By *smn* theorem, there is a total computable function $s : \mathbb{N} \to \mathbb{N}$ such that, for every x, y

$$\varphi_{s(x)}(y) = g(x,y) = \begin{cases} \theta(y) & x \in \bar{K} \\ f(y) & x \in K \end{cases}$$

We show that s is the reduction function for $\bar{K} \leqslant_m A$

- $x \in \bar{K} \Rightarrow \forall y \; \varphi_{s(x)}(y) = g(x,y) = \theta(y) \Rightarrow \varphi_{s(x)} = \theta \in \mathcal{A} \Rightarrow s(x) \in A$

- $x \notin \bar{K} \Rightarrow x \in K \Rightarrow \forall y \; \varphi_{s(x)}(y) = g(x,y) = f(y) \Rightarrow \varphi_{s(x)} = f \notin \mathcal{A} \Rightarrow s(x) \notin A$

Since $\bar{K} \leqslant_m A$ and $\bar{K}$ is not r.e. we conclude that A is not r.e.

(2) Let $f \in \mathcal{A} \wedge \theta \subseteq f$ be with θ finite, $\theta \notin \mathcal{A}$

Informally, we want

$$g(x,y) = \begin{cases} f(y) & x \in \bar{K} \; (\varphi_x(x) \uparrow) \\ \theta(y) & \text{for some } \theta \subseteq f \text{ finite, otherwise } (x \in K) \end{cases}$$

More formally

$$g(x,y) = \begin{cases} f(y) & \text{if } \neg H(x,x,y) \\ \uparrow & \text{if } H(x,x,y) \end{cases}$$

$$= f(y) + \mu z.\chi_H(x,x,y)$$

is computable.

By *smn* there exists $s : \mathbb{N} \to \mathbb{N}$ total computable such that

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

We show that s is a reduction function for $\bar{K} \leqslant_m A$

- $x \in \bar{K}$
  $\Rightarrow \varphi_x(x) \uparrow$
  $\Rightarrow \forall y \ \neg H(x, x, y)$
  $\Rightarrow \forall y \ \varphi_{s(x)}(y) = g(x, y) = f(y)$
  $\Rightarrow f = \varphi_{s(x)} \in \mathcal{A}$
  $\Rightarrow s(x) \in A$

- $x \notin \bar{K}$
  $\Rightarrow x \in K$
  $\Rightarrow \varphi_x(x) \downarrow$
  $\Rightarrow \exists t_0 \ (\forall t > t_0 \ H(x, x, t) \wedge \forall t < t_0 \ \neg H(x, x, t))$
  $\Rightarrow \varphi_{s(x)}(y) = g(x, y)$
  $\Rightarrow \varphi_{s(x)} \subseteq f$ finite
  $\Rightarrow s(x) \in \bar{A}$

$\square$

EXAMPLE 16.5. $A = \{x \mid \varphi_x \text{ total}\}$ is not r.e.

PROOF. Clearly A is saturated since $A = \{x \mid \varphi_x \in \mathcal{A}\}$, and $\mathcal{A} = \{f \in \mathcal{C} \mid f \text{ total}\}$. Given any function $f \in \mathcal{A}$ (total by definition) we have that $\forall \theta \subseteq f$ finite clearly $\theta \notin \mathcal{A}$, since each and every finite function is partial, then by Rice-Shapiro's theorem, A is not r.e. $\square$

EXAMPLE 16.6. $\bar{A} = \{x \mid \varphi_x \text{ not total}\}$ is not r.e.

PROOF. Let $\bar{\mathcal{A}} = \{f \in \mathcal{C} \mid f \text{ not total}\}$. We observe that each θ finite is in $\bar{\mathcal{A}}$, but no total extension of such θ can be included in $\bar{\mathcal{A}}$. Again, by Rice-Shapiro $\bar{A}$ is not r.e. $\square$

Examples 16.5 and 16.6 characterise the two basic situations in which we can apply the theorem. They are generalised in the observation below.

OBSERVATION 16.7. Let $\mathcal{A} \subseteq \mathcal{C}$ be a set of computable functions s.t. $A = \{x \mid \varphi_x \in \mathcal{A}\}$ is r.e. Then

(1) if, for every θ finite, $\theta \notin \mathcal{A} \Rightarrow \mathcal{A} = \varnothing$

(2) $\varnothing \in \mathcal{A} \Rightarrow \mathcal{A} = \mathcal{C}$

PROOF. (1) Consider a generic $f \in \mathcal{C}$. We know that $f \in \mathcal{A}$ if and only if there exists $\theta \subseteq f$ finite $\theta \in \mathcal{A}$. Since no finite function is in $\mathcal{A}$ we conclude that $f \notin \mathcal{A}$. Hence $\mathcal{A} = \varnothing$.

(2) Consider a generic $f \in \mathcal{C}$. Since $\varnothing \subseteq f$ and $\varnothing \in \mathcal{A} \Rightarrow f \in \mathcal{A}$, Hence $\mathcal{A} = \mathcal{C}$.

$\square$

EXAMPLE 16.8. Consider $A = \{x \mid \varphi_x = \mathbf{1}\}$

(1) A is not r.e.

The set of functions is $\mathcal{A} = \{\mathbf{1}\}$, which

- does not contain finite functions

- is not empty

therefore A is not r.e.

(2) $\bar{A}$ is not r.e.

$\bar{\mathcal{A}} = \mathcal{C} - \{\mathbf{1}\}$, and we have that

- $\varnothing \in \bar{\mathcal{A}}$

- $\bar{\mathcal{A}} \neq \mathcal{C}$

therefore $\bar{A}$ is not r.e.

OBSERVATION 16.9. The converse implication of Rice-Shapiro theorem does not hold, i.e. the following does not hold

(16.1) $\forall f \ (\ f \in \mathcal{A}$ iff $\exists \theta$ finite, $\theta \subseteq f,\ \theta \in \mathcal{A}\ )$ $\Rightarrow$ A r.e.

In other words, Rice-Shapiro can be used to prove that a set is *not* r.e., but <u>not</u> to prove that a set is r.e.

For a counterexample to (16.1), define $\mathcal{A} = \{f \in \mathcal{C} \mid dom(f) \cap \bar{K} \neq \varnothing\}$ and let $A = \{x \mid \varphi_x \in \mathcal{A}\}$

(1) $\mathcal{A}$ satisfies the premise of (16.1)

$$f \in \mathcal{A} \Rightarrow dom(f) \cap \bar{K} \neq \varnothing$$
$$\Rightarrow \text{let } x \in dom(f) \cap \bar{k} \text{ we have that } \theta = \{(x, f(x))\}$$
$$\text{is finite, } \theta \subseteq f \text{ and } dom(\theta) \cap \bar{k} = \{x\} \neq \varnothing$$
$$\Rightarrow \theta \in \mathcal{A}$$

$$\text{if } \theta \text{ finite, } \theta \subseteq f, \theta \in \mathcal{A} \Rightarrow dom(\theta) \subseteq dom(f)$$
$$\Rightarrow dom(f) \cap \bar{K} \supseteq dom(\theta) \cap \bar{K} \neq \varnothing$$
$$\Rightarrow f \in \mathcal{A}$$

(2) A is not r.e., since $\bar{K} \leqslant_m A$

Define

$$g(x, y) = \begin{cases} 0 & x = y \\ \uparrow & \text{otherwise} \end{cases}$$
$$= \mu z. |x - y|$$

is computable. Again, by *smn* theorem $\exists s : \mathbb{N} \to \mathbb{N}$ computable and total such that

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

and therefore $dom(\varphi_{s(x)}) = \{x\}$. so

- $x \in \bar{K} \Rightarrow dom(\varphi_{s(x)}) \cap \bar{K} = \{x\} \neq \emptyset \Rightarrow s(x) \in A$
- $x \notin \bar{K} \Rightarrow dom(\varphi_{s(x)}) \cap \bar{K} = \{x\} = \emptyset \Rightarrow s(x) \notin A$

CHAPTER 17

First recursion theorem

In programming languages, we have higher-order functions that take other functions as arguments and produce functional results. E.g. in ML, the function `succ` that given a function f returns $f + 1$ can be defined as

$$fun\ succ\ f\ x = f\ x + 1$$

From the computability point of view it is still somewhat natural to ask how effective/computable operations can be characterized on functions. We will later see that this idea leads to the concept of *recursive functional*.

DEFINITION 17.1. Let $\mathcal{F}(\mathbb{N}^k)$ denote the set of all the functions (possibly not computable) of k arguments $\mathbb{N}^k \to \mathbb{N}^k$.

A *functional* is a total function

$$\Phi : \mathcal{F}(\mathbb{N}^k) \to \mathcal{F}(\mathbb{N}^h)$$

When can we say that a functional is effective (computable)? Given $\Phi : \mathcal{F}(\mathbb{N}^k) \to \mathcal{F}(\mathbb{N}^h)$

- a function $f \in \mathcal{F}$ and its image $\Phi(f) \in \mathcal{F}(\mathbb{N}^k)$ are both infinite objects in general.

- we cannot ask for $\Phi(f)$ to be effectively computable in a finite time from f

17.0.1. Encoding of finite functions. As a first step we need a way of viewing finite functions as numbers. The encoding of a finite function $\theta = \{(x_1, y_1), \ldots, (x_2, y_2)\}$ is $\tilde{\theta} \in \mathbb{N}$ defined as

$$\tilde{\theta} = \prod_{i=1}^{n} p_{x_i+1}^{y_i+1}$$

which is both injective and effective. Given the encoding of a function $z = \tilde{\theta}$,

$$x \in dom(\theta) \quad \text{iff} \quad (z)_{x+1} \neq 0$$

$$app(z, x) = \theta(x) = \begin{cases} (z)_{x+1} \dot{-} 1 & x \in dom(\theta) \\ \uparrow & \text{otherwise} \end{cases}$$
$$= ((z)_{x+1} \dot{-} 1)\ \mathbb{1}(\mu\omega\ .\ \overline{sg}((z)_{x+1}))$$

In this way we can give the following definition of recursive functional

DEFINITION 17.2. A functional $\Phi : \mathcal{F}(\mathbb{N}^k) \rightarrow \mathcal{F}(\mathbb{N}^h)$ is *recursive* if there exists $\varphi : \mathbb{N}^{h+1} \rightarrow \mathbb{N}$ computable such that, for every $f \in \mathcal{F}(\mathbb{N}^k), \vec{x} \in \mathbb{N}^h, y \in \mathbb{N}$

$$\Phi(f)(y) = y \quad \text{iff} \quad \exists \theta \subseteq f \text{ finite s.t. } \varphi(\tilde{\theta}, \vec{x}) = y$$

EXAMPLE 17.3. The functional

$$\text{fib} : \mathcal{F}(\mathbb{N}) \rightarrow \mathcal{F}(\mathbb{N})$$

$$\text{fib}(f)(x) = \begin{cases} 1 & x = 0 \lor x = 1 \\ f(x-2) + f(x-1) & x \geqslant 2 \end{cases}$$

is recursive, the function $\varphi : \mathbb{N}^2 \rightarrow \mathbb{N}$ can be

$$\varphi(z, x) = \begin{cases} 1 & x = 0 \lor x = 1 \\ \theta(x-2) + \theta(x-1) & x \geqslant 2 \land z = \tilde{\theta} \end{cases}$$

$$= \begin{cases} 1 & x = 0 \lor x = 1 \\ app(z, x-2) + app(z, x-1) & x > 2 \end{cases}$$

which is computable.

EXAMPLE 17.4. The functional associated to the Ackermann's function

$$\Psi_{ack} : \mathcal{F}(\mathbb{N}^2) \rightarrow \mathcal{F}(\mathbb{N}^2)$$

$$\begin{cases} \Psi_{ack}(f)(0, y) = y + 1 \\ \Psi_{ack}(f)(x+1, 0) = f(x, 1) \\ \Psi_{ack}(f)(x+1, y+1) = f(x, f(x+1, y)) \end{cases}$$

is clearly recursive.

THEOREM 17.5. *Let* $\Phi : \mathcal{F}(\mathbb{N}^k) \rightarrow \mathcal{F}(\mathbb{N}^h)$ *be a recursive functional and let* $f \in \mathcal{F}(\mathbb{N}^k)$ *be computable. Then* $\Phi(f) \in \mathcal{F}(\mathbb{N}^h)$ *is computable*

17.1. Myhill-Sheperdson theorems

Given a recursive functional Φ, by (17.5)

$$f \text{ computable} \rightsquigarrow \Phi(f) \text{ computable}$$

$$f = \varphi_e \rightsquigarrow \Phi(f) = \varphi_{e'}$$

so we can see a recursive functional as a function that transforms indices (programs) into indices (other programs), but with the property that the transformation depends on the indexed function and *not* on the index itself.

DEFINITION 17.6 (Extensional function). Let $h : \mathbb{N} \rightarrow \mathbb{N}$ a total function. It is *extensional* if

$$\forall e, e' \quad \varphi_e = \varphi_{e'} \rightarrow \varphi_{h(e)} = \varphi_{h(e')}$$

THEOREM 17.7 (Myhill-Shepherdson (I)). *If* $\Phi : \mathcal{F}(\mathbb{N}^k) \rightarrow \mathcal{F}(\mathbb{N}^h)$ *is a recursive functional then there exists a total computable function* $h_\Phi : \mathbb{N} \rightarrow \mathbb{N}$ *s.t.*

$$\forall e \in \mathbb{N} \quad \Phi(\varphi_e) = \varphi_{h_\Phi(e)}^{(k)}$$

Intuitively, the behaviour of the recursive functional on computable functions is captured by a total extensional function on the indices.

THEOREM 17.8 (Myhill-Shepherdson (II)). *If $h : \mathbb{N} \to \mathbb{N}$ is a total computable extensional function, then there is a unique recursive functional Φ_h sych that*

$$\forall e \in \mathbb{N} \quad \Phi_h(\varphi_e) = \varphi_{h(e)}$$

Note that a total computable extensional function, which only explains how to transform programs (hence computable functions) uniquely identifies a recursive functional, which instead is defined also on non-computable functions. The reason is roughly connected to the fact that all functions (even if they are not computable) can be approximated with arbitrary precision by computable ones (e.g., by finite subfunctions).

THEOREM 17.9 (First recursion theorem (Kleene)). *Let $\Phi : \mathcal{F}(\mathbb{N}^k) \to \mathcal{F}(\mathbb{N}^h)$ be a **recursive functional**. Then Φ has a **least fixed point** f_Φ which is **computable**, i.e.*

(1) $\Phi(f_\Phi) = f_\Phi$

(2) $\forall g \in \mathcal{F}(\mathbb{N}^k) \quad \Phi(g) = g \Rightarrow f_\Phi \subseteq g$

(3) f_Φ is computable

and we can see that $f_\Phi = \bigcup_n \Phi^n(\varnothing)$.

The theorem above implies the closure of the set of computable functions with respect to extremely general forms of recursion.

EXAMPLE 17.10 (Primitive recursion). Given $f : \mathbb{N}^h \to \mathbb{N}$ and $g : \mathbb{N}^{h+2} \to \mathbb{N}$, the function defined by primitive recursion is the least fixed point of $\Phi_r \in \mathcal{F}(\mathbb{N}^{h+1})$, defined by

$$\Phi_r(h)(\vec{x}, 0) = f(\vec{x})$$
$$\Phi_r(h)(\vec{x}, y + 1) = g(\vec{x}, y h(\vec{x}, y))$$

and if f, g are computable, then Φ_r is a recursive functional. The theorem assures that

- there exists a least fixed point;

- it is computable.

EXAMPLE 17.11 (Minimalisation). Given a function $f : \mathbb{N}^{k+1} \to \mathbb{N}$, we can see the minimization $\mu y \, . \, f(\vec{x}, y)$ as a fixed point. Let us consider, for a fixed f

$$\Phi_\mu \in \mathcal{F}(\mathbb{N}^{k+1})$$

$$\Phi_\mu(h)(\vec{x}, y) = \begin{cases} y & f(\vec{x}, y) = 0 \\ h(\vec{x}, y + 1) & f(\vec{x}, y) \downarrow \wedge f(\vec{x}, y) \neq 0 \\ \uparrow & \text{otherwise} \end{cases}$$

it is recursive and has a least fixed point:

$$f_{\Phi_\mu}(\vec{x}, y) = \mu z \geqslant y \, . \, f(\vec{x}, y)$$

EXAMPLE 17.12 (Ackermann's function). We saw that Φ_{ack} is recursive, therefore it has a computable least fixed point (the Ackermann function itself ψ). The fact that ψ is total, actually implies that such fixed point is the *only* fixed point.

OBSERVATION 17.13. In general the fixed point is not unique. Counter-example:

$$\Phi(f)(x) = \begin{cases} 0 & x = 0 \\ f(x+1) & \text{otherwise} \end{cases}$$

is recursive, and therefore has a minimum fixed point

$$f_\Phi(x) = \begin{cases} 0 & x = 0 \\ \uparrow & \text{otherwise} \end{cases}$$

but it has other fixed points, for example, for every $k \in \mathbb{N}$

$$f(x) = \begin{cases} 0 & x = 0 \\ k & x > 0 \end{cases}$$

CHAPTER 18

Second recursion theorem

Let $f : \mathbb{N} \to \mathbb{N}$ be total, computable and extensional i.e.

$$\forall e, e' \ \varphi_e = \varphi_{e'} \Rightarrow \varphi_{f(e)} = \varphi_{f(e')}$$

Then, by Theorem 17.8 (Myhill-Shephedson) there exists a unique recursive functional Φ such that

$$\forall e \in \mathbb{N} \quad \Phi(\varphi_e) = \varphi_{f(e)}$$

Since Φ is recursive, by the First Recursion Theorem (Theorem 17.9) it has a least fixed point $f_\Phi : \mathbb{N} \to \mathbb{N}$ computable. Therefore there is $e_0 \in \mathbb{N}$ such that

$$\varphi_{e_0} = f_\Phi = \Phi(f_\Phi) = \Phi(\varphi_{e_0}) = \varphi_{f(e_0)}$$

This means that if f is total computable and extensional, then there exists e_0 such that

$$\varphi_{e_0} = \varphi_{f(e_0)}$$

The second recursion theorem states that this holds also when f is not extensional.

THEOREM 18.1 (Second recursion theorem (Kleene)). *Let $f : \mathbb{N} \to \mathbb{N}$ a total computable function. Then there exists $e_0 \in \mathbb{N}$ such that*

$$\varphi_{e_0} = \varphi_{f(e_0)}$$

PROOF. Let $f : \mathbb{N} \to \mathbb{N}$ a total computable. Take

$$\begin{aligned}
g(x, y) &= \varphi_{f(\varphi_x(x))}(y) \\
&= \Psi_U(f(\varphi_x(x)), y) \\
&= \Psi_U(f(\Psi_U(x, x)), y)
\end{aligned}$$

it is computable. By the *smn* theorem there exists $s : \mathbb{N} \to \mathbb{N}$ total computable such that

$$\varphi_{s(x)}(y) = g(x, y) = \varphi_{f(\varphi_x(x))}(y) \quad \forall x, y$$

Since s is computable there exists $m \in \mathbb{N}$ such that

$$S = \varphi_m$$

hence

$$\varphi_{\varphi_m(x)}(y) = \varphi_{f(\varphi_x(x))}(y) \quad \forall x, y$$

For $x = m$

$$\varphi_{\varphi_m(m)}(y) = \varphi_{f(\varphi_m(m))}(y) \quad \forall y$$

We set $e_0 = \varphi_m(m) \downarrow$ and we replace in the previous equation

$$\varphi_{e_0}(y) = \varphi_{f(e_0)}(y) \quad \forall y$$

i.e.

$$\varphi_{e_0} = \varphi_{h(e_0)}$$

<div style="text-align: right">□</div>

This theorem can therefore be interpreted in the following manner *given any effective procedure to transform programs, there is always at a program such that when modified by the procedure it does exactly what it did before* or *it is impossible to write a program that change the extensional behaviour of of all programs.*

The proof of the theorem can appear mysterious, but after a closer inspection, it clearly appears to be a simple diagonalization. Nevertheless, the result of this theorem is extremely deep; in this way, many theorems we've seen up until now are just corollaries.

COROLLARY 18.2 (Rice's theorem). *Let $\varnothing \neq A \subsetneq \mathbb{N}$ saturated, then A is not recursive.*

PROOF. Let $\varnothing \neq A \subset \mathbb{N}$ saturated. Take $e_1 \in A$ and $e_0 \notin A$ and assume by contradiction that A is recursive. Define $f : \mathbb{N} \to \mathbb{N}$

$$f(x) = \begin{cases} e_0 & x \in A \\ e_1 & x \notin A \end{cases}$$
$$= e_0 \cdot \chi_A(x) + e_1 \cdot \chi_{\bar{A}}(x)$$

Since A is recursive then also $\bar{A}$ is recursive, thus χ_A and $\chi_{\bar{A}}$ are computable. Thus, f is computable and total, then by the Second Recursion Theorem (18.1) there exists $e \in \mathbb{N}$ such that $\varphi_e = \varphi_{f(e)}$; there are two possibilities

- if $e \in A$, then $f(e) = e_0 \notin A$ and since A saturated, $\varphi_e \neq \varphi_{e_0} = \varphi_{f(e)}$

- if $e \notin A$, then $f(e) = e_1 \in A$ and since A saturated, $\varphi_e \neq \varphi_{e_1} = \varphi_{f(e)}$

that is absurd, so A cannot be recursive.   □

COROLLARY 18.3. *The halting set $K = \{x \mid \varphi_x(x) \downarrow\}$ is not recursive.*

PROOF. Let $k = \{x \mid x \in W_x\}$ recursive for the sake of the argument. and let e_0, e_1 be indexes s.t. $\varphi_{e_0} = \varnothing$ and $\varphi_{e_1} = \lambda x \, . \, x$.

Define $f : \mathbb{N} \to \mathbb{N}$

$$f(x) = \begin{cases} e_0 & x \in K \\ e_0 & x \notin K \end{cases}$$
$$= e_0 \cdot \chi_K(x) + e_1 \cdot \chi_{\bar{K}}(x)$$

If K were recursive, then χ_K and $\chi_{\bar{K}}$ would be computable, thus f would be both computable and total, then by (18.1), there would be $e \in \mathbb{N}$ such that $\varphi_e = \varphi_{f(e)}$, but

- if $e \in K$, then $f(e) = e_0$, so $\varphi_e(e) \downarrow \neq \varphi_{f(e)}(e) = \varphi_{e_0}(e) \uparrow$

- if $e \in \bar{K}$, then $f(e) = e_1$, so $\varphi_e(e) \uparrow \neq \varphi_{f(e)}(e) = \varphi_{e_1}(e) = e \downarrow$

which is absurd, so K cannot be recursive. $\qquad\square$

COROLLARY 18.4. $K = \{x \mid \varphi_x(x) \downarrow\}$ *is not saturated.*

PROOF. First observe that there is n_0 s.t. $\varphi_{n_0} = \{(n_0, n_0)\}$. In fact, define

$$g(n, x) = \begin{cases} 0 & \text{if } x = n \\ \uparrow & \text{otherwise} \end{cases} = \mu z.\ |x - n|$$

Since g is computable there is $s : \mathbb{N} \to \mathbb{N}$ total computable such that $\varphi_{s(n)}(x) = g(x, y)$. By the Second Recursion Theorem there is n_0 such that $\varphi_{n_0} = \varphi_{s(n_0)}$. Therefore

$$\varphi_{n_0}(x) = \varphi_{s(n_0)}(x) = g(n_0, x) = \begin{cases} 0 & \text{if } x = n_0 \\ \uparrow & \text{otherwise} \end{cases}$$

Observe that $n_0 \in K$. Moreover we know that there are infinitely many indices for the same function. Thus let $n \neq n_0$ s.t. $\varphi_n = \varphi_{n_0}$. Then

$$\varphi_n(n) = \varphi_{n_0}(n) \uparrow$$

Hence $n \notin K$ and thus K is not saturated. $\qquad\square$

Bibliography

[Fri58]   Richard M. Friedberg. *Three Theorems on Recursive Enumeration. I. Decomposition. II. Maximal Set. III. Enumeration Without Duplication.* Vol. 23. 3. Journal of Symbolic Logic, 1958, pp. 309–316.

[Dav11]   Martin Davis. *The Universal Computer: The Road from Leibniz to Turing.* 1st. USA: A. K. Peters, Ltd., 2011. ISBN: 1466505192.