Lectures Notes : Algorithmic Information Theory

Itisan Halias
itha.mail0@gmail.com

September 2025

Contents

Chapter 1

Algorithmic "Turing" Computability

In this chapter $\mathcal{A}$ and $\mathcal{A}'$ are arbitrary alphabets. We also set $\mathbb{B} = \{0, 1\}$. We further assume $\mathbb{B} \subset \mathcal{A}$. We set $\mathbb{B} = \{0; 1\}^*$.

1.1 Turing Machine

1.1.1 Definition of a multi-tape Turing machine

A multi-tape Turing machine allows the use of several work tapes in parallel, instead of a single one. Each tape possesses its own read/write head which can move independently of the others.

The operation is similar to a classical Turing machine, but at each step:

- The machine simultaneously reads the symbols under each read head
- Depending on these symbols and the current state, the machine:
  - Changes state
  - Writes new symbols on one or several tapes
  - Moves each head independently (left, right, or stays in place)

This variant facilitates the design and description of certain algorithms by allowing intermediate data to be stored and manipulated on separate tapes. It is important to note that a multi-tape machine is not more powerful than a single-tape machine: any computation realizable with k tapes can be simulated by a single-tape machine (We will show this result in a section dedicated to reduction later in the course).

Definition: k-tape Turing Machine

A k-tape Turing machine is a 7-tuple

$$M = (Q, \mathcal{A}, \Gamma, \#, \delta, q_0, q_a)$$

where:

1. Q is an arbitrary finite set whose elements are called states.
2. $\mathcal{A}$ is an arbitrary alphabet called the input alphabet.
3. Γ is a work alphabet (or tape alphabet), which includes the blank symbol $\#$ (or empty symbol), that is $\# \in \Gamma$. Furthermore $\mathcal{A} \subset \Gamma$.

4. $\#$ is the blank symbol, used to represent the empty cells of the tapes.
5. $\delta : (Q/\{q_a\}) \times \Gamma^k \to Q \times \Gamma^k \times \{\leftarrow, |, \rightarrow\}^k$ is the transition function, where Γ^k represents the symbols on the k tapes and each direction of movement is specified for each tape.
6. $q_0 \in Q$ is the initial state.
7. $q_a \in Q$ is the accepting state, where the machine halts and accepts the input.

We then set $\mathcal{T}$ as the set of Turing machines.

1.1.2 Configurations of a Turing machine

We will formally introduce the notion of configuration of a k-tape Turing machine which describes the current state of the machine, the content of each tape, and the position of the read/write heads.

Definition: Configuration of a Turing machine

Let there be a k-tape Turing machine

$$M = (Q, \mathcal{A}, \Gamma, \#, \delta, q_0, q_a).$$

We call a configuration of M an element:

$$C = (q, (c_j^1, \dots, c_j^k)_{j\in\mathbb{Z}}, (p_1, \dots, p_k)) \in Q \times (\Gamma^k)^{\mathbb{Z}} \times \mathbb{Z}^k.$$

In other words, C represents the state q of the machine, where the content of the i-th tape is described by the sequence $(c_j^{(i)})_{j\in\mathbb{Z}}$ and the i-th read head is positioned at location p_i of this sequence. Furthermore, we call:

- the sequence $(c_j^1)_{j\in\mathbb{Z}}$ the input tape,
- the sequences $(c_j^i)_{j\in\mathbb{Z}}$ for $i \in \{2, \dots, k-1\}$ the work tapes,
- the sequence $(c_j^k)_{j\in\mathbb{Z}}$ the output tape.

We can then informally describe the operation of a Turing machine: it functions by passing from one configuration to the next, determined by its transition function δ. This can be formally specified according to the definition opposite.

Definition: Successive configurations of a Turing machine

Let there be a k-tape Turing machine

$$M = (Q, \mathcal{A}, \Gamma, \#, \delta, q_0, q_a)$$

and two arbitrary configurations C, C' of M

$$\begin{cases} C = (q, (r_j^{(1)}, \dots, r_j^{(k)})_{j\in\mathbb{Z}}, (p_1, \dots, p_k)) \\ C' = (q', (r_j'^{(1)}, \dots, r_j'^{(k)})_{j\in\mathbb{Z}}, (p_1', \dots, p_k')) \end{cases}$$

We then say that C' succeeds C and we denote $C \vdash_M C'$ if:

- $\delta(q, (r_{p_1}^{(1)}, \dots, r_{p_k}^{(k)})) = (q', (a_1, \dots, a_k), (\theta_1, \dots, \theta_k))$
- $\forall i \in [\![1; k]\!], \begin{cases} \text{if } \theta_i = \leftarrow & \text{then } p'_i = p_i - 1 \\ \text{if } \theta_i = | & \text{then } p'_i = p_i \\ \text{if } \theta_i = \rightarrow & \text{then } p'_i = p_i + 1 \end{cases}$
- $\forall i \in [\![1; k]\!] : r_{p_i}^{\prime(i)} = a_i$ and $\forall j \in \mathbb{Z}$ such that $j \neq p_i$ we have $r_j^{\prime(i)} = r_j^{(i)}$

Furthermore, if there exist configurations $C_1, \dots, C_n$ such that $\forall i \in [\![1; n-1]\!], C_i \vdash_M C_{i+1}$ we will then denote

$$C_1 \vdash_M^* C_n$$

1.1.3 Accepted language, looping, and output of a Turing machine

In its intuitive definition, a program allows checking certain properties on its inputs and providing a result which is the fruit of a computation as output. The definitions of accepted/looping languages and of the outputs of a Turing machine allow capturing this notion.

Definition: Accepted or looping words and output

Let there be a k-tape Turing machine, denoted

$$M = (Q, \mathcal{A}, \Gamma, \#, \delta, q_0, q_a)$$

and a word $w \in \mathcal{A}^*$, which we denote in the form $w = w_0 w_1 \dots w_n$. We define the initial configuration as follows:

$$C_0 = \left(q_0, (c_j^{(1)}, \dots, c_j^{(k)})_{j \in \mathbb{Z}}, (0, \dots, 0)\right),$$

where for each tape i and each position j:

- For the input tape ($i = 1$):

$$c_j^{(1)} = \begin{cases} w_j & \text{if } 0 \leq j \leq n \\ \# & \text{otherwise} \end{cases}$$

- For the other tapes ($i \in \{2, \dots, k\}$):

$$\forall j \in \mathbb{Z}, \quad c_j^{(i)} = \#$$

We then define:

- If there exists $(\alpha, p) \in (\Gamma^k)^{\mathbb{Z}} \times \mathcal{Z}^k$ such that $C_0 \vdash_M^* (q_a, \alpha, p)$ then w is said to be accepted by M, denoted $\downarrow M(w)$.

  Furthermore, if M accepts w, we define its output as the value of its output tape

$$\downarrow M(w) := (c_j^{(k)})_{j \in [\![1; l]\!]}$$

  with $(c_j^{(k)})_{j \in [\![l, \dots, l']\!]}$ being the output tape from which the symbols # at the beginning and end are trimmed (see remark).

- Otherwise w is said to be looping for M, denoted $\circlearrowleft M(w)$.

Remark. When we say $(c_j^{(k)})_{j\in[\![l,\dots,l']\!]}$ [is] the output tape from which the symbols # at the beginning and end are trimmed, it must be understood [as]

$$\begin{cases} l &= \min\{i \in \mathbb{Z} \mid c_i^{(k)} \neq \#\} \\ l' &= \max\{i \in \mathbb{Z} \mid c_i^{(k)} \neq \#\} \end{cases}$$

◇

From the definition of a word accepted or looping for a Turing machine, we can naturally introduce the notion of accepted and looping language. The accepted language designates the words for which the machine eventually accepts the input, whereas a looping language is the one for which the machine never halts.

Definition: Accepted and looping languages

Let M be a k-tape Turing machine, we then set respectively the language accepted and refused by M as

$$\begin{cases} L_{\downarrow}(M) := \{x \in \mathcal{A}^* \mid \downarrow M(x)\} \\ L_{\circlearrowleft}(M) := \{x \in \mathcal{A}^* \mid x \notin L_{\downarrow}(M)\} \end{cases}$$

1.1.4 Output of a Turing machine after t transitions

Until now, we have not introduced notation to represent the output of a Turing machine during its operation. This is why we introduce the following notation which indicates the value of the output after a number t of transitions. We will use this notation in particular to define monotone Turing machines.

Definition: Output of M after t transitions

Let there be a k-tape Turing machine, denoted

$$M = (Q, \mathcal{A}, \Gamma, \#, \delta, q_0, q_a).$$

Let us consider the initial configuration C_0 with w on the input tape. If there exists a sequence of configurations $C_0, \dots, C_t$ such that

$$\underbrace{C_0 \vdash_M \cdots \vdash_M C_t}_{\text{t transitions}},$$

and if $(c_j^{(k)})_{j\in\{l,\dots,l'\}}$ represents the content of the output tape of C_t from which the symbols # at the beginning and end are trimmed, that is formally

$$\begin{cases} l &= \min\{i \in \mathbb{Z} \mid c_i^{(k)} \neq \#\} \\ l' &= \max\{i \in \mathbb{Z} \mid c_i^{(k)} \neq \#\} \end{cases}$$

then we define

$$M(w \mid t) \underset{def.}{=} c_1 c_2 \dots c_l = (c_j^{(k)})_{j\in\{1,\dots,l\}}$$

as the output after t transitions for the input w.

1.1.5 History of a Turing machine

We wish to introduce a definition to describe the successive variations of the output of a machine. This notion can be illustrated by a concrete example. Consider a Turing machine whose successive outputs are the following: 0, 01, 011, 011, 01, 011, 011, 011, 010, 010, 000, 000, 000. The history of this Turing machine is then summarized as: 0, 01, 011, 01, 011, 010, 000, which means that we have eliminated the duplicates from the output.

Definition: History of a Turing machine

Let M be an arbitrary Turing machine. Let us set for all $t \in \mathbb{N}$ the output after t transitions, that is $o_t = M(p \mid t)$. We then call output history of a Turing machine M for an input $p \in L_{\downarrow}(M)$ the sequence $(h_k)_{k\in\{0,1,\dots,l\}}$ with $l \in \mathcal{N} \cup \{\infty\}$ such that

$$\begin{cases} h_0 = o_0 \\ n_0 = 0 \end{cases}$$

and defined by the recurrence,

$$\forall k > 0, \quad \begin{cases} n_k = \min\{n > n_{k-1} \mid o_n \neq h_{k-1}\} \\ h_k = o_{n_k} \quad \text{if } n_k \neq \infty \end{cases}$$

Note that we use the convention $\min\{\emptyset\} = \infty$.

1.1.6 Equivalence between Turing machines for an input

Equivalence between Turing machines, on an input, means that the behavior of two machines on specific inputs is the same. We will introduce a notation to designate the fact that two Turing machines are equivalent on specific inputs.

Definition: Equivalence between Turing machines for inputs

We say that M on input p is equivalent to M' on input q if one of the two conditions is verified:

1. $\downarrow M(p) \wedge \downarrow M'(q)$ with equality of outputs i.e. $M(p) = M'(q)$
2. $\circlearrowleft M(p) \wedge \circlearrowleft M'(q)$

We then denote

$$M(p) \equiv M'(q)$$

Furthermore if $M(p)$ and $M'(q)$ have the same output history then we denote

$$M(p) \underset{h}{\equiv} M'(q)$$

Similarly if for a given input, two Turing machines have the same behavior we will say that they are equivalent.

Definition: Equivalence between Turing machines

We say that M is equivalent to M' if for all $p \in \mathbb{B}^*, M(p) \equiv M'(p)$. We then denote

$$M' \equiv M$$

Similarly if for all $p \in \mathbb{B}^*$, $M(p) \underset{h}{\equiv} M'(p)$. We then denote

$$M' \underset{h}{\equiv} M$$

1.1.7 Computable functions

We speak of partial functions when the domain of definition can be a strict subset of the definition domain of the function.

Definition: Partially computable function

Let $\mathcal{I}$ and $\mathcal{O}$ be two arbitrary languages in $\mathcal{A}^*$. Let there be a function $f : \mathcal{I} \to \mathcal{O}$ and a set $dom(f) \subset \mathcal{I}$. The function f is said to be partially computable with domain $dom(f)$ if there exists a Turing machine M such that:

$$\begin{cases} \downarrow M(x) \text{ and } f(x) = M(x) & \text{if } x \in dom(f) \\ \circlearrowleft M(x) & \text{if } x \in \mathcal{I} \setminus dom(f) \end{cases}$$

Remark. Two important points to consider:

- Note that for x in $\mathcal{A}^* \setminus \mathcal{I}$ the Turing machine can very well accept x, that is $M(x) \downarrow$. The constraint applies only to the x in $\mathcal{I}$.
- In practice we will use this definition only in the case of the binary alphabet $\mathbb{B} = \{0, 1\}$. That is $\mathcal{I}$ and $\mathcal{O}$ [are] two arbitrary languages in $\mathbb{B}^*$.

◇

Let us first establish a convention which is often present in the literature without necessarily being specified.

Convention:

In the literature the domain of a partially computable function is not always explicit. In our course the domain will always be a subset of $\mathbb{B}^*$. That is to say that "f is partially computable" without specifying the domain must always be read as "$f : \mathbb{B}^* \mapsto \mathbb{B}^*$ is partially computable with domain $dom(f) \subset \mathbb{B}^*$".

Let us now define the notion of totally computable function which can be seen as an additional constraint on the domain of a partially computable function.

Definition: Totally computable function

Let $\mathcal{I}$ and $\mathcal{O}$ be two arbitrary languages in $\mathcal{A}^*$. A function $f : \mathcal{I} \to \mathcal{O}$ is said to be totally computable if there exists a Turing machine M such that:

$$\downarrow M(x) \text{ and } f(x) = M(x) \quad \text{if } x \in \mathcal{I}$$

Remark. This definition is equivalent to f [being] partially computable with domain $dom(f) = \mathcal{I}$. We can make the same remark as previously, the constraint applies only to the domain. Thus if $x \notin dom(f)$ we can very well have $\downarrow M(x)$ or $\circlearrowleft M(x)$. ◇

1.2 Church-Turing Thesis and high-level language

1.2.1 Examples

A three-tape machine that tests if an integer written in binary on the input tape is even functions as follows: it moves the read head to the right until the end of the word and checks if the last digit is 0. To locate the end, it advances by one cell to find the first blank symbol. The machine is described by:

- Input alphabet: $\mathcal{A} = \{0, 1\}$;
- Work alphabet: $\Gamma = \{0, 1, B\}$;
- States: $Q = \{q_0, q_1, q_a\}$;
- Transition function:

$$\begin{aligned}
&\delta(q_0, (u, B)) = (q_0, (B, B), (D, S, S)) \quad \text{for all } u \in \{0, 1\},\\
&\delta(q_0, (B, B)) = (q_1, (B, B), (G, S, S)),\\
&\delta(q_1, (0, B)) = (q_a, (B, B), (S, S, S)),\\
&\delta(q_1, (1, B)) = ((B, B), (S, S, S)),\\
&\delta(q, (u, v)) = ((B, B), (S, S, S)) \quad \text{for the others } (q, (u, v)).
\end{aligned}$$

Note that the work and output tapes are not used here.

1.2.2 High-level language

Thesis: Church-Turing

> "Any problem that can be solved by an algorithm can be solved by a Turing machine."

By algorithm, we mean here a high-level language such as C, C++ or others. In other words, variable storage, basic operations $(+, -, \times)$, comparisons $>, <, \leq, \geq$, conditional instructions ("if", "else"), loops ("while") and function calls can be simulated by a Turing machine.

Proof idea.

- Variable storage: The integer variables x_i are represented by a given encoding in binary and separated by † on a tape: $x_1 \dagger x_2 \dagger \cdots \dagger x_n$
- Basic operations: The operations of assignment, addition and multiplication are simulated by instructions on the Turing machine:
  - Assignment $\leftarrow$: Writing of the value at the corresponding location on the tape.
  - Addition $+$, Multiplication $\times$ and Subtraction $-$: Realized by algorithms of incrementation and repeated additions.
- Condition Operations: A condition "$=$" is performed by comparing bit by bit two values x_i and x_j. The conditions "$>, <, \leq, \geq$" are performed by comparing bit by bit starting from the most significant bit.

- Conditional instructions: The instructions "if", "else" are managed by states, allowing the machine to change execution path according to the result of the instruction's condition.
- Loops: The loop "while" is simulated by a state, allowing the machine to repeat instructions as long as a condition is true. Once the condition is no longer met the Turing machine returns to the previous state.
- Function call: Consider two Turing machines, M and M', which dispose respectively of k and k' tapes such that $(k' \geq 2k)$. When a Turing machine M calls M' with input w, it simulates $M'(w)$ in the following manner: the states of M before the call are saved using a specific encoding on a dedicated backup tape. Then, M simulates the operation of M' on k' additional tapes, in addition to its k tapes. Once the execution of M' is finished, M restores its previous state thanks to the information stored on the backup tape.

■

1.2.3 Encoding of an integer (binary representation)

Definition-Property: Encoding of an integer (binary representation)

Let $n \in \mathbb{N}^*$, there exists a unique word $b = b_l b_{l-1} \dots b_0$ in $\mathbb{B}^*$ with $b_l = 1$ and $l = \lfloor \log(n) \rfloor + 1$ such that

$$n = \sum_{i=0}^{l} b_i 2^i$$

We say that b is the binary representation associated with n. Furthermore, we associate to $0 \in \mathbb{N}$ the binary representation $0 \in \mathbb{B}$. We then define the bijection

$$bin : \mathbb{N} \mapsto \mathbb{B}^*$$

which associates its binary representation to an integer.

Proof. Let us show the uniqueness of the binary representation. Let $b = b_l b_{l-1}$ and $b' = b'_{l'} b'_{l'-1} \dots b'_0$ be two binary representations of $n \in \mathbb{N}$. If $n = 0$ then immediately $b = b' = 0$, so we will assume in the following that $n > 0$. We immediately have $l = l' = \lfloor \log(n) \rfloor + 1$. Assume $b \neq b'$ and let us set the largest $1 \leq m \leq l$ such that $b_m \neq b'_m$. Without loss of generality, assume $b'_m = 1$ and $b_m = 0$. We then obtain

$$\underbrace{\sum_{i=0}^{l} (b'_i - b_i) 2^i}_{=n-n=0} = \sum_{i=0}^{m} (b'_i - b_i) 2^i = 2^m + \sum_{i=0}^{m-1} (b'_i - b_i) 2^i \geq 2^m + \frac{2^{l+1} - 1}{2 - 1} > 0$$

which is absurd. We therefore necessarily have $\forall 1 \leq i \leq l, b_i = b'_i$ which demonstrates the uniqueness. ■

It would be desirable to have, as for integers, an addition operation and a comparison relation for numbers in binary representation. To this end, we introduce the set $\mathcal{N}$ with which we associate an addition operation $+$ and a comparison operation $\leq$ as follows.

Definition: Binary integers

We define the set

$$\mathcal{N} \underset{def.}{=} \{bin(n) \mid n \in \mathbb{N}\}$$

We then define an order relation $\dot{\leq}$ and an internal composition law $\dot{+}$ on the set $\mathcal{N}$ such that for all x, y in $\mathcal{N}$,

$$\begin{cases} x \dot{\leq} y \underset{def.}{\iff} bin^{-1}(x) \leq bin^{-1}(y) \\ x \dot{\times} y \underset{def.}{\iff} bin^{-1}(x) \times bin^{-1}(y) \\ x \dot{+} y \underset{def.}{=} bin^{-1}(x) + bin^{-1}(y) \end{cases}$$

In the following, we will write $+$, $\leq$ and $\times$ for $\dot{+}$, $\dot{\leq}$ and $\dot{\times}$.

Property: $\mathcal{N}$ is "recognizable"

There exists a partially computable function $Assert_{\mathcal{N}}$ with domain $\mathcal{N}$.

Proof. It is trivial. It suffices to set an algorithm that checks if its input does not start with a 0. If an input satisfies such a condition then necessarily the input is in $\mathcal{N}$. Thus there exists a partially computable function $Assert_{\mathcal{N}}$ with domain $\mathcal{N}$. ■

Until now, in order to measure the size of a word, we had the function $|\cdot|$ returning the size of a word in $\mathbb{N}$. It would be useful for us to have an analogue that returns the size of a word but in binary representation. We thus introduce the function ℓ returning the size of a word in binary representation.

Definition: Binary size of a word

We define the function $\ell : \mathbb{B}^* \mapsto \mathcal{N}$ such that

$$\forall x \in \mathbb{B}^*, \quad \ell(x) = bin(|x|)$$

In the following, in order not to overload the writing, we will allow ourselves the following abuse of notation.

Convention:

For $x \in \mathbb{B}^*$, we will sometimes denote, in an abusive manner, $1^{|x|} = 1^{\ell(x)}$. This notation aims to avoid overloading the writing.

1.2.4 Encoding of relative integers

Definition: Encodings of relative integers

We define the set

$$\mathcal{Z} \underset{def.}{=} \{0bin(n) \mid n \in \mathbb{N}\}$$

We then define an order relation $\dot{\leq}$ and an internal composition law $\dot{+}$ on the set $\mathcal{N}$

such that for all x, y in $\mathcal{Z}$,

$$\begin{cases} x \dot{\leq} y \underset{def.}{\iff} bin^{-1}(x) \leq bin^{-1}(y) \\ x \dot{\times} y \underset{def.}{\iff} bin^{-1}(x) \times bin^{-1}(y) \\ x \dot{+} y \underset{def.}{=} bin^{-1}(x) + bin^{-1}(y) \end{cases}$$

In the following, we will write $+$, $\leq$ and $\times$ for $\dot{+}$, $\dot{\leq}$ and $\dot{\times}$.

Property: $\mathcal{Z}$ is "recognizable"

There exists a partially computable function $Assert_{\mathcal{Z}}$ with domain $\mathcal{Z}$.

Proof. It is trivial. It suffices to set an algorithm that checks if its input does not start with a 01. If an input satisfies such a condition then necessarily the input is in $\mathcal{Z}$. Thus there exists a partially computable function $Assert_{\mathcal{Z}}$ with domain $\mathcal{Z}$. ■

1.2.5 Encoding of rationals

We introduce here the encoding of rationals which will allow us in the following section

Definition: Encoding of rationals

We call encoding of rationals in binary form the set

$$\mathcal{Q} := \mathcal{N} \times (\mathcal{N} \setminus \{0\}).$$

To $\mathcal{Q}$, we associate two internal composition laws, denoted $\oplus$ and $\otimes$ and an order relation $(\mathcal{Q}, \leqslant)$ which we define [a]

$$\forall n_1, n_2 \in \mathcal{N}, \quad \forall d_1, d_2 \in \mathcal{N}\setminus\{0\}, \quad \begin{cases} \langle n_1, d_1 \rangle \oplus \langle n_2, d_2 \rangle = \langle n_1 \dot{\times} d_2 + n_2 \dot{\times} d_1, d_1 \dot{\times} d_2 \rangle \\ \langle n_1, d_1 \rangle \otimes \langle n_2, d_2 \rangle = \langle n_1 \dot{\times} n_2, d_1 \dot{\times} d_2 \rangle \\ \langle n_1, d_1 \rangle \leqslant \langle n_2, d_2 \rangle \underset{def.}{\iff} n_1 \times d_2 \dot{\leq} n_2 \times d_1. \end{cases}$$

In the following, we will denote the operations $\oplus$ and $\otimes$ by $+$ and $\times$, respectively. Similarly, we will use $\leq$ to designate the order relation $\leqslant$. Furthermore there clearly exists a bijection

$$Qbin : \begin{cases} \frac{n}{d} & \mapsto \langle bin(n), bin(d) \rangle \\ \mathbb{Q} & \mapsto \mathcal{Q} \end{cases}$$

[a] We denote here $\dot{+}$ and $\dot{\times}$ for the operations on binaries, and $\dot{\leq}$ for comparisons.

Convention:

Let $(q_n)_{n\in\mathbb{N}}$ be a sequence in $\mathcal{Q}$ such that $\lim_{n\to\infty} Qbin(q_n) = l \in \mathbb{R}$, we will then denote for the sake of conciseness $\lim_{n\to\infty} q_n \underset{not.}{=} l \in \overline{\mathbb{R}}$.

1.2.6 Lexicographical order of $\mathbb{B}^*$

We will now define an order that we call lexicographical order. Lexicographical order allows ordering words based on alphabetical order (here the binary alphabet $\mathbb{B}$). In this order, elements are compared character by character, following the order of letters in the alphabet.

Definition: Lexicographical order

We set the ordered set $(\mathbb{B}^*, \preccurlyeq)$ such that for all $b = b_1 b_2 \dots b_l$ and $b' = b'_1 b'_2 \dots b_{l'}$ in $\mathbb{B}^*$:

$$b \preccurlyeq b' \underset{def.}{\iff} (|b| < |b'|) \text{ or } (\exists k, \forall i \in \{1, \dots k\}, b_i = b'_i \text{ and } b_k < b'_k)$$

Remark. Let us enumerate $\mathbb{B}^*$ in lexicographical order: 0 , 1 , 00 , 01 , 10 , 11 , 000 , 001 , 010 , 011 , 100 , 101 , 110 ... ◇

It is possible to present a program which allows enumerating the set $\mathbb{B}^*$ in lexicographical order. We will see later in the course that the computable function $\pi_{lex.}$ is what is called an enumeration of $\mathbb{B}^*$.

Property: Enumeration in lexicographical order

There exists a totally computable function $\pi : \mathcal{N} \mapsto \mathbb{B}^*$ such that

$$\forall i, j \in \mathcal{N}, \quad i < j \implies \pi(i) \preccurlyeq \pi(j)$$

Proof. Let us set the following totally computable function lex:

For an input i in $\mathcal{N}$.
Initialize $k \leftarrow 0$ and $n \leftarrow 1$ and $s \leftarrow 0$
Do for $j = 1, \dots, i$:

- If s is composed only of 1s then
  - Assign $n \leftarrow n + 1$ and $k \leftarrow 0$
  - Assign $s \leftarrow 00 \dots 0$ with n times 0.
- Otherwise
  - $k \leftarrow k + 1$
  - $s \leftarrow 0^{n-\ell(k)}k$

Write s to output and accept

We justify firstly that lex is totally computable. Indeed for any input i in $\mathcal{N}$ the procedure terminates because all operations (finite loop, addition, concatenation, ...) are executed in a finite number of transitions.

For a call $lex(i)$, let us set the following loop invariant for all $0 \leq j \leq i$:

$I(j)$: "At the beginning of the loop at iteration j, the variable $s = 0^{n-\ell(k)}k$ which is equal to $lex(j)$."

Base ($j = 0$): By the initialization $s = \varepsilon$ which is the smallest word in the lexicographical order, that is $lex(0)$. The loop will not be executed because $i < 1$ and we will have writing of $s = lex(0)$. We have $I(0)$

Maintenance (j): Assume that the invariant $I(j)$ is true for a $j < i$ and let us show $I(j+1)$. There will necessarily be a $j+1$-th iteration because exactly i iterations take place. Two cases are then possible at the j-th iteration:

- If at the beginning of the iteration $s = lex(j) = 0^{n-\ell(k)}k$ is composed only of 1s then $n = \ell(k)$ and necessarily $s = k = 11\ldots1$ with n times 1. The next term in the lexicographical order $lex(j+1)$ is therefore $11\ldots1$ composed of $n+1$ times 1. Now this corresponds exactly to the update of s, so at the end of the iteration we will have $s = lex(j+1)$. Furthermore we will have the assignments $n \leftarrow n+1$ and $k \leftarrow 0$ therefore $s = 0^{n-\ell(k)}$. $I(j+1)$ is maintained.
- If at the beginning of the iteration $s = lex(j) = 0^{n-\ell(k)}k$ is not composed only of 1s then the word following s in the lexicographical order is $0^{n-\ell(k+1)}(k+1)$. Now this is exactly the update performed on s. We thus indeed have that s updated is $lex(j+1)$. Furthermore $k \leftarrow k+1$, so we indeed have at the end of the iteration $s = 0^{n-\ell(k+1)}(k+1)$. $I(j+1)$ is maintained.

Termination: For the call $lex(i)$, the loop is executed exactly i times so s equals $lex(i)$ and will be written to output. We thus have termination. ■

1.2.7 Dovetail execution

Before proceeding, it is necessary to introduce a particular type of execution of a Turing machine. Suppose we wish to determine on which word a Turing machine M halts for a sequence of words $w_1, w_2, \ldots$. A naive approach would consist of executing M on w_1, then waiting for the end of this execution before moving to w_2, and so on. However, this method presents a major drawback: given that a Turing machine can potentially enter an infinite loop on a word, we risk waiting indefinitely for certain words. Dovetail execution offers a solution to this problem by allowing several words to be processed simultaneously.

Principle: Dovetail call of M

Dovetail execution consists of simulating several Turing machines in parallel on a sequence of words.

- Initialization: We consider an enumeration of possible input words, $w_1, w_2, w_3, \ldots$, for example ordered lexicographically.
- Parallel Simulation: Dovetail execution consists of simulating n instances of the Turing machine M on $w_1, w_2, w_3, \ldots w_n$ with n increasing at each step. At each step of the simulation, we interleave the transitions of the different instances on a single tape with † as separator:

$$\dagger M(w_1 \mid k) \dagger M(w_2 \mid k-1) \dagger \cdots \dagger M(w_{k-1} \mid 2) \dagger M(w_k \mid 1)\dagger$$

  For example, for $n+1$ simulations, we will have:

  - At step 1: $\dagger M(w_1 \mid 1)\dagger$
  - At step 2: $\dagger M(w_1 \mid 2) \dagger M(w_2 \mid 1)\dagger$.
  - At step 3: $\dagger M(w_1 \mid 3) \dagger M(w_2 \mid 2) \dagger M(w_3, 1)\dagger$.
  - At step 4: $\dagger M(w_1 \mid 4) \dagger M(w_2 \mid 3) \dagger M(w_3 \mid 2) \dagger M(w_4 \mid 1)\dagger$
  - ...
  - At step n: $\dagger M(w_1 \mid n) \dagger M(w_2 \mid n-1) \dagger \cdots \dagger M(w_{n-1} \mid 2) \dagger M(w_n \mid 1)\dagger$

- Recovery of Results at each step: At the end of each simulation step, it is possible to recover the results for each instance to examine if they are accepted, compare them...

1.3 Reduction

One might be tempted to believe that a multi-tape Turing machine, using a non-binary alphabet, is more expressive than a single-tape Turing machine with a binary alphabet. However, this idea is erroneous. Indeed, a single-tape Turing machine, even with a binary alphabet, can simulate any multi-tape Turing machine or [one] using a more complex alphabet. Although this simulation may be less efficient in terms of time or space.

1.3.1 Alphabet reduction

The notion of translation morphism is quite explicit: it transforms a word of one alphabet into its corresponding word of another alphabet.

Definition: Translation morphism

A morphism $\varphi : \mathcal{A}^* \mapsto \mathcal{A}'^*$ is called a translation [morphism] if:

$$\forall v, w \in \mathcal{A}^*, \quad \varphi(vw) = \varphi(v)\varphi(w)$$

In other words φ encodes each letter of $\mathcal{A}$ by a word over $\mathcal{A}'$.

One would be tempted to say to oneself that a Turing machine having a rich alphabet would be more powerful in what it allows to compute. However, the following theorem shows that this is false. Thus the following result establishes that it is possible to reduce any Turing machine to an equivalent machine that operates solely on a binary alphabet. This reduction is realized using a bijective translation morphism, which associates to each symbol of the original alphabet a binary representation, while preserving the behavior of the machine.

Theorem: Reduction to the binary alphabet

Let there be two Turing machines M and M' with respective work alphabets $\mathcal{A}$ and $\mathbb{B}$. Then, there exists $\varphi : \mathcal{A}^* \mapsto \mathbb{B}^*$ a bijective translation morphism such that

$$\forall w \in \mathcal{A}, \quad M'(\varphi(w)) \equiv M(w)$$

Proof idea. We assume the hypotheses of the theorem. Let us set $\mathcal{A} = \{s_1, \ldots, s_n\}$. We can then take the morphism φ which associates to s_i the binary notation of i on $\lceil log_2(n) \rceil$ bits. We thus have φ which is bijective by uniqueness of the binary representation.

The idea consists for the Turing machine M' to consider the block of cells encoding $\varphi(s_i)$ as a single and unique cell. The reading phase consists of reading and storing in a corresponding state the value read on the $\lceil log_2(n) \rceil$ bits then making it correspond to its symbol in the alphabet $\mathcal{A}$ by the translation morphism φ. The writing consists of writing on the block of $\lceil log_2(n) \rceil$ bits the corresponding value of the symbol of the alphabet $\mathcal{A}$ by the translation morphism φ. From there, the process is in every point analogous to that of running the Turing machine M. ■

1.3.2 Tape reduction

Previously we saw that a Turing machine with a binary input alphabet is as powerful as a Turing machine with a richer alphabet. Similarly for the number of tapes: a Turing machine with one tape is as powerful as a machine with k tapes. Indeed, the following theorem demonstrates that a multi-tape Turing machine can be simulated by a single-tape Turing machine. This reduction is essential because it allows simplifying the analysis of algorithms while preserving their behavior, thus guaranteeing that the two machines accept the same inputs.

Theorem: Reduction to a single-tape Turing machine

Let there be a k-tape Turing machine M then there exists a one-tape Turing machine M' such that

$$\forall w \in \mathcal{A}, \quad M(w) \equiv M'(w)$$

Proof idea. To prove that any k-tape Turing machine, denoted M, can be simulated by a single-tape Turing machine, denoted M', we proceed as follows:

- **Encoding of the tapes**: We represent the content of the k tapes on a single tape by using a separator symbol, for example †. If the tapes contain respectively w_1, w_2, ..., w_k, we encode them as:
$$w_1 \dagger w_2 \dagger \ldots \dagger w_k$$
- **Simulation of movements**: At each step, M' reads the current state of M and the symbols under the read heads. It performs the transition defined by the transition function δ of M, by moving the read heads on the single tape. To access a symbol on a specific tape, it traverses the tape until reaching the desired symbol.
- **Management of states and halting conditions**: M' maintains the current state of M and checks if M has reached an acceptance or rejection state after each transition.

■

1.4 Encoding

In order to introduce this section let us give, in an informal manner, a definition of an encoding is a representation of objects, data or instructions in the form of a word of an alphabet which can be interpreted by a Turing machine. A more formal definition notion would be that The notion of encoding will be useful to us later in order to define a universal Turing machine.

1.4.1 Prefix encoding

Definition: Prefix encoding

A prefix encoding is a partially computable function $E : \mathbb{B}^* \mapsto \mathbb{B}^*$ such that the set of code words $\{E(x) \mid x \in dom(E)\}$ is prefix-free.

We can then associate a self-delimiting decoder with each prefix encoding.

Definition-Theorem: Self-Delimiting Decoder for a Prefix Encoding

Let $E : \mathbb{B}^* \mapsto \mathbb{B}^*$ be a prefix encoding. Let us set $\mathcal{E} := \{E(x) \mid x \in dom(E)\}$ which by definition is prefix-free. Then, there exists a partially computable function $D_E : \mathbb{B}^* \mapsto \mathbb{B}^*$ with domain $\mathcal{E}\mathbb{B}^*$:

$$D_E : \begin{cases} ew & \mapsto e \\ \mathcal{E}\mathbb{B}^* \subset \mathbb{B}^* & \mapsto \mathcal{E} \end{cases}$$

We call D_E the decoder associated with the prefix encoding E.

Proof. Assume that E is a prefix encoding. Let us set D_E the computable function with the following operation for $e \in \mathbb{B}^*$:

> For an input e in $\mathbb{B}^*$. Call E in dovetail on the $\sigma_0, \sigma_1, \ldots$ in $\mathbb{B}^*$ arranged in lexicographical order. For each computation that terminates $E(\sigma_i)$: If $E(\sigma) \leq e$ then write $E(\sigma)$ to output and accept.

Let us show that D_E is a decoder associated with E. For an input e in $\mathbb{B}^*$ two cases are possible:

- For $e = aw$ such that $a \in \mathcal{E}$ and $w \in \mathbb{B}^*$, there exists a $\sigma \in \mathbb{B}^*$ in the enumeration in lexicographical order such that $E(\sigma) = e$. As $E(\sigma) \leq_p e$ we will have during the passage through the condition that $E(\sigma)$ will be written to the output before acceptance. Thus, $D_E(e) = E(\sigma)$.
- For $e \notin \mathcal{E}\mathbb{B}^*$, by definition there does not exist any $\sigma \in \mathbb{B}^*$ in the enumeration in lexicographical order such that $E(\sigma) \leq_p e$. Thus during any passage through the condition, D_E will not accept its input. In other words D_E loops indefinitely.

We then indeed have that D_E is such that $dom(D_E) = \mathcal{E}\mathbb{B}^*$, with the outputs corresponding to that of the statement. ■

Definition-Theorem: Self-Delimiting Decoder for a Prefix Encoding

Let $E : \mathbb{B}^* \mapsto \mathbb{B}^*$ be a prefix encoding. Let us set $\mathcal{E} := \{E(x) \mid x \in dom(E)\}$ which by definition is prefix-free. Then, there exists a totally computable function:

$$D_E^{-1} : \begin{cases} ew & \mapsto v \quad \text{with } E(v) = e \\ \mathcal{E}\mathbb{B}^* & \mapsto \mathcal{E} \end{cases}$$

We call D_E the decoder associated with the prefix encoding E.

Proof. It suffices to remark that for all $e \in \mathbb{B}^*$ we have $D_E^{-1}(e) = E(D_E(e))$. ■

1.4.2 Canonical prefix encoding

Informally, a self-delimiting encoding of a word is a representation of a word that includes information about the length of this word in its own encoding. This means that the encoding contains elements that allow determining the size of the word or its end.

Definition-Property: Canonical prefix encoding

Let $i \geq 1$ be an integer. We define the function

$$\forall x \in \mathbb{B}^*, \quad E_i(x) = \begin{cases} 1^{\ell(x)}0 & \text{if } i = 0 \\ E_{i-1}(\ell(x))x & \text{otherwise} \end{cases}$$

We then have that E_i is a prefix code (see proof). We can then define the prefix-free languages

$$\forall i \in \mathbb{N}^*, \quad \mathcal{E}_i \underset{def.}{=} \{E_i(x) \mid x \in \mathbb{B}^*\}$$

Proof. Let us prove by induction on $i \geq 0$ that $\mathcal{E}_{i-1}$ is a prefix code.

Base ($i = 0$): For any string x, y, we remark that $E_0(x) = E_i(y)$ implies $1^{\ell(x)}0 = 1^{\ell(y)}0$ therefore $x = y$.

Inductive step (i): Assume that E_{i-1} is a prefix code. Let $x, y \in \mathbb{B}^*$ such that $E_i(x) = E_i(y)$. By definition, this is written $E_{i-1}(\ell(x))x = E_{i-1}(\ell(y))y$. By the induction hypothesis, $\mathcal{E}_{i-1}$ is a prefix code. The unique decodability theorem gives $E_{i-1}(\ell(x)) = E_{i-1}(\ell(y))$. By the induction hypothesis, the function E_{i-1} is injective, therefore $\ell(x) = \ell(y)$. We then deduce $x = y$. Consequently, $\mathcal{E}_i$ is a prefix code. The proposition is established by induction. ■

Property: Non-recursive form of E_i

For all $i \geq 1$ and for all $x \in \mathbb{B}^*$, we have:

$$E_i(x) = 1^{\ell(\ell^i(x))}0 \cdot \ell^{i-1}(x) \cdot \ell^{i-2}(x) \cdot \ldots \cdot \ell^0(x) \quad \dagger_i$$

where $\ell^j(x) = \ell(\ell(\ldots\ell(x)\ldots))$ with j compositions and $\ell^0(x) = x$.

Proof. We proceed by induction on $i \geq 1$.

Base ($i = 1$): By definition, $E_1(x) = E_0(\ell(x)) \cdot x$. Since $E_0(y) = 1^{\ell(y)}0$, we have $E_1(x) = 1^{\ell(\ell(x))}0 \cdot x$. Using the notation ℓ^j, this is written $E_1(x) = 1^{\ell(\ell^1(x))}0 \cdot \ell^0(x)$. The formula of the lemma for $i = 1$ gives $1^{\ell(\ell^1(x))}0 \cdot \ell^{1-1}(x) = 1^{\ell(\ell^1(x))}0 \cdot \ell^0(x)$. The two expressions coincide. The property is therefore true for $i = 1$.

Inductive step ($i \to i + 1$): Assume by the induction hypothesis that $\dagger_i$ is true for $i \geq 1$. Let us show it for $i + 1$. By definition, $E_{i+1}(x) = E_i(\ell(x)) \cdot x$. Let us apply the induction hypothesis to the input $y = \ell(x)$:

$$E_i(\ell(x)) = 1^{\ell(\ell^i(\ell(x)))}0 \cdot \ell^{i-1}(\ell(x)) \cdot \ldots \cdot \ell^0(\ell(x))$$

Let us simplify the terms using $\ell^j(\ell(x)) = \ell^{j+1}(x)$:

$$E_i(\ell(x)) = 1^{\ell(\ell^{i+1}(x))}0 \cdot \ell^i(x) \cdot \ldots \cdot \ell^1(x)$$

Substituting into the expression of $E_{i+1}(x)$, we obtain:

$$\begin{aligned} E_{i+1}(x) &= \left(1^{\ell(\ell^{i+1}(x))}0 \cdot \ell^i(x) \cdot \ldots \cdot \ell^1(x)\right) \cdot x \\ &= 1^{\ell(\ell^{i+1}(x))}0 \cdot \ell^i(x) \cdot \ldots \cdot \ell^1(x) \cdot \ell^0(x) \end{aligned}$$

This is exactly the formula of the lemma for rank $i + 1$. ■

Property: E_i is a prefix encoding

For all integers $i \geq 0$, we have E_i is a prefix encoding.

Proof. We had already shown that E_i is a prefix code. Let us set the following partially computable function E_i

For an input $e \in \mathbb{B}^*$
Initialize:
- $k \leftarrow i$
- $r_k \leftarrow e$
- $a_k \leftarrow e$

Loop while $k \geq 1$:
- $a_{k-1} \leftarrow \ell(a_k)$.
- $r_{k-1} \leftarrow a_{k-1} r_k$.
- $k \leftarrow k - 1$.

Let us prove the correctness by setting for $1 \leq k \leq i$ the loop invariant $\mathcal{I}_k$ according to which at the beginning of Step 2, when the loop variable is k, we have:

1. $a_k = \ell^{i-k}(x)$
2. $r_k = \ell^{i-k}(x) \cdot \ell^{i-k-1}(x) \cdot \ldots \cdot \ell^0(x)$

Base ($k = i$): Step 1 initializes $k = i$, $a_i = x$ and $r_i = x$. The invariant $\mathcal{I}_i$ is verified, because $a_i = \ell^{i-i}(x) = \ell^0(x) = x$ and $r_i = \ell^0(x) = x$.

Maintenance ($k \to k-1$): Assume $\mathcal{I}_k$ true for a $k > 1$. The algorithm executes the "Else" branch:

- $a_{k-1} \leftarrow \ell(a_k) = \ell(\ell^{i-k}(x)) = \ell^{i-(k-1)}(x)$.
- $r_{k-1} \leftarrow a_{k-1} \cdot r_k = \ell^{i-(k-1)}(x) \cdot (\ell^{i-k}(x) \cdot \ldots \cdot x)$.

The new values a_{k-1} and r_{k-1} satisfy the invariant $\mathcal{I}_{k-1}$ for the next pass.

Termination ($k = 1$): The loop terminates when $k = 1$. According to the invariant $\mathcal{I}_1$, we have:

- $a_1 = \ell^{i-1}(x)$
- $r_1 = \ell^{i-1}(x) \cdot \ell^{i-2}(x) \cdot \ldots \cdot x$

We will then have written on the output $1^{\ell(a_1)}0 \cdot r_1$. Substituting the values, it returns:

$$1^{\ell(\ell^{i-1}(x))}0 \cdot (\ell^{i-1}(x) \cdot \ell^{i-2}(x) \cdot \ldots \cdot x) = 1^{\ell(\ell^i(x))}0 \cdot \ell^{i-1}(x) \cdot \ldots \cdot x$$

By the lemma which precedes this is the non-recursive form of $E_i(x)$. ■

Among the prefix codes E_i those which interest us particularly are those for $i = 1, 2$. These are sufficiently important to introduce a notation which is specific to them.

Definition-Property: Elias Delta Encoding

We set the Elias Delta encoding defined for all x in $\mathbb{B}^*$ by,

$$\widehat{x} := E_2(x)$$

Furthermore we prove that $\ell(\widehat{x}) = \ell(E_2(x)) = \ell(x) + \ell(\ell(x)) + 1$. We set moreover

$$\mathcal{E} := \{\widehat{x} \mid x \in \mathbb{B}^*\}$$

Proof. We apply the non-recursive form of the lemma, $E_2(x) = 1^{\ell(\ell^2(x))}0 \cdot \ell^1(x) \cdot \ell^0(x)$, which provides us

$$\begin{aligned}\ell(E_2(x)) &= \ell(1^{\ell(\ell^2(x))}0) + \ell(\ell^1(x)) + \ell(\ell^0(x)) \\ &= \big(\ell(\ell^2(x)) + 1\big) + \ell(\ell(x)) + \ell(x) \\ &= \ell(\ell(\ell(x))) + \ell(\ell(x)) + \ell(x) + 1\end{aligned}$$

■

1.4.3 n-Code

We will now define a particular class of n-codes, called "associated" with a prefix code, which are fundamental in Kolmogorov complexity. The following recursive construction highlights a structure where the first $n-1$ words are encoded using a prefix code, allowing their delimitation, while the last word is not encoded. This particularity is crucial for the definition of conditional Kolmogorov complexity.

Definition: Associated n-Code E

Let E be a prefix encoding. For all integers $n \geq 1$, we define the function $E^n : (\mathbb{B}^*)^n \mapsto \mathbb{B}^*$ such that for all strings $x_1, \ldots, x_n$ we have:

$$E^n(x_1, x_2, \ldots, x_n) \underset{\text{def.}}{=} \begin{cases} x_1 & \text{if } n = 1 \\ E(x_1) \cdot E^{n-1}(x_2, \ldots, x_n) & \text{if } n > 1 \end{cases}$$

Property: Injectivity of an n-code and non-recursive form

For all $n \geq 1$, the function E^n defined above is injective. Furthermore, it admits the following non-recursive form:

$$E^n(x_1, \ldots, x_n) = E(x_1)E(x_2)\ldots E(x_{n-1})x_n$$

Proof. We prove subsequently the formula of the statement then the injectivity.

Non-recursive form: Let us show by induction on $n \geq 1$ that

$$E^n(x_1, \ldots, x_n) = E(x_1)\ldots E(x_{n-1})x_n$$

Base ($n = 1$): By definition, $E^1(x_1) = x_1$. The base property is verified.

Inductive step ($n \to n+1$): Assume the property true at rank n. By definition of E^{n+1} and by the induction hypothesis applied to $(x_2, \ldots, x_{n+1})$:

$$E^{n+1}(x_1, \ldots, x_{n+1}) = E(x_1) \cdot E^n(x_2, \ldots, x_{n+1})$$

$$= E(x_1) \cdot (E(x_2) \dots E(x_n)x_{n+1}) \quad \text{[By induction hypothesis]}$$
$$= E(x_1)E(x_2) \dots E(x_n)x_{n+1}$$

The property is therefore true for rank $n+1$.

Injectivity: Let $\vec{x} = (x_1, \dots, x_n)$ and $\vec{y} = (y_1, \dots, y_n)$ in $(\mathbb{B}^*)^n$ such that $E^n(\vec{x}) = E^n(\vec{y})$. Using the non-recursive form, we have:

$$E(x_1)E(x_2) \dots E(x_{n-1})x_n = E(y_1)E(y_2) \dots E(y_{n-1})y_n$$

Since E is a prefix code, the language $\mathcal{E} = \{E(z) \mid z \in \mathbb{B}^*\}$ is prefix-free. By the non-ambiguity of a prefix code we obtain

$$E(x_j) = E(y_j) \quad \text{for all } j \in \{1, \dots, n-1\}$$

Furthermore, as E is injective (since it is a code), we deduce that $x_j = y_j$ for all $j \in \{1, \dots, n-1\}$.

By removing from the non-recursive form the common prefix $E(x_1) \dots E(x_{n-1}) = E(y_1) \dots E(y_{n-1})$, it remains:

$$x_n = y_n$$

That is, $\vec{x} = \vec{y}$. The function E^n is therefore injective. ■

Definition: Language of n-codes

For $n \geq 1$, the image of the function E^n is denoted $\mathcal{E}_E^{(n)}$:

$$\mathcal{E}_E^{(n)} \underset{\text{def.}}{=} \{E^n(x_1, \dots, x_n) \mid (x_1, \dots, x_n) \in (\mathbb{B}^*)^n\}$$

Remark. Contrary to the code words of a prefix code/encoding, $\{E^n(x_1, \dots, x_n) \mid (x_1, \dots, x_n) \in (\mathbb{B}^*)^n\}$ is not prefix-free. This is due to the last term x_n which is not encoded and can potentially be a prefix of another x'_n or of another sequence. ◇

Convention:

We denote throughout the rest of the course for all strings $x_1, x_2, \dots, x_n$,

$$\langle x_1, x_2, \dots, x_n \rangle := E_2(x_1)E_2(x_2) \dots E(x_{n-1})x_n$$

We note that this is also written $\widehat{x_1}\widehat{x_2} \dots \widehat{x_{n-1}}x_n$. We set furthermore

$$\mathcal{E}^{(n)} := \{\langle x_1, x_2, \dots, x_n \rangle \mid x_1, x_2, \dots x_n \text{ in } \mathbb{B}^*\}$$

1.4.4 Manipulation of an n-Code

Property: $\mathcal{E}_E^{(n)}$ is recognizable

Let E be a prefix encoding and $n \geq 1$. There exists a partially computable function *Assert* with domain $dom(Assert) = \mathcal{E}_E^{(n)}$.

Proof. Let $n, i \geq 1$ be two integers. Let us set *Assert* as the function with the operation:

For an input e in $\mathbb{B}^*$.

Initialize $r_1 \leftarrow e$.

Loop over $k = 1, 2, \ldots, n-1$:

- $a_k \leftarrow D_E(r_k)$
- $r_{k+1} \leftarrow r_k[|a_k| : |r_k|]$, that is r_k with a_k removed.

Accept the input.

Let us set the following loop invariant for $1 \leq k < n$:

$\mathcal{I}_k$: "At the beginning of the k-th iteration of Step 2, we have $e = a_1 a_2 \ldots a_{k-1} r_k$ with $a_j \in E$ for all $j < k$."

Base ($k = 1$): At the beginning, $r_1 = e$ and a_j is empty. The invariant $\mathcal{I}_1$ is verified.

Maintenance ($k \to k+1$): Assume $\mathcal{I}_k$ true for $k < n-1$. Iteration k computes $a_k \in E$ and r_{k+1} such that $r_k = a_k r_{k+1}$. Substituting into the invariant, we thus obtain $e = a_1 \ldots a_{k-1}(a_k r_{k+1}) = a_1 \ldots a_k r_{k+1}$, that is $\mathcal{I}_{k+1}$ is verified.

Termination ($k = n-1$): The algorithm accepts after the last iteration of the loop for $k = n-1$. According to the invariant $\mathcal{I}_{n-1}$, we have $e = a_1 \ldots a_{n-2} r_{n-1}$. The last execution decomposes r_{n-1} into $a_{n-1} r_n$, where $a_{n-1} \in E$. The input thus has the form $e = a_1 \ldots a_{n-1} r_n$ with $a_j \in E$ and $r_n \in \mathbb{B}^*$, which corresponds to the definition of an element of $\mathcal{E}^{(n)}$.

Conversely, if $e \notin \mathcal{E}^{(n)}$, there will exist a step k where r_k does not begin with a word from E, and the computation $D_E(r_k)$ will not terminate. The function indeed has $\mathcal{E}_E^{(n)}$ as its domain. ■

Property: Manipulation of an n-Code: Reading

Let $n \geq 1$ and $n \geq j \geq 1$ be integers and E be a prefix encoding. There exists a totally computable function

$$D_E^{(j)} : \begin{cases} \langle\, e_1, \ldots, e_n \,\rangle & \mapsto w_j \quad \text{for } 1 \leq j \leq n \\ \mathcal{E}_E^{(n)} & \mapsto \mathbb{B}^* \end{cases}$$

with $\langle\ \rangle$ corresponding to the encoding for E (not necessarily E_2).

Proof. Let us set the computable function $D_E^{(j)}$ having the operation

For an input $e \in \mathcal{E}_E^n$.
Initialize $r_1 \leftarrow e$
Loop over $k = 1, 2, \ldots$

- $a_k \leftarrow D_E(r_k)$
- $r_{k+1} \leftarrow r_k[|a_k| + 1 : |r_k|]$, (that is r_k with w_k removed) [Note: The text says w_k, but based on context it should likely be a_k. I will translate literally as w_k per instructions, or stick to the intended meaning if it's a clear typo. Given strict fidelity instructions, I will use "removed from w_k" if that's what the French implies ("retranché de w_k"), but "retranché de w_k" usually means w_k is subtracted from it, or it is removed from w_k. Context suggests a_k (the prefix) is removed from r_k. The French text says "c'est à dire r_k retranché de w_k". I will translate literally:

"that is r_k with w_k removed" or "that is r_k subtracted by w_k". Let's stick to "that is r_k with w_k removed" as it's the standard interpretation of "retranché de" in this context of string manipulation, assuming w_k is the prefix just read.]

- If $k+1 == n$ then
  - Write r_{k+1} to output then accept.
- If $k == j$ then
  - Write $D_E^*(a_k)$ to output then accept.

We remark firstly that we necessarily have termination of the procedure because k is incremented until verifying one of the two stopping conditions. We thus already have that $D_E^{(j)}$ is totally computable.

For the call $D_E^{(j)}(e_1, \ldots, e_n)$ let us set the loop invariant for $1 \leq k \leq n$ at the beginning of iteration k of the loop:

$$I(k): \quad r_k = E(e_{k+1}) \ldots E(e_{n-1}) w_n \quad \text{and} \quad \forall 1 \leq i < k, \quad a_i = e_i.$$

Base ($k = 1$): Before the first iteration, $r = e = E(e_1) \ldots E(e_n)$. $I(1)$ is therefore true.

Maintenance (k → k+1): Assume that $I(k)$ is true for $k < j$. During iteration $k+1$ we will have:

1. $a_k \leftarrow D_E(r)$: By $I(k)$ and by definition of a self-delimiting decoding we necessarily have $a_k = E(e_k)$.
2. $r_{k+1} \leftarrow r_k[|e_k| + 1 : |r_k|]$: By $I(k)$ we then verify $E(e_{k+1}) \ldots E(e_{n-1}) e_n$.

Thus the invariant $I(k+1)$ is maintained.

Termination: The loop executes exactly j times. At the end two cases are possible: If $j = n$ then the condition $k+1$ equals n is verified so $r_{k+1} = e_n$ will be on the output. If $j < n$ then the condition k equals j will be verified so $D_E^*(a_j) = D_E^*(E(e_j)) = e_j$ will be on the output. In both cases we have termination with the output corresponding to the function of the statement. ■

Property: Manipulation of an n-Code: Update

Let $n \geq 1$ and $1 \leq j \leq n$ be two integers and E be a prefix encoding. Then there exists a totally computable function

$$Updt_E^{(j)} : \begin{cases} \langle \langle e_1, \ldots, e_n \rangle, x \rangle & \mapsto \langle e_1, \ldots, e_{j-1}, x, e_{j+1}, \ldots, e_n \rangle \\ \langle \mathcal{E}_E^{(n)}, \mathbb{B}^* \rangle & \mapsto \mathcal{E}_E^{(n+1)} \end{cases}$$

with $\langle\ \rangle$ corresponding to the encoding for E.

Proof idea. The proof does not present any difficulty, that is why we will informally describe the construction of *Updt*. For an input $\langle \langle e_1, \ldots, e_n \rangle, x \rangle$. Extract $\alpha \leftarrow \langle e_1, \ldots, e_n \rangle$ and x using respectively $D_E^{(1)}(\alpha)$ and $D_E^{(2)}(\alpha)$. Read and concatenate on the output the encodings of the first j elements of α with $D_E^{(1)}(\alpha), ..., D_E^{(j-1)}(\alpha)$ which gives $E(e_1)E(e_2)\ldots E(w_{j-1})$. Concatenate on the output $E(x)$ if $j < n$ otherwise concatenate x. Concatenate on the output the remainder of α using $D_E^{(j+1)}(\alpha)$ and $D_E^{(n)}(\alpha)$. We will then have the desired result. ■

Thesis: Church-Turing Continuation

These properties of manipulation of an n-Code demonstrate the computability of fundamental operations on an n-Code. These capabilities are directly analogous to the usual manipulations on a list type object in computer science (access, modification).

Consequently, in the rest of our developments, we will implicitly manipulate n-Codes as such lists. This abstraction will allow us to focus on the higher-level logic, without reiterating the details of their encoding and decoding.

1.4.5 Encoding of Turing Machines

We will now present an encoding of Turing machines which will be useful to us in particular for the construction of a universal Turing machine. This encoding stores the states, the number of tapes, the transitions of a Turing machine, the size of the input alphabet (here it will be two as it is equal to $\mathbb{B}$) and the size of the work alphabet.

Definition: Code of a k-tape Turing machine

Let M be a k-tape Turing machine

$$M = (Q, \mathbb{B}, \Gamma, \#, \delta, q_0, q_a)$$

We will denote $\prod_{i=1}^{n} x_1 \underset{not.}{=} \langle x_1, \dots, x_n \rangle \in \mathcal{E}_2^{(n)}$. We set the following indexings of the states, input alphabets, reading direction:

- Let us set $Q = \{q_1, \dots, q_{n_1}\}$ the states of M with, without loss of generality, $q_1 = q_0$ and $q_2 = q_a$. We index, in binary writing, the states Q, that is we set the function
$$\mathcal{I}_Q : q_i \in Q \mapsto bin(i) \in \mathcal{N}$$
- Let us set $\Gamma = \{s_1, \dots, s_{n_2}\}$ the work alphabet [a] of M with, without loss of generality, $s_{n_2} = \#$. We index, in binary writing, the letters of Γ, that is we set the function
$$\mathcal{I}_\Gamma : s_i \in \Gamma \mapsto bin(i) \in \mathcal{N}$$
- We index the direction of movement of a read head, that is we set the function:

  $\mathcal{I}_\leftrightarrow : \{\leftarrow; \rightarrow; |\} \mapsto \{1; 2; 3\}$ such that $\mathcal{I}_\leftrightarrow(\leftarrow) = 1$ and $\mathcal{I}_\leftrightarrow(|) = 2$ and $\mathcal{I}_\leftrightarrow(\rightarrow) = 3$

We will define the encoding of a transition then of M with the help of these indexings:

- Let there be an arbitrary transition of M for $q \in Q$ and $r \in \Gamma^k$
$$\delta(q, \underbrace{(r^{(1)}, \dots, r^{(k)})}_{=r}) = (q', (s^{(1)}, \dots, s^{(k)}), (\theta^{(1)}, \dots, \theta^{(k)}))$$
  We then define the encoding of this transition by
$$\Delta_{q,r} \underset{def.}{=} \langle \mathcal{I}_Q(q), \mathcal{I}_Q(q'), \langle \prod_{i=1}^{k} \langle \mathcal{I}_\Gamma(r^{(i)}), \mathcal{I}_\Gamma(s^{(i)}), \mathcal{I}_\leftrightarrow(\theta^{(i)}) \rangle \rangle \rangle$$

- We then define the encoding of the Turing machine M

$$\langle M \rangle \underset{def.}{=} \langle bin(k), bin(n_1), bin(n_2), \langle \prod_{(q,r)\in Q\times\Gamma^k} \Delta_{q,r} \rangle \rangle$$

that is the encoding of the tuple comprising the number of tapes, the number of states, the size of the work alphabet and the encoding of the transitions [b]. Furthermore we set $\langle \mathcal{T} \rangle$ the set of encodings of Turing machines and the encoding of transitions $\Delta_{\langle \mathcal{T} \rangle}$.

[a] We recall that by definition the input alphabet is contained in the work alphabet, that is here $\mathbb{B} \subset \Gamma$.

[b] the order of the $\Delta_{q,r}$ does not matter in the product

We will now introduce partially computable functions allowing to verify that a word indeed belongs to $\langle \mathcal{T} \rangle$ and to extract the corresponding transition.

Property:

There exists a partially computable function $Assert : \mathbb{B}^* \mapsto \mathbb{B}^*$ with domain $dom(Assert) = \langle \mathcal{T} \rangle$.

Proof. We resume the indexing functions of the states, of the work alphabet defined in the definition of the code of a Turing machine. The function $Assert$ has the following operation:

- For an input x in $\mathbb{B}^*$
- Verify that x belongs to $\langle \mathcal{N}, \mathcal{N}, \mathcal{N}, \mathbb{B}^* \rangle$ which allows us to write

$$x = \langle k, n_1, n_2, \zeta \rangle$$

where k, n_1, n_2 are supposed to represent respectively the number of tapes, the number of states, the size of the work alphabet.

- We verify that ζ belongs to $\mathcal{E}^{(n_1-1)\times n_2^k}$: Assuming that x is the code of a Turing machine, we must examine ζ, which is supposed to represent the transitions of this machine. This implies that ζ must be encoded as a function δ having as domain $Q/\{q_a\} \times \Gamma^k$. For this, we verify that ζ represents a $(n_1 - 1) \times n_2^k$-Code, which means that ζ must belong to $\mathcal{E}^{(n_1-1)\times n_2^k}$. We can thus write:

$$\zeta = \langle \prod_{i=1}^{(n_1-1)\times n_2^k} \Delta_i \rangle$$

where each Δ_i is supposed to represent a valid transition of the Turing machine.

- Verify that each Δ_i is indeed of the form $\langle q, q', \langle \prod_{j=1}^{k} \langle r_j, s_j, \theta_j \rangle \rangle \rangle$, where:
  - q and q' are binary integers representing states according to the indexing $\mathcal{I}_Q$ defined previously. We verify that $1 \leq q \leq n_1$ and $1 \leq q' \leq n_1$.
  - For each j from 1 to k:
    - r_j, s_j are binary integers representing symbols of the work alphabet, in the indexing $\mathcal{I}_\Gamma$, that is we verify that $1 \leq r_j \leq n_2$ and $1 \leq s_j \leq n_2$.

* θ_j is a binary integer representing a direction, that is we verify $1 \leq bin(\theta_j) \leq 3$.

- Verify that the set of transitions is complete and deterministic. That is, for each possible pair (q, r), where q is a non-accepting state ($q \neq 2$) and r is a k-tuple of alphabet symbols (obtained by decomposing each Δ_i and verifying the r_j), there exists **exactly one** corresponding transition in the set of Δ_i. This is crucial to ensure that the Turing machine is well-defined. More formally:
  - We verify that for all $1 \leq i \leq n_1 - 1$ and $(a_1, \dots a_k)$ such that $\forall j \leq k, 1 \leq a_j \leq n_2$. There exists l such that

$$\Delta_l = \langle\, i, q', \langle \prod_{j=1}^{k} \langle\, a_j, s_j, \theta_j \,\rangle \,\rangle \,\rangle$$

- If all the verifications above succeed, we accept x, otherwise the machine loops indefinitely.

We indeed have that this effective procedure accepts its input if and only if it is in $\langle\, \mathcal{T} \,\rangle$, which clearly shows that *Assert* is the function of the statement. ■

Property: Manipulation of $\langle\, \mathcal{T} \,\rangle$

For any Turing machine M with s states (Let $S = \{1, 2, \dots, s\}$). There exists a totally computable function

$$Extract_{\langle \mathcal{T} \rangle} : \begin{cases} (\langle\, M \,\rangle, q, r_1, \dots, r_k) & \mapsto \Delta_{(q,r)} \\ \langle\, \langle\, \mathcal{T} \,\rangle, S, \underbrace{\mathcal{N}, \dots, \mathcal{N}}_{k} \rangle & \mapsto \Delta_{\langle \mathcal{T} \rangle} \end{cases}$$

Proof. We resume the indexing functions of the states, of the work alphabet defined in the definition of the code of a Turing machine. Let us define the computable function *Extract* having the following operation:

For an input $\langle\, \langle\, M \,\rangle, q, r_1, \dots, r_k \,\rangle$ with $\langle\, M \,\rangle = \langle\, k, n_1, n_2, \langle \prod_{i=1}^{(n_1-1)\times n_2^k} \Delta_i \,\rangle \,\rangle \in \langle\, \mathcal{T} \,\rangle$

Search for the transition that we wish to extract by iterating over the Δ_i until finding the transition corresponding to $(q, r_1, \dots, r_k)$:

- For each $\Delta_i = \langle\, q_i, q_i', \langle \prod_{j=1}^{k} \langle\, r_j^i, s_j^i, \theta_j^i \,\rangle \,\rangle \,\rangle$:
  * If $q_i = q$ and $\forall 1 \leq j \leq k,\ r_j^i = r_j$:
    · Write Δ_i to output then accept.

■

Thesis: Church Turing

Thanks to the function *Extract* opposite we can extract a given transition. We assume in the following that in an effective procedure one can call a function for which one holds the code of the corresponding Turing machine.

1.5 Semi-decidable and Enumerable Language

We are going to explore the notions of semi-decidable and enumerable languages. A language is enumerable if it is possible to enumerate all its words, potentially with repetition, by a Turing machine.

The notion of semi-decidable language, on the other hand, is more subtle. A language is considered semi-decidable if there exists a Turing machine that accepts the words belonging to it. However, for words that do not belong to this language, the Turing machine may enter an infinite loop, never halting. This is what gives semi-decidability its name: it is possible to determine if a word belongs to a semi-decidable language, but it is impossible to conclude definitively if a word is not part of it.

We will see, however, that semi-decidable languages are enumerable and vice versa. Thus, these two notions are equivalent.

1.5.1 Definitions

Definition: Semi-decidable language

A language $L \subset \mathbb{B}^*$ is said to be semi-decidable if there exists $\phi : \mathbb{B}^* \mapsto \mathbb{B}^*$, a partially computable function with domain $dom(\phi) = L$.

Remark. Let us make two remarks:

- It is also called recursively enumerable; this is due to the equivalence between an effectively enumerable language and a semi-decidable one (see below).
- This definition is equivalent to the existence of a Turing machine M such that

$$\forall x \in \mathbb{B}^*, \quad x \in L \iff \downarrow M(x)$$

◇

Intuitively, a language is enumerable if each word composing it can be displayed one after the other, or more formally:

Definition: Effectively enumerable language

A language $L \subset \mathbb{B}^*$ is said to be effectively enumerable if there exists a partially computable function π such that $\{\pi(i) \mid i \in dom(\pi)\} = L$.

Remark. A few remarks:

- One sometimes finds in the literature the definition "A language $L \subset \mathbb{B}^*$ is said to be effectively enumerable if there exists a surjective partially computable function $\pi : \mathcal{N} \mapsto L$". This definition is strictly speaking false, because it does not necessarily imply that the set $\{\pi(i) \mid i \in dom(\pi)\}$ equals L but only that $\pi(\mathcal{N})$ equals L.
- An enumeration of a language L can contain the same occurrence several times in its enumeration.
- A finite language is necessarily effectively enumerable. Indeed, for $L = \{\nu_1, \nu_2, \ldots, \nu_n\}$, it suffices to write a procedure with n cases of the form "If the input is equal to i, write ν_i as output".

◇

1.5.2 Reformulation of enumerable languages

We have the following theorem which states that by removing duplicates from an enumeration $\pi : \mathcal{N} \to L$, it is possible to find a bijection $\pi_{bij} : \{1, 2, \ldots, \delta(L)\} \subset \mathcal{N} \mapsto L$. Conversely, from such a bijection, it is possible to construct an enumeration of L.

Theorem: Reformulation of an enumerable language

We have equivalence between

1. L is an effectively enumerable language
2. There exists a bijective effective enumeration $\pi : \{1, 2, \ldots, |L|\} \mapsto L$ such that $dom(\pi) = \{1, 2, \ldots, |L|\}$

Note that it is possible that $|L| = \infty$.

Proof. (1 $\implies$ 2) Suppose that L is an effectively enumerable language, that is, there exists π_L, an effective enumeration of L. Let us then define π as a function with the behavior

For an input i in $\mathcal{N}$

Initialize $\theta \leftarrow \{\,\}$.

Call π_L in dovetail on $j = 0, 1, 2, \ldots$. For each computation $\pi_L(j)$ that halts, do:

- If $\pi_L(j) \notin \theta$ then [addition condition]
  - Assign $\theta \leftarrow \theta \cup \{\pi_L(j)\}$
- If $size(\theta) = i$ then [stopping condition]
  - Write $\pi_L(j)$ and accept.

Let us show the surjectivity of π: Let $\nu \in L$. Let $j_1, j_2 \ldots$ be such that $\pi_L(j_n)$ is the n-th computation of the dovetail that halts. Since π_L enumerates L, there necessarily exists a smallest index m such that $\pi_L(j_m) = \nu$.

For each processing of the computations $\pi_L(j_1), \pi_L(j_2), \ldots, \pi_L(j_{m-1})$, we potentially have an addition to θ. Let r be the total number of unique additions performed during the processing of these $m - 1$ first computations. During the processing of $\pi_L(j_m)$ at the beginning of the addition condition, the size of θ is therefore r.

By the minimality of m, the word $\nu = \pi_L(j_m)$ is not in $\{\pi_L(j_1), \pi_L(j_2), \ldots, \pi_L(j_{m-1})\}$, therefore $\nu \notin \theta$. The addition condition is thus satisfied: ν is added to θ and its size becomes $r + 1$. Thus for the input $i = r + 1$. The computation of $\pi(r + 1)$ therefore returns ν.

Let us show the injectivity of π: Let a and b be two integers in the domain of π such that $a < b$. We must show that $\pi(a) \neq \pi(b)$.

By the addition and stopping conditions, we observe that $\pi(i)$ is the i-th word added to θ. Thus, letting θ_i be the set θ at the halting of the call $\pi(i)$, we have $\theta_i = \{\pi(1), \pi(2), \ldots, \pi(i)\}$.

Since $a < b$, we have $a \leq b - 1$. This implies that $\pi(a)$ is an element of the set $\{\pi(1), \ldots, \pi(b - 1)\}$, which is θ_{b-1}. The word $\pi(b)$ is, by the addition and stopping conditions, the word which, once added to θ_{b-1}, increased the size of θ to b. By the addition condition, this is only possible if $\pi(b)$ is not already in θ_{b-1}.

In summary, $\pi(a) \in \theta_{b-1}$ and $\pi(b) \notin \theta_{b-1}$, which implies $\pi(a) \neq \pi(b)$.

$(1 \impliedby 2)$ It is immediate. If there exists an effective enumeration π [which is a] bijection of L such that $dom(\pi) = \{1, 2, \ldots, |L|\}$, then in particular π is a partially computable function such that $\{\pi(i) \mid i \in dom(\pi)\} = L$. We indeed have that L is an effectively enumerable language. ■

1.5.3 Equivalence of enumerable and semi-decidable languages

We will demonstrate the equivalence between semi-decidable languages and effectively enumerable languages. This equivalence is particularly useful, because it is often simpler to prove that a language is semi-decidable than to show that it is enumerable.

Theorem: semi-decidable $\iff$ enumerable

Let $L \subset \mathbb{B}^*$ be a language. We then have the following equivalence:

$$L \text{ is semi-decidable} \iff L \text{ is enumerable}$$

Proof. $(\implies)$ Suppose that L is semi-decidable. Then there exists a partially computable function ϕ with domain $dom(\phi) = L$. Let us define the function π with the behavior

> For an input i in $\mathcal{N}$. Call ϕ in dovetail on $\sigma_0, \sigma_1, \ldots$ of $\mathbb{B}^*$ arranged in lexicographical order. For $\phi(\sigma)$ the i-th computation that halts, write σ as output and accept.

We must now verify that the function is an enumeration of L, that is, a surjection. Let $\nu \in L$. By definition of a semi-decidable set, $\nu \in dom(\phi)$. Thus in a dovetail call the computation $\phi(\nu)$ will halt. Suppose that $\phi(\nu)$ is the i-th computation to halt in the dovetail call. Then $\pi(i)$ is equal by construction to ν.

$(\impliedby)$ Suppose L is enumerable. There exists by definition a surjective partially computable function $\pi : \mathcal{N} \mapsto L$. Let us then define the function ϕ having the behavior

> For an input e in $\mathbb{B}^*$. Call $\pi(t)$ in dovetail on $t = 0, 1, 2, \ldots$. For each computation $\pi(t)$ that halts, if the computation equals e then accept.

Let us show $L \subset dom(\phi)$: Let ν be in L. As π is surjective there exists $i \in dom(\pi)$ such that $\pi(i) = \nu$. Thus for the call $\phi(\nu)$, the computation $\pi(i)$ of the dovetail will stop and the stopping condition will be verified, hence the acceptance of $\phi(\nu)$. Therefore ν belongs to $dom(\phi)$.

Let us show $L \supset dom(\phi)$: Let ν be in $dom(\phi)$. This means that $\phi(\nu)$ is accepted. Thus necessarily by the stopping condition there exists a t such that $\pi(t)$ equals the input, that is ν. Therefore ν belongs to L. ■

1.5.4 Enumerability of standard sets

Without explicitly stating it, we have previously shown that several languages are semi-decidable. By equivalence with enumerable languages, we can therefore state that these languages are also enumerable.

Property: Effective enumerability of standard sets

The languages $\mathbb{B}^*, \mathcal{N}, \mathcal{Z}, \mathcal{E}_i, \mathcal{E}_i^n$ and $\langle \mathcal{T} \rangle$ are effectively enumerable.

Proof. This is a consequence of properties/theorems that we have already shown. For $\mathbb{B}^*$, it suffices to posit a partially computable function that accepts all its inputs. For the others, we had given for each language a partially computable function *Assert* such that its domain is the language in question. Thus, by definition, these languages are semi-decidable, therefore by equivalence effectively enumerable. ■

1.6 Decidable Language

In the previous section, we studied semi-decidable languages. That is to say, those for which there exists an effective procedure that halts only for inputs in this language. However, one might also sometimes wish to know if an input is not in a given language. This is what the notion of decidable language defines.

1.6.1 Definition

Definition: Decidable language

A language $L \subset \mathbb{B}^*$ is decidable or recursive if there exists a totally computable function $\phi : \mathbb{B}^* \mapsto \{0, 1\}$ such that

$$\forall x \in \mathbb{B}^*, \quad \phi(x) = \begin{cases} 1 \text{ if } x \in L \\ 0 \text{ if } x \notin L \end{cases}$$

We say that ϕ decides L.

Remark. This definition is equivalent to the existence of a Turing machine M such that

$$\forall w \in \mathbb{B}^*, \quad \begin{cases} w \in L \iff \downarrow M(w) = 1 \\ w \notin L \iff \downarrow M(w) = 0 \end{cases}$$

◇

The following theorem establishes an essential relation between the decidability and semi-decidability of languages. It shows that the property of being decidable for a language L is equivalent to that of L and its complement $\mathbb{B}^*/L$ being semi-decidable. This equivalence justifies the use of the term "semi-decidable".

Theorem: L is decidable $\iff$ L and $\mathbb{B}^*/L$ are semi-decidable

Let $L \subset \mathbb{B}^*$ be a language; we have the equivalence

$$L \text{ is decidable} \iff L \text{ semi-decidable and } \mathbb{B}^* \setminus L \text{ is semi-decidable}$$

Proof. ($\implies$) Suppose that there exists a totally computable function $\phi : \mathbb{B}^* \mapsto \{0, 1\}$ which decides L. Let us define the function ϕ_1 having the behavior

For an input e in $\mathbb{B}^*$, if $\phi(e) = 1$ then accept.

We have immediately, by definition of decidability, that every input in L will be accepted. Conversely, let us define ϕ_2 having the same behavior with the exception of changing "if $\phi(e) = 1$" to "if $\phi(e) = 0$". We then also have immediately by definition of decidability that every input in $\mathbb{B}^* \setminus L$ will be accepted.

($\Longleftarrow$) Suppose L is semi-decidable and $\mathbb{B}^* \setminus L$ is semi-decidable. There exist two partially computable functions ϕ_1 and ϕ_2 with respective domains L and $\mathbb{B}^* \setminus L$. Let us then define the function ϕ with the following behavior

For an input e in $\mathbb{B}^*$

Loop over $t = 0, 1, 2, \ldots$:

- If $\phi_1(e)$ terminates in t transitions then
  - Write 1 as output and accept
- If $\phi_2(e)$ terminates in t transitions then
  - Write 0 as output and accept

We observe already that any input e belongs either to L or to $\mathbb{B}^* \setminus L$. Suppose that e belongs to L; then $\phi_1(e)$ terminates in a finite number of transitions, let us call it t'. Then in the call $\phi(e)$, when iteration t' is reached, a stopping condition will be verified, hence the halt with 1 as output. Similarly, for an input e in $\mathbb{B}^* \setminus L$, we will have $\phi(e)$ stopping with 0 as output. Thus we have just proved that ϕ terminates for every input with the output consistent with the definition of L being decidable by ϕ. ■

1.6.2 Undecidable Language and Halting Problem

We now introduce the notion of undecidable language.

Definition: Undecidable language

A language L is said to be undecidable if it is not decidable.

Remark. In terms of Turing machines, this is equivalent to saying that there does not exist a Turing machine such that

$$\forall w \in \mathbb{B}^*, \quad \begin{cases} w \in L & \Longleftrightarrow \downarrow M(w) = 1 \\ w \notin L & \Longleftrightarrow \downarrow M(w) = 0 \end{cases}$$

◇

The halting problem consists of determining, for a given program and a specific input, whether this program will halt or continue to execute indefinitely.

The proof of the undecidability of the halting problem relies on a proof by contradiction: if a Turing machine could decide whether a program halts, one could construct a program that halts if and only if the machine predicts that it does not halt, leading to a contradiction. This proves that no machine can solve the halting problem.

Theorem: Undecidability of Halting

The language $\mathcal{H} \subset \mathbb{B}^*$ such that [a]

$$\mathcal{H} = \{\langle \langle M \rangle, p \rangle \mid M \in \mathcal{T},\ p \in \mathbb{B}^* \text{ and } \downarrow M(p)\}$$

is undecidable.

[a] Recall that $\mathcal{T}$ represents the set of Turing machines

Proof. By contradiction, suppose that there exists a computable function that decides $\mathcal{H}$. In terms of Turing machines, this amounts to saying that there exists a Turing machine H such that

$$\forall \langle M \rangle \in \langle \mathcal{T} \rangle, \quad \forall x \in \mathbb{B}^*, \quad \begin{cases} \langle \langle M \rangle, p \rangle \in \mathcal{H} \iff \downarrow H(\langle M \rangle, p) = 1 \\ \langle \langle M \rangle, p \rangle \notin \mathcal{H} \iff \downarrow H(\langle M \rangle, p) = 0 \end{cases}$$

We can construct a Turing machine P with the following behavior

$$\forall \langle M \rangle \in \langle \mathcal{T} \rangle, \quad \begin{cases} \downarrow P(\langle M \rangle) & \text{if } \downarrow H(\langle M \rangle, \langle M \rangle) = 0 \\ \circlearrowleft P(\langle M \rangle) & \text{if } \downarrow H(\langle M \rangle, \langle M \rangle) = 1 \end{cases}$$

Thus, for an input $\langle M \rangle \in \langle \mathcal{T} \rangle$, the machine P calls $H(\langle \langle M \rangle, \langle M \rangle \rangle)$ and, according to the output of this call, accepts or loops indefinitely.

We have now constructed P, so the encoding $\langle P \rangle$ exists. Let us now examine the call $P(\langle P \rangle)$ to establish a contradiction:

- If $\downarrow P(\langle P \rangle)$ then by the construction of P we have $H(\langle P \rangle, \langle P \rangle) = 0$, that is $\langle \langle P \rangle, \langle P \rangle \rangle \notin \mathcal{H}$ or equivalently $\circlearrowleft P(\langle P \rangle)$. We therefore have $\downarrow P(\langle P \rangle)$ and $\circlearrowleft P(\langle P \rangle)$, which is absurd.
- If $\circlearrowleft P(\langle P \rangle) = 0$ then by the construction of P we have $H(\langle P \rangle, \langle P \rangle) = 1$, that is $\langle \langle P \rangle, \langle P \rangle \rangle \in \mathcal{H}$ or equivalently $\downarrow P(\langle P \rangle)$. We therefore have $\circlearrowleft P(\langle P \rangle)$ and $\downarrow P(\langle P \rangle)$, which is absurd.

In both cases, we obtain a contradiction, which is absurd. ■

1.7 Universal Turing Machine

1.7.1 Compact Enumeration

Definition-Property: Effective compact enumeration

We call a compact enumeration of $\langle \mathcal{T} \rangle$ a totally computable function $\pi : i \in \mathcal{N} \mapsto \langle M_i \rangle \in \langle \mathcal{T} \rangle$ satisfying

$$\forall M \in \mathcal{T}, \quad \exists i \in \mathcal{N}, \quad M \equiv M_i$$

Moreover, such a compact enumeration exists.

Proof. It is immediate. We know that $\langle \mathcal{T} \rangle$ is semi-decidable. It then suffices to take an effective enumeration of $\langle \mathcal{T} \rangle$ which, as one can observe, is a compact enumeration. ■

Remark. Two remarks:

- A compact enumeration is not necessarily an effective enumeration of $\langle \mathcal{T} \rangle$. Indeed, there may exist a code of a Turing machine $\langle M \rangle$ such that there is no i with $\pi(i) = \langle M \rangle$.

- This definition is not a standard definition. I introduced it because it will be necessary for me to rigorously define certain subsequent definitions. In the literature, this notion is captured by an enumeration of partially computable functions. Strictly speaking, it is not possible to enumerate partial functions (a code of a partially computable function does not exist; it is the code of the Turing machine computing it that exists).

◇

Definition: "Enumeration" of partially computable functions

We call an enumeration of partially computable functions a compact enumeration π of the set of Turing machine codes $\langle \mathcal{T} \rangle$. We will then denote

$$\pi : i \in \mathcal{N} \mapsto \langle \phi_i \rangle \in \langle \mathcal{T} \rangle$$

with the convention that $\langle \phi_i \rangle$ is the code of a Turing machine computing the partially computable function ϕ_i.

1.7.2 Universal Turing Machine

Definition-Theorem: Existence of a universal Turing machine

There exists a Turing machine called Universal Turing Machine $\mathcal{U}$ associated with

- $I : i \in \mathcal{N} \mapsto \sigma \in \mathbb{B}^*$ a prefix encoding
- $\pi : i \in \mathcal{N} \mapsto M_i \in \langle \mathcal{T} \rangle$ a compact enumeration of Turing machines

verifying the following points

1. If $e = \sigma\, p \in \{I(i)p \mid i \in \mathcal{N}, p \in \mathbb{B}^*\}$ then
$$\mathcal{U}(\sigma\, p) \underset{h}{\equiv} M_i(p)$$
2. Otherwise $\circlearrowleft\, \mathcal{U}(e)$ loops indefinitely without writing to the output, that is ε.

Proof. Let us construct a Turing machine $\mathcal{U}$ whose behavior we will describe for an input $e = e_1 e_2 \dots e_l \in \mathbb{B}^*$:

Recovery of $\langle M \rangle$ to be simulated and its input p:

- Assign $\sigma \leftarrow D_I(e)$ and $i \leftarrow D_I^*(e)$ and $p \leftarrow e_{|\sigma|} e_{\sigma_{|\sigma|+1}} \dots e_l$

  (With $D_{\mathcal{I}}$ the self-delimiting decoding associated with I, thus $I(i) = \sigma$ and $e = \sigma p$. Moreover, if e does not start with a word in $\{I(i) \mid i \in \mathcal{N}\}$ then this step loops indefinitely)

- Assign $\langle M \rangle \leftarrow \pi(i)$.

Role of tapes and initialization: Let us recall the elements of the definition of a Turing machine code. Let us explain the role of each of the tapes of $\mathcal{U}$ and give their respective initializations:

- The first tape contains the input p and will remain unchanged.
- The second tape contains the encoding of $\langle M \rangle$ and will also remain unchanged

$$\langle M \rangle = \langle n_1, n_2, k, \langle \prod_{j=1}^{(n_1-1)\times n_2^k} \Delta_j \rangle \rangle$$

  By definition of a Turing machine code, we have then n_1 is the number of states, n_2 the size of the working alphabet, k the number of tapes, and $\langle \prod_{j=1}^{(n_1-1)\times n_2^k} \Delta_j \rangle$ the representation of the transitions of M.

- The third tape represents the content of the k tapes in the form of a k-Code, that is $\langle tape_k, \ldots, tape_k \rangle$, where $tape_i \in \mathbb{B}^*$ corresponds to the content of the i-th tape of M. Thus initially, the third tape contains $\langle p, \varepsilon, \ldots, \varepsilon \rangle \in \mathcal{E}^{(k)}$, representing the input tape of M with the other tapes empty.
- The fourth tape represents the position of the k read/write heads also in the form of a k-Code, that is $\langle pos_1, \ldots, pos_k \rangle$ where pos_i represents the position of the i-th head of M. Thus the fourth tape initially contains $\langle 0, 0, \ldots, 0 \rangle$, corresponding to the initial positions of the read/write heads.
- The fifth tape represents $(q, (r_1, \ldots, r_k)) \in Q \times (\Gamma^k)$, where q is the current state and r_j is the letter pointed to by the j-th read/write head, in the form $\langle \mathcal{I}_Q(q), \langle e_1, \#, \#, \ldots, \# \rangle \rangle$ in $\langle \mathcal{N}, \mathcal{E}^{(k)} \rangle$. Thus initially the fifth tape contains $\langle \mathcal{I}_Q(q_0), \langle e_1, \#, \#, \ldots, \# \rangle \rangle$.
- The sixth tape represents the output of M and must contain successively the same content as the output of M so that $\mathcal{U}(\sigma p)$ and $M(p)$ have the same output history. Initially, the sixth tape is therefore empty.

Behavior of $\mathcal{U}$: In the following, we will conflate a state, a letter of the working alphabet with its corresponding indexing by $\mathcal{I}_Q$ and $\mathcal{I}_\Gamma$ respectively. For example "q_0" should be read as $I_Q(q_0)$, that is "1". Iterate successively the following steps:

Step 1. We retrieve the transition to be performed $\langle q, \langle r_1, \ldots, r_k \rangle \rangle$, that is the value of the 5th tape, which represents $(q, (r_1, \ldots, r_k)) \in Q \times (\Gamma^k)$. We iterate over the set of transition encodings until we obtain

$$\Delta_{(q,r)} = \langle q, q', \langle \prod_{j=1}^{k} \langle r_j, s_j, \theta_j \rangle \rangle \rangle$$

Step 2. We update the third tape based on $\Delta_{(q,r)}$. That is, for a value of $\langle tape_1, tape_2, \ldots tape_k \rangle$ for the third tape (representing the k tapes of M to be simulated) and $\langle p_1, \ldots p_k \rangle$ the value of the fourth tape (representing the positions of the read/write heads), perform the following updates for all $1 \leq j \leq k$:

  * If $1 \leq j \leq k-1$ then replace the p_j-th letter of $tape_j$ with s_j
  * If $j = k$ then replace the p_k-th letter of $tape_k$ with s_k then copy $tape_k$ onto the sixth tape of $\mathcal{U}$. (This ensures that $\mathcal{U}(e)$ and $M(p)$ have the same output history)

Note that if the word is not long enough for writing the letter at a given position, then extend the word with blank symbols # then perform the writing.

Step 3. We update the fourth tape based on $\Delta_{(q,r)}$. That is: For $\langle\, p_1, p_2, \ldots, p_k \,\rangle$ the positions of the read/write heads, we perform the following update for all $1 \leq j \leq k$:

* If $\theta_j = 1$ then tape 4 $\leftarrow \langle\, p_1, \ldots, p_j - 1, \ldots, p_k \,\rangle$
* If $\theta_j = 2$ then tape 4 $\leftarrow \langle\, p_1, \ldots, p_j + 0, \ldots, p_k \,\rangle$
* If $\theta_j = 3$ then tape 4 $\leftarrow \langle\, p_1, \ldots, p_j + 1, \ldots, p_k \,\rangle$

Step 4. We update the fifth tape, denoted $\langle\, q, \langle\, l_1, \ldots, l_k \,\rangle\,\rangle$, with the new letters pointed to by the read/write heads. That is, let $\langle\, p_1, p_2, \ldots, p_k \,\rangle$ be the value of the fourth tape and $\langle\, x_1, x_2, \ldots x_k \,\rangle$ the value of the third tape, then we perform the following updates for all $1 \leq j \leq k$:

* Replace q with q'. If $q' = 1$ then $\downarrow \mathcal{U}(e)$.
* Replace l_j with the p_j-th value of x_j (if it does not exist, replace with #).

Then we reiterate these steps, that is, Goto Step 1.

■

1.7.3 Enumerative Universal Turing Machine

Definition-Theorem: Existence of an enumerative universal Turing machine

Let $\pi : i \in \mathcal{N} \mapsto \langle\, M_i \,\rangle \in \langle\, \mathcal{T} \,\rangle$ be a compact enumeration of Turing machines. There exists a universal Turing machine $\mathcal{U}^\pi$ called enumerative universal Turing machine associated with π such that

1. If $e = \langle\, i, p \,\rangle \in \langle\, \mathcal{N}, \mathbb{B}^* \,\rangle$ then
$$\mathcal{U}(i, p) \underset{h}{\equiv} M_i(p)$$
2. Otherwise $\circlearrowleft \mathcal{U}(e)$ loops indefinitely without writing to the output ($= \varepsilon$).

Proof. It suffices to define $\mathcal{U}^\pi$ as a universal Turing machine associated with the prefix encoding $I : i \in \mathcal{N} \mapsto \widehat{i} \in \{\widehat{i} \mid i \in \mathcal{N}\}$ and the enumeration π. Indeed, let us notice that, for all $i \in \mathcal{N}$ and $p \in \mathbb{B}^*$, we have
$$I(i)\, p = \widehat{i}\, p = \langle\, i, p \,\rangle$$
Let us now verify that $\mathcal{U}^\pi$ is indeed a universal Turing machine

1. If $e = I(i)p \in \{I(i)p \mid i \in \mathcal{N}, p \in \mathbb{B}^*\}$ then we can write $e = \widehat{i}\, p = \langle\, i, p \,\rangle$. We then verify
$$\mathcal{U}^\pi(I(i)p) \underset{h}{\equiv} \mathcal{U}(i, p) \underset{h}{\equiv} M_i(p)$$
2. Otherwise $\circlearrowleft \mathcal{U}^\pi(e)$ loops indefinitely without writing to the output.

■

1.7.4 Auxiliary enumerative universal Turing machine

Definition-Theorem: ∃ Auxiliary enumerative universal Turing machine

Let $\pi : i \in \mathcal{N} \mapsto \langle M_i \rangle \in \langle \mathcal{T} \rangle$ be a compact enumeration of Turing machines. There exists a universal Turing machine $\mathcal{U}^\pi$ called auxiliary enumerative universal Turing machine associated with π, such that

1. If $e = \langle y, i, p \rangle \in \langle \mathbb{B}^*, \mathcal{N}, \mathbb{B}^* \rangle$ and
$$\mathcal{U}^\pi(y, i, p) \equiv \begin{cases} M_i(y, p) & \text{If } y \neq \varepsilon \\ M_i(p) & \text{Otherwise} \end{cases}$$
2. Otherwise $\circlearrowleft \mathcal{U}^\pi(e)$ loops indefinitely without writing to the output ($= \varepsilon$).

Proof. We define the set $V = \{\langle M_{y,i} \rangle \mid y \in \mathbb{B}^*, i \in \mathcal{N}\}$ such that for all $i \in \mathcal{N}$ and $y \in \mathbb{B}^*$, we have:
$$\forall p \in \mathbb{B}^*, \quad M_{y,i}(p) \equiv \begin{cases} M_i(y, p) & \text{If } y \neq \varepsilon \\ M_i(p) & \text{Otherwise} \end{cases}$$
Let us then set $\Lambda := \{\widehat{y\,i} \mid y \in \mathbb{B}^*, i \in \mathcal{N}\}$ and define the computable function
$$\alpha : \widehat{y\,i} \in \Lambda \mapsto \langle M_{y,i} \rangle \in V$$
Let us set the prefix encoding $I : \mathcal{N} \mapsto \Lambda$, which exists because Λ is prefix-free and semi-decidable (I is in fact a bijective enumeration of Λ). Thus we can set the enumeration π such that
$$\pi_V : j \in \mathcal{N} \mapsto \alpha(I(j)) \in V$$
Let us justify that π_V is a compact enumeration. Let M be a Turing machine. Since π is a compact enumeration there exists $i \in \mathcal{N}$ such that $M \equiv M_i$. Thus by definition we have $M_{\varepsilon,i} \equiv M_i$ which implies by transitivity $M_{\varepsilon,i} \equiv M$. We therefore indeed have that π_V is also a compact enumeration.

Let us then define the universal Turing machine $\mathcal{U}^\pi$ associated with I and π_V. Let us verify that it corresponds to the points of its definition

1. If $e = I(j)p \in \{I(j)p \mid j \in \mathcal{N}, p \in \mathbb{B}^*\}$ then there exists $\widehat{y\,i} \in \Lambda$ such that $I(j) = \widehat{y\,i}$, which allows us to write $e = \langle y, i, p \rangle$ and $\pi_V(j) = \alpha(I(j)) = \langle M_{y,i} \rangle$. We then verify
$$\mathcal{U}^\pi(y, i, p) \equiv \mathcal{U}^\pi(I(j)p) \equiv M_{y,i}(p) \equiv \begin{cases} M_i(y, p) & \text{If } y \neq \varepsilon \\ M_i(p) & \text{Otherwise} \end{cases}$$
2. Otherwise $\circlearrowleft \mathcal{U}^\pi(e)$ loops indefinitely without writing to the output.

■

1.8 Prefix Turing Machine

In this section, we will introduce a particular type of Turing machine: prefix-free Turing machines. These Turing machines will specifically allow us to define, in another chapter, the notion of semi-discrete computable measure.

1.8.1 Definition

Definition: Prefix Turing machine

A Turing machine

$$M_{pf} = (Q, \mathbb{B}, \Gamma, \#, \delta, q_0, q_a)$$

is called a prefix Turing machine if $L_\downarrow(M_{pf})$ is prefix-free. In other words, this means that

$$\forall p, q \in \mathbb{B}^*, \quad \downarrow M_{pf}(p) \text{ and } \downarrow M_{pf}(q) \text{ and } p \leq_p q \implies p = q$$

Remark. We note the slight abuse of language. The name prefix Turing machine can be confusing, suggesting that the set of inputs must be prefixes, whereas they must on the contrary be prefix-free. ◇

In the same vein, we can define the notion of a prefix-free computable function.

Definition: Prefix partially computable function

A partially computable function ϕ is said to be prefix if $dom(\phi)$ is prefix-free.

1.8.2 Self-delimiting decoding associated with M_{pf}

Definition-Property: Encoding associated with a prefix machine

Let M_{pf} be a prefix Turing machine. There exists a bijective prefix encoding

$$E_{M_{pf}} : \{1, 2, \ldots, |L_\downarrow(M_{pf})|\} \mapsto L_\downarrow(M_{pf})$$

called the encoding associated with M_{pf}.

Proof. Let M_{pf} be a prefix Turing machine. Let us define the function $E_{M_{pf}}$ taking inputs in $i \in \mathcal{N}$ having the behavior

> Call M_{pf} in dovetail on $\sigma_0, \sigma_1, \ldots$ arranged in lexicographical order. Write the i-th word accepted in the dovetail call, denoted $\sigma_{\downarrow,i}$, to the output and accept.

Let us prove that $E_{M_{pf}}$ is a prefix encoding:

Prefix property: Consider i, j in $\{1, 2, \ldots, |L_\downarrow(M_{pf})|\}$ and suppose $E_{M_{pf}}(i) \leq_p E_{M_{pf}}(j)$. This means that $\sigma_{\downarrow,i} \leq_p \sigma_{\downarrow,j}$ or equivalently $M_{pf}(\sigma_{\downarrow,i}) \leq_p M_{pf}(\sigma_{\downarrow,j})$. By definition of a prefix machine $\sigma_{\downarrow,i} = \sigma_{\downarrow,j}$. This implies $i = j$ and therefore $E_{M_{pf}}(i) = E_{M_{pf}}(j)$.

Injectivity: Let i, j in $\{1, 2, \ldots, |L_\downarrow(M_{pf})|\}$ be such that $E_{M_{pf}}(i) = E_{M_{pf}}(j)$. Then in particular $E_{M_{pf}}(i) \leq_p E_{M_{pf}}(j)$, thus by the prefix property $\sigma_{\downarrow,i} = \sigma_{\downarrow,j}$, so $i = j$.

Surjectivity: Suppose $\sigma \in L_\downarrow(M_{pf})$. Then there exists a j such that $\sigma_j = \sigma$. Thus in the dovetail call σ_j will eventually be accepted, that is $\downarrow M_{pf}(\sigma)$. Suppose that σ is the i-th word accepted in the dovetail call. Then $E_{M_{pf}}(i) = \sigma$, hence the surjectivity. ■

Starting from a prefix Turing machine, one can construct a self-delimiting function allowing one to automatically identify and extract, within a string composed of symbols, the portion that constitutes a program acceptable by the machine. The process is as follows: for a string $x = b_1 b_2 b_3 \ldots$, we call the machine in dovetail on each of its prefixes $(b_1, b_1 b_2, b_1 b_2 b_3, \ldots)$.

The prefix property of the machine excludes the possibility that two prefixes are accepted. This thus allows one to unambiguously delimit the prefix of the string corresponding to a recognized program. One can also more directly use a self-delimiting encoding associated with this prefix Turing machine.

Definition-Theorem: Self-delimiting decoder associated with a prefix machine

For any prefix Turing machine M_{pf} there exists a computable function $D_{M_{pf}}$, called self-delimiting decoder associated with M_{pf}, such that

$$D_{M_{pf}} : \begin{cases} ez & \mapsto e \\ L_{\downarrow}(M_{pf})\,\mathbb{B}^* & \mapsto L_{\downarrow}(M_{pf}) \end{cases} \quad \text{and} \quad D^*_{M_{pf}} : \begin{cases} ez & \mapsto M_{pf}(e) \\ L_{\downarrow}(M_{pf})\,\mathbb{B}^* & \mapsto \mathbb{B}^* \end{cases}$$

Proof. Let $E_{M_{pf}}$ be the encoding associated with M_{pf}. It suffices to remark that $D_{E_{M_{pf}}} = D_{M_{pf}}$. We then also have $D^*_{E_{M_{pf}}} = D^*_{M_{pf}}$. ■

We notice that with the help of a self-delimiting decoder, it is possible to successively extract the words e_i from a string $x = e_1e_2\dots e_l$. We begin by extracting the first word e_1 using $D_{M_{pf}}(x)$, then we remove e_1 from x to obtain a residual string $x' = e_2e_3\dots e_l$. It then suffices to repeat the process on x' until obtaining e_i.

Corollary: Extraction by a self-delimiting decoder

Let $k \geq 1$ be an integer and M_{pf} be a prefix Turing machine. There exists a computable function

$$D^{(k)}_{M_{pf}} : \begin{cases} e_1e_2\dots e_l \in & \mapsto e_k \\ L_{\downarrow}(M_{pf})^{\geq k} & \mapsto L_{\downarrow}(M_{pf}) \end{cases}$$

with the integer $l \geq k$ and $L_{\downarrow}(M_{pf})^{\geq k}$ the words [composed of] at least k concatenations of $L_{\downarrow}(M_{pf})$.

Proof. Let us define the computable function $D^{(k)}_{M_{pf}}$ having the following behavior for an input $e = e_1e_2\dots e_l$ in $L_{\downarrow}(M_{pf})^+$:

Step 1. Initialize $i \leftarrow 1$ and $r_1 \leftarrow e$

Step 2. Assign in sequence
- $a_i \leftarrow D_{M_{pf}}(r_i)$
- $r_{i+1} \leftarrow r_i[|a_i| : |r_i|]$, that is r_{i+1} stripped of a_i.

Step 3. If $i == k$ then
- Write a_k to output then accept the input.

Else
- $i \leftarrow i + 1$ then Goto Step 2.

Let an input be $e = e_1e_2\dots e_m \in L_{\downarrow}(M_{pf})^+$, with $m \geq k$. Consider for this input the loop invariant $\mathcal{I}(i)$ for $1 \leq i \leq k$:

$$\mathcal{I}(i) : \text{"At the beginning of step 2, the variable } r_i = e_ie_{i+1}\dots e_m\text{."}$$

Base (i = 1) : By step 1 we have $r_1 = e$, so the invariant is true.

Maintenance (i → i+1): The invariant is true for $i = 1$ by initialization ($r_1 \leftarrow w$). Suppose that $\mathcal{I}(i)$ is true. The algorithm calculates $a_i \leftarrow D_{M_{pf}}(r_i)$. Since $L_{\downarrow}(M_{pf})$ is a prefix

code and r_i starts with c_i, we necessarily have $a_i = c_i$. The update $r_{i+1} \leftarrow r_i[|a_i| :]$ then gives $r_{i+1} = e_{i+1} \dots e_m$, which establishes $\mathcal{I}(i+1)$. The invariant is therefore true for all $i \leq k$.

Termination: For $D^{(k)}_{M_{pf}}(e_1 e_2 \dots e_m)$ with $k \leq m$, the loop executes exactly k times then terminates. During the k-th iteration, the invariant $\mathcal{I}(k)$ ensures that $r_k = e_k \dots e_m$. $D^{(k)}_{M_{pf}}$ calculates and then returns $a_k = D_{M_{pf}}(r_k) = e_k$, which is the expected result. Moreover, if an input is not in at least k concatenations of $L_\downarrow(M_{pf})$, then necessarily there exists an $i \leq k$ where the assignment $a_i \leftarrow D_{M_{pf}}(r_i)$ will loop indefinitely. Thus it is necessary that the input be in $L_\downarrow(M_{pf})^{\geq k}$. ■

1.8.3 Mapping from $\langle\, \mathcal{T} \,\rangle$ to $\langle\, \mathcal{T}_{pf} \,\rangle$

We will now demonstrate a lemma transforming a given Turing machine into a prefix Turing machine. This "mapping" lemma will be useful to us for the construction of an enumeration of prefix-free Turing machines.

Lemma: Mapping $\langle\, M \,\rangle \xrightarrow{\langle \mathcal{T} \rangle \Rightarrow \langle \mathcal{T}_{pf} \rangle} \langle\, M' \,\rangle$

Let M, then there exists [a] M_{pf} such that

1. M_{pf} is prefix-free
2. $L_\downarrow(M_{pf}) \subset L_\downarrow(M)$
3. If M is prefix-free then

$$\forall e \in \mathbb{B}^*, \quad M(e) \equiv M_{pf}(e)$$

[a] constructively, that is to say, we will exhibit this Turing machine

Proof. Let M be a Turing machine. We construct M_{pf} having the following behavior:

Input : $e = b_1 b_2 \dots b_l \in \mathbb{B}^*$

Step 1. Initialize $p \leftarrow \varepsilon$

Step 2. M is called in dovetail on the words pq_i for $q_0, q_1, q_2, q_3 \dots$ in lexicographical order. For each computation $\downarrow M(pq_j)$ that halts, go to step 3.

Step 3. If $q_j == \varepsilon$:

- If $p == e$:
  - $\downarrow M_{pf}(e)$ with $M(e)$ as output [†]
- Otherwise :
  - $\circlearrowleft M_{pf}(e)$ [¬†]

Otherwise :

- If $|p| < l$:
  - $p \leftarrow b_1 \dots b_{|p|+1}$ and Goto Step 2 [↯]
- Otherwise :
  - $\circlearrowleft M_{pf}(e)$ [¬↯]

Let us show that M_{pf} verifies the points of the lemma:

1. Let us prove that $L_\downarrow(M_{pf})$ is prefix-free by supposing, for the sake of contradiction, the existence of $e = b_1 \dots b_l$ and $e' = b_1 \dots b_l b_{l+1} \dots b_{l+l'}$ such that $\downarrow M_{pf}(e)$ and $\downarrow M_{pf}(e')$. For the execution of $M_{pf}(e')$:

   At the l-th pass through line $\natural$, we will have $p = b_1 \dots b_l$, hence $p = e$. Since $p \in L_\downarrow(M_{pf})$, we have, for the $l+1$-th pass into step 3, that $q_j == \varepsilon$ and $p \neq e'$, hence the execution of $\neg\dagger$, that is $\circlearrowleft M_{pf}(e')$. We therefore have $\circlearrowleft M_{pf}(e')$ and $\downarrow M_{pf}(e')$, which is absurd.

2. Let $e \in L_\downarrow(M_{pf})$, necessarily line [†] was executed; hence, by the "If" conditions, we have $q_j == \varepsilon$ and $p == e$, therefore $e = pq_j$. However, to access step 3, it is necessary that $\downarrow M(pq_j)$, hence $e \in L_\downarrow(M)$, which shows (2).

3. Suppose M is prefix-free. Let $e = b_1 b_2 \dots b_l$ such that $\downarrow M(e)$. During the execution of $M_{pf}(e)$:

   Since M is assumed to be prefix[-free], then $\forall k < l, \downarrow M(b_1 b_2 \dots b_k q_j)$ implies $q_j \neq \epsilon$. It is therefore impossible to pass through line † for $k < l$. If $k = l$ then $\downarrow M(pq_j)$ with $p = e$ and $q_j = \varepsilon$, hence the passage through †, therefore $\downarrow M_{pf}(e)$ with $M(e)$ as output, that is $M_{pf}(e) = M(e)$. We therefore have $L_\downarrow(M) \subset L_\downarrow(M_{pf})$, which with (2) gives $L_\downarrow(M) = L_\downarrow(M_{pf})$.

■

1.8.4 Enumeration of prefix Turing machines

The previous mapping allows us to associate a prefix-free Turing machine with every Turing machine, thus enabling us to construct an enumeration in the sense of $\equiv$.

Definition-Theorem: Compact enumeration of prefix Turing machines

There exists an enumeration $\pi_{pf} : i \in \mathcal{N} \mapsto \langle\, M_{pf,i} \,\rangle \in \langle\, \mathcal{T}_{pf} \,\rangle$ called a compact enumeration of prefix Turing machines, satisfying

$$\forall M_{pf}, \quad \exists i \in \mathcal{N}, \quad \pi_{pf}(i) = \langle\, M_{pf,i} \,\rangle \text{ and } M_{pf,i} \equiv M_{pf}$$

Proof. Let π be an enumeration of $\langle\, \mathcal{T} \,\rangle$. Let us define π_{pf} as a computable function with the following behavior:

For an input $i \in \mathcal{N}$:

- Assign $\langle\, M_i \,\rangle \leftarrow \pi(i)$.
- Apply the Mapping $\langle\, M_i \,\rangle \xrightarrow{\langle \mathcal{T} \rangle \Rightarrow \langle \mathcal{T}_{pf} \rangle} \langle\, M_{pf,i} \,\rangle$.
- Accept the input with $\langle\, M_{pf,i} \,\rangle$ as output.

Let us now prove that π_{pf} is a compact enumeration. Let M_{pf} be a prefix Turing machine. By surjectivity of π, there exists $i \in \mathcal{N}$ such that $\pi(i) = \langle\, M_{pf} \,\rangle$. Thus, according to the Mapping lemma, $\pi_{pf}(i) = \langle\, M_{pf,i} \,\rangle$ is prefix[-free] and satisfies $M_{pf,i} \equiv M_{pf}$. ■

Remark. We note that π_{pf} is not an enumeration of the set of encodings of prefix Turing machines. Indeed, there exist prefix Turing machines M_{pf} such that $\langle\, M_{pf} \,\rangle$ is not part of the

compact enumeration π_{pf}. However, what the previous theorem states is that there necessarily exists an $i \in \mathcal{N}$ such that $\pi(i) = \langle\, M_{pf} \,\rangle$. $\diamond$

Definition: Enumeration of prefix partially computable functions

We call an enumeration of prefix partially computable functions a compact enumeration π_{pf} of the set of $\langle\, \mathcal{T}_{pf} \,\rangle$. We will then denote

$$\pi_{pf} : i \in \mathcal{N} \mapsto \langle\, \phi_{pf,i} \,\rangle \in \langle\, \mathcal{T}_{pf} \,\rangle$$

with the convention that $\langle\, \phi_{pf,i} \,\rangle$ is the code of a prefix Turing machine computing the prefix partially computable function $\phi_{pf,i}$.

1.8.5 Existence of a universal / enumerative / auxiliary prefix Turing machine

We will now introduce the notion of Universal Prefix Turing Machine. This is very largely similar to that of the universal Turing machine with two exceptions. The first is

Definition-Theorem: Existence of a universal prefix Turing machine

There exists a prefix Turing machine called Universal Prefix Turing Machine U associated with

- $I : i \in \mathcal{N} \mapsto \sigma \in \mathbb{B}^*$ a prefix encoding.
- $\pi_{pf} : i \in \mathcal{N} \mapsto M_i \in \langle\, \mathcal{T}_{pf} \,\rangle$ a compact enumeration of prefix Turing machines.

verifying the following points

1. If $e = \sigma\, p \in \{I(i)p \mid i \in \mathcal{N}, p \in \mathbb{B}^*\}$ then
$$U(\sigma\, p) \underset{h}{\equiv} M_i(p)$$
2. Otherwise $\circlearrowleft U(e)$ loops indefinitely without writing to the output, that is ε.

Proof. Below we prove the existence of an enumerative and auxiliary universal prefix Turing machine. These two types of Turing machines are also universal prefix Turing machines, hence the proof of existence. ■

Definition-Theorem: Existence of an enumerative universal Turing machine

Let $\pi_{pf} : i \in \mathcal{N} \mapsto \langle\, M_{pf,i} \,\rangle \in \langle\, \mathcal{T}_{pf} \,\rangle$ be a compact enumeration of prefix Turing machines, denoted π to simplify the notation. There exists a universal prefix Turing machine U^{π} called enumerative universal prefix Turing machine associated with π such that

1. If $e = \langle\, i, p \,\rangle \in \langle\, \mathcal{N}, \mathbb{B}^* \,\rangle$ then
$$U^{\pi}(i, p) \underset{h}{\equiv} M_{pf,i}(p)$$

2. Otherwise $\circlearrowleft U^\pi(e)$ loops indefinitely without writing to the output, that is ε.

Proof. It suffices to define the enumerative universal Turing machine U^π for π_{pf} (denoted π). We then have by definition of an enumerative universal Turing machine that

1. If $e = \langle\, i, p \,\rangle \in \langle\, \mathcal{N}, \mathbb{B}^* \,\rangle$ then
$$U^\pi(i, p) \underset{h}{\equiv} M_{pf,i}(p)$$
2. Otherwise $\circlearrowleft U(e)$ loops indefinitely without writing to the output, that is ε.

Which shows that U is a universal prefix Turing machine for the prefix encoding $I : i \in \mathcal{N} \mapsto \widehat{i}$ and the enumeration π.

It remains for us to prove that U^π is a prefix Turing machine. We verify that if the input is not in $\langle\, \mathcal{N}, \mathbb{B}^* \,\rangle$ then U loops indefinitely, which gives $L_\downarrow(U^\pi) \subset \langle\, \mathcal{N}, \mathbb{B}^* \,\rangle$. Consider then $e = \langle\, i, p \,\rangle$ and $e' = \langle\, i', p' \,\rangle$ both in $L_\downarrow(M_{pf,i})$ such that $e \leq_p e'$. By non-ambiguity we necessarily have $i = i'$ and $p \leq_p p'$. Now, by definition of a prefix Turing machine $M_{pf,i}(p) \leq_p M_{pf,i'}(p')$. Which, with point 1 of a universal Turing machine that we have just proved, indeed shows that U is a prefix Turing machine. ■

Definition-Theorem: ∃ Auxiliary enumerative universal Turing machine

Let $\pi_{pf} : i \in \mathcal{N} \mapsto \langle\, M_i \,\rangle \in \langle\, \mathcal{T} \,\rangle$ be a compact enumeration of prefix Turing machines. There exists a universal prefix Turing machine U^π called auxiliary enumerative universal prefix Turing machine associated with π, such that

1. If $e = \langle\, y, i, p \,\rangle \in \langle\, \mathbb{B}^*, \mathcal{N}, \mathbb{B}^* \,\rangle$ and
$$U^\pi(y, i, p) \equiv \begin{cases} M_i(y, p) & \text{If } y \neq \varepsilon \\ M_i(p) & \text{Otherwise} \end{cases}$$
2. Otherwise $\circlearrowleft U^\pi(e)$ loops indefinitely without writing to the output ($= \varepsilon$).

Proof. The proof is similar to the previous case. It suffices to define the auxiliary enumerative universal Turing machine U^π for π_{pf} (denoted π). We then have by definition of an enumerative universal Turing machine that

1. If $e = \langle\, y, i, p \,\rangle \in \langle\, \mathbb{B}^*, \mathcal{N}, \mathbb{B}^* \,\rangle$ and
$$U^\pi(y, i, p) \equiv \begin{cases} M_i(y, p) & \text{If } y \neq \varepsilon \\ M_i(p) & \text{Otherwise} \end{cases}$$
2. Otherwise $\circlearrowleft U^\pi(e)$ loops indefinitely without writing to the output ($= \varepsilon$).

[we must add the justification that it is a universal prefix Turing machine] It remains for us to show that U^π is a prefix Turing machine. We verify that if the input is not in $\langle\, \mathbb{B}^*, \mathcal{N}, \mathbb{B}^* \,\rangle$ then U loops indefinitely, which gives $L_\downarrow(U^\pi) \subset \langle\, \mathbb{B}^*, \mathcal{N}, \mathbb{B}^* \,\rangle$. Consider then $e = \langle\, y, i, p \,\rangle$ and $e' = \langle\, y, i', p' \,\rangle$ both in $L_\downarrow(M_{pf,i})$ such that $e \leq_p e'$. By non-ambiguity we necessarily have $y = y'$ and $i = i'$ and $p \leq_p p'$. Now, by definition of a prefix Turing machine $M_{pf,i}(y, p) \leq_p M_{pf,i'}(y, p')$. Which, with point 1 of a universal Turing machine that we have just proved, indeed shows that U^π is a prefix Turing machine. ■

1.9 Monotone Turing Machine

We will now introduce the notion of monotone Turing machine. The fundamental principle of this machine is that it cannot erase its output and can only append words to the end of its output tape. Finally, we will establish the existence of a universal monotone Turing machine, which we will define precisely.

1.9.1 Definition

Definition-Property: Monotone Turing Machine

A monotone Turing machine is a Turing machine with input in $\mathbb{B}^\infty$

$$M_{mt} = (Q, \mathbb{B}, \Gamma, \#, \delta, q_0, q_a)$$

such that we have

1. **Output monotonicity:** M_{mt} has prefix output, i.e., for all p in $\mathbb{B}^*$, we have

$$\forall t, t' \in \mathbb{N} : \quad t \leq t' \implies M_{mt}(p \mid t) \leq_p M_{mt}(p \mid t')$$

Thus for every word p in $\mathbb{B}^*$ there exists (see proof) a unique s in $\mathbb{B}^\infty$ such that

$$\forall t \in \mathbb{N}, \quad M(p \mid t) \leq_p s \quad \text{and} \quad \lim_{t \to \infty} |M(p \mid t)| = |s|$$

We then define [a]

$$M_{mt}(p) \underset{def.}{=} s$$

2. **Input monotonicity:** For all p, q in $\mathbb{B}^*$, we have

$$p \leq_p q \implies M_{mt}(p) \leq_p M_{mt}(q)$$

[a] even if M_{mt} loops on the input p

Proof. It suffices to remark that the sequence $(M_{mt}(p \mid t))_{t \in \mathbb{N}}$ in $\mathbb{B}^*$ is increasing with respect to $\leq_p$. According to the convergence theorem for monotone sequences in the sense of $\leq_p$, this implies that there exists a unique word $s \in \mathbb{B}^\infty$ such that $\lim_{t \to \infty} |M(p \mid t)| = |s|$ and $\forall t, M(p \mid t) \leq_p s$. ■

Remark. We can establish the notion of a monotone partially computable function. This is done in certain books. However, we will not use this notion and therefore we will not do so here. ◇

We establish the following convention: it is equivalent to considering that a Turing machine enters an empty loop once it reaches its accepting state. This translates to the following convention:

Convention:

For a Turing machine M, we set the following convention: if there exists a $t_\downarrow \in \mathbb{N}^*$ such

that $\downarrow M(p \mid t_\downarrow)$, that is to say that the Turing machine accepts p in $t_\downarrow$ transitions, then

$$\forall t > t_\downarrow, \quad M(p \mid t) \underset{def.}{=} M(p \mid t_\downarrow).$$

1.9.2 Mapping M to M_{mt}

Algorithm: Mapping $M \xrightarrow{\mathcal{T} \Rightarrow \mathcal{T}_{mt}} M_{mt}$

Recall that a language is said to be compatible if all its elements are pairwise compatible. Let M be any Turing machine. We then define the following Turing machine M_{mt}:

Input : $p = p_1 p_2 \dots p_l \in \mathbb{B}^*$, (and let us set $p_0 = \varepsilon$ and $p_i = \varepsilon$ for $i > l$)

Loop indefinitely for $k = 0, 1, 2, \dots$:

- Assign for all $w \in \mathbb{B}^{=k}$

$$I_w^k \leftarrow \{M(z \mid t) \mid z \leq_p w,\, t \leq k\} \qquad [\natural]$$

- Verify that I_w^k is compatible for each $w \in \mathbb{B}^{=k}$. If this is not the case, then loop indefinitely. $[\flat]$
- Assign the longest string in $I_{p_{1:k}}^k$, that is

$$\alpha_p^k \leftarrow argmax_{\alpha \in I_{p_{1:k}}^k} \{|\alpha|\} \qquad [\dagger]$$

Then write α_p^k to the output (without stopping).

The proof of the Mapping lemma relies on certain properties regarding the set of candidate words I_w^k and the chosen candidate α_p^k, which furthermore corresponds to the output of M_{mt}. Let us perform this preliminary analysis.

Lemma: Analysis of the Mapping

Let $p, p' \in \mathbb{B}^*$ be such that $p \leq_p p'$. Let $k, k' \in \mathcal{N}$ satisfying $k \leq k'$. If the output $\alpha_{k'}^{p'}$ is assigned (that is, if I_w^k is compatible during $\flat$ at iteration k'), then:

1. $I_{p_{0:k}}^k \subseteq I_{p'_{0:k'}}^{k'}$
2. $\alpha_p^k \leq_p \alpha_{p'}^{k'}$

Proof. 1) Consider an $s \in I_{p_{0:k}}^k$. By definition, there exist $z \in \mathbb{B}^*$ and $t \in \mathcal{N}$ such that $s = M(z \mid t)$, with $z \leq_p p_{0:k}$ and $t \leq k$. Knowing that $p \leq_p p'$ and $k \leq k'$, we have respectively $p_{0:k} \leq_p p'_{0:k'}$ and $t \leq k'$. By transitivity of the prefix relation, $z \leq_p p_{0:k}$ and $p_{0:k} \leq_p p'_{0:k'}$, we obtain $z \leq_p p'_{0:k'}$. Thus, $s = M(z \mid t)$ satisfies the conditions to belong to $I_{p'_{0:k'}}^{k'}$. The inclusion is therefore proved.

2) As we have just shown that $I_{p_{0:k}}^k \subseteq I_{p'_{0:k'}}^{k'}$, we deduce that α_p^k, being an element of $I_{p_{0:k}}^k$, is an element of $I_{p'_{0:k'}}^{k'}$. Since the output $\alpha_{p'}^{k'}$ is defined, the set $I_{p'_{0:k'}}^{k'}$ is compatible. In such a set, every element is a prefix of the element of greatest length. Consequently, α_p^k is a prefix

of $\alpha_{p'}^{k'}$, which establishes the result. ■

Lemma: Mapping M to M_{mt}

Let M, then there exists M_{mt} such that

1. M_{mt} is monotone
2. If M is monotone then

$$\forall p \in \mathbb{B}^*, \quad M_{mt}(p) = M(p)$$

Proof. 1) Output monotonicity: Let $p \in \mathbb{B}^*$. By setting $p = p'$ in Lemma 1, we obtain that for all $k \leq k'$, $\alpha_k^p \leq_p \alpha_{k'}^p$. The output history $(\alpha_k^p)_k$ is therefore increasing in the sense of $\leq_p$.

Input monotonicity: Let $p, p' \in \mathbb{B}^*$ be such that $p \leq_p p'$. By setting $k = k'$ in the previous lemma, we obtain $\alpha_k^p \leq_p \alpha_k^{p'}$ for all k. By passing to the limit:

$$M_{mt}(p) = \lim_{k\to\infty} \alpha_k^p \leq_p \lim_{k\to\infty} \alpha_k^{p'} = M_{mt}(p')$$

M_{mt} is therefore indeed a monotone Turing machine.

2) Suppose that M is a monotone Turing machine.

We remark first that the input and output monotonicity of M implies that each element $M(z|t)$, for $z \leq_p w$ and $t \leq k$, is a prefix of $M(w)$. The set $I_w^k := \{M(z \mid t) \mid z \leq_p w, t \leq k\}$ is therefore always compatible, and the algorithm never loops indefinitely.

We can now show that $\lim_{k\to\infty} \alpha_k^p = M(p)$: Let $p \in \mathbb{B}^*$ and the set of its prefixes, denoted Z_p. For each $z \in Z_p$, by output monotonicity $\lim_{k\to\infty} M(z|k) = M(z)$. By definition of a limit, there exists therefore an integer k_m such that for all $k \geq k_m$ and all $z \in Z_p$, we have $M(z|k) = M(z)$. Consider $k \geq \max(k_m, |p|)$. Since $k \geq |p|$ we already have that $p = p_{1:k}$. Moreover, the output α_k^p is the element of greatest length in $I_p^k = \{M(z|t) \mid z \in Z_p, t \leq k\}$. Now for a $z \in Z_p$ and $t \leq k$ we have:

- By the output monotonicity of M, every $M(z|t)$ is a prefix of $M(z|k)$.
- For $k \geq k_m$, we have $M(z|k) = M(z)$.
- By the input monotonicity of M, every $M(z)$ is a prefix of $M(p)$.

We therefore have by transitivity that every element of I_p^k is a prefix of $M(p)$. The element $M(p)$ itself (obtained for $z = p, t = k$) is in the set of output candidates I_p^k. It is therefore necessarily the element of greatest length. Thus, for all $k \geq \max(k_m, |p|)$, we have $\alpha_k^p = M(p)$. This yields $M(p) = \lim_{k\to\infty} \alpha_k^p$.

Now, by construction, $(\alpha_k^p)_{k\in\mathcal{N}}$ represents the history of M_{mt} for the input p. We therefore also have $M_{mt}(p) = \lim_{k\to\infty} \alpha_p^k$. We can therefore conclude $M_{mt}(p) = M(p)$. ■

1.9.3 Enumeration of monotone Turing machines

Definition-Theorem: Compact enumeration of monotone Turing machines

There exists an enumeration $\pi_{mt} : i \in \mathcal{N} \mapsto \langle\, M_{mt,i} \,\rangle \in \langle\, \mathcal{T}_{mt} \,\rangle$ called a compact enumeration of monotone Turing machines, satisfying

$$\forall M_{pf}, \quad \exists i \in \mathcal{N}, \quad \pi_{mt}(i) = \langle\, M_{mt,i} \,\rangle \text{ and } M_{mt,i} \equiv M_{pf}$$

Proof. Let π be an enumeration of $\langle\, \mathcal{T} \,\rangle$. Let us define π_{mt} as a computable function with the following behavior:

For an input $i \in \mathcal{N}$:

- Assign $\langle\, M_i \,\rangle \leftarrow \pi(i)$.
- Apply the Mapping $\langle\, M_i \,\rangle \xrightarrow{\langle \mathcal{T} \rangle \Rightarrow \langle \mathcal{T}_{mt} \rangle} \langle\, M_{mt,i} \,\rangle$.
- Accept the input with $\langle\, M_{mt,i} \,\rangle$ as output.

Let us prove then that π_{mt} is a compact enumeration of $\langle\, \mathcal{T}_{mt} \,\rangle$. Let M_{mt} be a monotone Turing machine. By surjectivity of π, there exists $i \in \mathcal{N}$ such that $\pi(i) = \langle\, M_{mt} \,\rangle$. Thus according to the Mapping, $\langle\, M_{mt,i} \,\rangle$ is prefix and satisfies $M \equiv M_{mt}$. ■

1.9.4 Existence of a universal monotone Turing machine

Definition-Theorem: Existence of a universal monotone Turing machine

There exists a monotone Turing machine called Universal Monotone Turing Machine $\mathcal{U}_{mt}$ associated with

- $I : i \in \mathcal{N} \mapsto \sigma \in \mathbb{B}^*$ a prefix encoding.
- $\pi_{mt} : i \in \mathcal{N} \mapsto M_i \in \langle\, \mathcal{T}_{mt} \,\rangle$ a compact enumeration of monotone Turing machines.

verifying the following points

1. If $e = \sigma\, p \in \{I(i)p \mid i \in \mathcal{N}, p \in \mathbb{B}^*\}$ then
$$\mathcal{U}_{mt}(\sigma\, p) \underset{h}{\equiv} M_i(p)$$
2. Otherwise $\circlearrowleft\, \mathcal{U}_{mt}(e)$ loops indefinitely without writing to the output, that is ε.

Proof. Below we prove the existence of an enumerative universal monotone Turing machine. This Turing machine is also a universal monotone Turing machine, hence the proof of existence. ■

Definition-Theorem: Existence of an enumerative universal Turing machine

Let $\pi_{mt} : i \in \mathcal{N} \mapsto \langle\, M_{mt,i} \,\rangle \in \langle\, \mathcal{T}_{mt} \,\rangle$ be a compact enumeration of monotone Turing machines, denoted π to simplify the notation. There exists a universal monotone Turing machine $\mathcal{U}^{\pi}_{mt}$ called enumerative universal monotone Turing machine associated with π

such that

1. If $e = \langle i, p \rangle \in \langle \mathcal{N}, \mathbb{B}^* \rangle$ then
$$\mathcal{U}^{\pi}_{mt}(i, p) \underset{h}{\equiv} M_{mt,i}(p)$$
2. Otherwise $\circlearrowleft \mathcal{U}^{\pi}_{mt}(e)$ loops indefinitely without writing to the output, that is ε.

Proof. It suffices to define the enumerative universal Turing machine $\mathcal{U}^{\pi}_{mt}$ for π_{mt} (denoted π). We then have by definition of an enumerative universal Turing machine that

1. If $e = \langle i, p \rangle \in \langle \mathcal{N}, \mathbb{B}^* \rangle$ then
$$\mathcal{U}^{\pi}_{mt}(i, p) \underset{h}{\equiv} M_{mt,i}(p)$$
2. Otherwise $\circlearrowleft \mathcal{U}_{mt}(e)$ loops indefinitely without writing to the output, that is ε.

Which shows that $\mathcal{U}_{mt}$ is a universal monotone Turing machine for the prefix encoding $I : i \in \mathcal{N} \mapsto \widehat{i}$ and the enumeration π. It remains for us to prove that $\mathcal{U}^{\pi}_{mt}$ is a monotone Turing machine:

- Output monotonicity: If the input is not in $\langle \mathcal{N}, \mathbb{B}^* \rangle$ then the machine loops indefinitely without writing to the output. We therefore already have output monotonicity in this case. Suppose now that the input $\langle i, p \rangle$ is in $\langle \mathcal{N}, \mathbb{B}^* \rangle$. By definition of a universal Turing machine, we have $\mathcal{U}^{\pi}_{mt}(i, p) \underset{h}{\equiv} M_{mt,i}(p)$. Thus $\mathcal{U}^{\pi}_{mt}(i, p)$ has the same history as $M_{mt,i}(p)$ with $M_{mt,i}$ being a monotone Turing machine. We therefore also have output monotonicity in this case.

- Input monotonicity: Let e, e' be in $\mathbb{B}^*$ such that $e \leq_p e'$. If e or e' is in $\langle \mathcal{N}, \mathbb{B}^* \rangle$ then by non-ambiguity e, e' are both in $\langle \mathcal{N}, \mathbb{B}^* \rangle$. Thus we must verify the cases where e, e' are both in $\langle \mathcal{N}, \mathbb{B}^* \rangle$ or where neither are. Suppose $e = \langle i, p \rangle$ and $e' = \langle i', p' \rangle$ are in $\langle \mathcal{N}, \mathbb{B}^* \rangle$; then by non-ambiguity $i = i'$ and $p \leq_p p'$. By input monotonicity of $M_{mt,i}$ we obtain $M_{mt,i}(p) \leq_p M_{mt,i}(p')$. Which by point 1 gives $\mathcal{U}^{\pi}_{mt}(i, p) \leq_p \mathcal{U}^{\pi}_{mt}(i, p')$. Suppose now that e, e' are not in $\langle \mathcal{N}, \mathbb{B}^* \rangle$; then the output of $\mathcal{U}_{mt}$ remains empty for these two inputs, hence the input monotonicity in this case.

■

1.10 Semi-computable Functions

In the following chapter dedicated to Kolmogorov complexity, we will see that certain functions cannot be computed by a Turing machine. However, it is possible to approximate some of these functions using computable functions from below. This is why we introduce the notion of lower semi-computable functions.

1.10.1 Definition

Definition: Lower semi-computable functions

We say that $f : X \subset \mathbb{B}^* \mapsto \mathbb{R} \cup \{\infty\}$ is a lower semi-computable function if there exists a totally computable function $\phi_{lc} : \langle X, \mathcal{N} \rangle \mapsto \mathcal{Q}^+$ verifying, for all $x \in X$, we have

1. For all $k \in \mathcal{N}$ that $\phi_{lc}(x, k) \leq \phi_{lc}(x, k+1)$.
2. $\lim_{k\to\infty} \phi_{lc}(x, k) = f(x)$.

We also say that ϕ_{lc} is a lower approximator of f and we denote $\phi_{lc} \xrightarrow{lc} f$. Moreover, the set of lower semi-computable functions is denoted $\Pi_{\nearrow}$.

Remark. A few remarks:

- Several definitions of lower [semi-]computable functions exist in the literature. Sometimes the domain of f is not specified. Also sometimes ϕ_{lc} is partially computable, which can pose a problem for the limit, which requires that $\phi_{lc}(x, k)$ be defined for all k.
- We also denote in the following $\langle \mathcal{T}_{lc} \rangle$ the set of codes of Turing machines that compute a lower approximator.
- In general, we call a lower approximator any totally computable function $\phi_{lc} : \langle X, \mathcal{N} \rangle \mapsto \mathcal{Q}$ verifying points 1 and 2 for any $X \subset \mathbb{B}^*$.

$\diamond$

Definition: Upper semi-computable functions

We say that $f : X \subset \mathbb{B}^* \mapsto \mathbb{R} \cup \{-\infty\}$ is an upper semi-computable function if there exists a totally computable function $\phi_{sc} : \langle X, \mathcal{N} \rangle \mapsto \mathcal{Q}^+$ verifying, for all $x \in X$, we have

1. For all $k \in \mathcal{N}$ that $\phi_{sc}(x, k) \geq \phi_{sc}(x, k+1)$.
2. $\lim_{k\to\infty} \phi(x, k) = f(x)$.

We also say that ϕ_{sc} is an upper approximator of f and we denote $\phi_{sc} \xrightarrow{lc} f$.

1.10.2 Mapping $\langle \phi \rangle$ to $\langle \phi_{lc} \rangle$

Lemma: Mapping $\langle \phi \rangle \xrightarrow{\langle \mathcal{T} \rangle \Rightarrow \langle \mathcal{T}_{lc} \rangle} \langle \phi_{lc} \rangle$

Let ϕ be a partially computable function. There exists then a function ϕ_{lc} such that:

1. ϕ_{lc} is a lower approximator.
2. If ϕ is a lower approximator for a function f, then ϕ_{lc} is also a lower approximator of f.

Proof. Let ϕ be a partially computable function. Let us denote by $\phi(x,i)[k]$ the result of the computation, of a Turing machine that computes ϕ, on input $\langle\, x,i \,\rangle$ if it terminates in at most k steps, and undefined otherwise. We define ϕ_{lc} for any string x and integer $k \geq 0$ as follows:

$$\phi_{lc}(x,k) = \max\left(\{0\} \cup \{\phi(x,i)[k] \mid 0 \leq i \leq k \text{ and } \phi(x,i)[k] \text{ is defined}\}\right)$$

1) ϕ_{lc} is a lower approximator:

- Let us show that ϕ_{lc} is a totally computable function. For any input $\langle\, x,k \,\rangle$, the computation of ϕ_{lc} requires simulating a finite number of inputs $(k+1)$, each having a bounded number of steps (k). This process always terminates. ϕ_{lc} is therefore a total and computable function.
- Let us show that $\phi_{lc}(x,\cdot)$ is non-decreasing. The set of computations terminating in at most k steps is included in that of computations terminating in at most $k+1$ steps. Consequently, $\phi_{lc}(x,k) \leq \phi_{lc}(x,k+1)$ for all $k \in \mathcal{N}$.

2) Convergence. Suppose that ϕ is a lower approximator for a function f.

($\leq$) By construction, $\phi_{lc}(x,k)$ is the maximum of a finite set of values in the set $\{\phi(x,i) \mid i \in \mathcal{N}\}$. Since ϕ is an approximator for f, we have $\phi(x,i) \leq f(x)$ for all i. Thus, $\phi_{lc}(x,k) \leq f(x)$ for all k, and by passing to the limit, $\lim_{k\to\infty} \phi_{lc}(x,k) \leq f(x)$.

($\geq$) Let $i \in \mathcal{N}$. Since $\phi(x,\cdot)$ is a total function (because ϕ is an approximator), the computation of $\phi(x,i)$ terminates in a finite number of steps, say t_i. For all $k \geq \max(i,t_i)$, the value $\phi(x,i)$ is a candidate in the calculation of the maximum for $\phi_{lc}(x,k)$. Therefore, for sufficiently large k, $\phi_{lc}(x,k) \geq \phi(x,i)$. By taking the limit over k, it follows that $\lim_{k\to\infty} \phi_{lc}(x,k) \geq \phi(x,i)$. This inequality being true for all $i \in \mathcal{N}$, by passing to the limit:

$$\lim_{k\to\infty} \phi_{lc}(x,k) \geq \lim_{i\to\infty} \phi(x,i) = f(x)$$

The two inequalities imply that $\lim_{k\to\infty} \phi_{lc}(x,k) = f(x)$. ■

1.10.3 Compact enumeration of lower approximators

Definition-Theorem: Compact enumeration of lower approximators

We call a compact enumeration of lower computable approximators the totally computable function $\pi_{lc} : i \in \mathcal{N} \mapsto \langle\, \phi_{lc,i} \,\rangle \in \langle\, \mathcal{T}_{lc} \,\rangle$ such that

$$\forall f \in \Pi_{\nearrow}, \quad \exists i \in \mathcal{N}, \quad \pi_{lc}(i) = \langle\, \phi_{lc,i} \,\rangle \text{ and } \phi_{lc,i} \xrightarrow{lc} f$$

with the convention that $\langle\, \phi_{lc,i} \,\rangle$ is the code of a Turing machine in $\langle\, \mathcal{T}_{lc} \,\rangle$ that computes the function $\phi_{lc,i}$.

Proof. Let π be an effective enumeration of $\langle\, \mathcal{T} \,\rangle$. Let us define π_{lc} as a computable function with the following behavior:

For an input $i \in \mathcal{N}$:

– Assign $\langle\, M_i \,\rangle \leftarrow \pi(i)$.

- Apply the Mapping $\langle M_i \rangle \xrightarrow{\langle \mathcal{T} \rangle \Rightarrow \langle \mathcal{T}_{lc} \rangle} \langle \phi_{lc,i} \rangle$.
- Accept the input with $\langle M_{lc,i} \rangle$ as output.

Let us prove that π_{lc} is a compact enumeration of $\Pi_{\nearrow}$. Let ϕ_{lc} be a lower approximator of a function f. By definition of π, there exists $i \in \mathcal{N}$ such that $\pi(i) = \langle \phi_{lc} \rangle$. Thus according to the Mapping, $\langle \phi_{lc,i} \rangle$ is the code of a lower approximator which verifies $\phi_{lc,i} \xrightarrow{l.c} f$. ■

1.10.4 Variant: strict lower approximator

Definition: Strict lower approximator

Let $f : X \subseteq \mathbb{B}^* \to \mathbb{R} \cup \{\infty\}$ be a function. We say that a function $\phi_{lc}^{<} : X \times \mathcal{N} \to \mathcal{Q}$ is a strict lower approximator for f if:

1. $\phi_{lc}^{<}$ is a lower approximator for f.
2. For all $x \in X$ and all $k \in \mathcal{N}$, the inequality is strict: $\phi_{lc}^{<}(x, k) < f(x)$.

A function that admits such an approximator is said to be strictly lower semi-computable.

Remark. We also denote in the following $\langle \mathcal{T}_{lc}^{<} \rangle$ the set of codes of Turing machines that compute a strict lower approximator. ◇

Lemma: Mapping $\langle \phi_{lc} \rangle \xrightarrow{\langle \mathcal{T}_{lc} \rangle \Rightarrow \langle \mathcal{T}_{lc}^{<} \rangle} \langle \phi_{lc}^{<} \rangle$

Let ϕ_{lc} be a lower approximator for a function f. There exists a strict lower approximator $\phi_{lc}^{<}$ for this same function f.

Proof. Let ϕ_{lc} be a lower approximator for f. Let us define the function $\phi_{lc}^{<} : X \times \mathcal{N} \to \mathcal{Q}$ defined for any string x and integer k [as]:

$$\phi_{lc}^{<}(x, k) = \phi_{lc}(x, k) - 2^{-(k+1)}$$

Let us now verify that $\phi_{lc}^{<}$ is indeed a strict lower approximator for f.

- Let us show that $\phi_{lc}^{<}$ is totally computable. The function $k \mapsto 2^{-(k+1)}$ is total and computable. Since ϕ_{lc} is also [so], their difference $\phi_{lc}^{<}$ remains a total and computable function.
- Let us show that $\phi(x, \cdot)$ is strictly increasing for all $x \in X$. For an integer k:

$$\begin{aligned} \phi_{lc}^{<}(x, k+1) - \phi_{lc}^{<}(x, k) &= \left(\phi_{lc}(x, k+1) - 2^{-(k+2)}\right) - \left(\phi_{lc}(x, k) - 2^{-(k+1)}\right) \\ &= \left(\phi_{lc}(x, k+1) - \phi_{lc}(x, k)\right) + \left(2^{-(k+1)} - 2^{-(k+2)}\right) \\ &= 0 + 2^{-(k+2)} > 0 \end{aligned}$$

  Therefore $\phi_{lc}^{<}$ is even strictly increasing.

- Let us show that $\lim_{k\to\infty} \phi^<_{lc}(x,k) = f(x)$. As the sum of a finite number of limits is the limit of the sum:

$$\lim_{k\to\infty} \phi^<_{lc}(x,k) = \lim_{k\to\infty} \left(\phi_{lc}(x,k) - 2^{-(k+1)}\right) = \left(\lim_{k\to\infty} \phi_{lc}(x,k)\right) - 0 = f(x)$$

- Let us show that $\phi^<_{lc}(x,\cdot)$ never reaches its limit. By definition of a lower approximator, we know that $\phi_{lc}(x,k) \leq f(x)$ for all k and since $2^{-(k+1)}$ is strictly positive:

$$\phi^<_{lc}(x,k) = \phi_{lc}(x,k) - 2^{-(k+1)} < \phi_{lc}(x,k) \leq f(x)$$

  This demonstrates that $\phi^<_{lc}(x,k) < f(x)$ for all $x \in X$ and $k \in \mathcal{N}$.

The function $\phi^<_{lc}$ therefore satisfies all the required conditions. ■

1.10.5 Variant: Dyadic and compatible lower approximators

Definition: Approximator: dyadic / compatible / non-degenerate

A lower approximator $\phi^\sim_{lc}$ is qualified as dyadic, compatible, and non-degenerate if for all $x \in \mathbb{B}^*$ and $k \in \mathcal{N}$, it satisfies:

1. Dyadicity: The value $\phi^\sim_{lc}(x,k)$ is a dyadic rational (that is, of the form $m/2^n$ for $m,n \in \mathcal{N}$).
2. Compatibility: The sub-additivity property is verified:

$$\phi^\sim_{lc}(x,k) \geq \phi^\sim_{lc}(x0,k) + \phi^\sim_{lc}(x1,k)$$

3. Non-degeneracy: For $\Delta(x,k) := \phi^\sim_{lc}(x,k+1) - \phi^\sim_{lc}(x,k)$, we have,

$$\Delta(x,k) \geq \Delta(x0,k) + \Delta(x1,k)$$

Remark. We denote by $\langle\, \mathcal{T}^\sim_{lc} \,\rangle$ the set of codes of Turing machines that compute a dyadic compatible non-degenerate approximator. ◇

Property: Existence of a dyadic and compatible and non-degenerate approximator

For any function $f \in \Pi_\nearrow$, there exists a lower approximator $\phi^\sim_{lc}$ for f which is dyadic, compatible, and non-degenerate.

Proof. Admitted. ■

Theorem: Compact enumeration of $\Pi_\nearrow$ by $\langle\, \mathcal{T}^\sim_{lc} \,\rangle$

There exists a compact enumeration of lower semi-computable functions $\pi^\sim_{lc} : i \in \mathcal{N} \mapsto \langle\, \phi^\sim_{lc,i} \,\rangle \in \langle\, \mathcal{T}^\sim_{lc} \,\rangle$ such that

$$\forall f \in \Pi_\nearrow, \quad \exists i \in \mathcal{N}, \quad \pi^\sim_{lc}(i) = \langle\, \phi^\sim_{lc,i} \,\rangle \text{ and } \phi^\sim_{lc,i} \xrightarrow{lc} f$$

with the convention that $\langle\, \phi^{\sim}_{lc,i} \,\rangle$ is the code of a Turing machine in $\langle\, \mathcal{T}^{\sim}_{lc} \,\rangle$ that computes the function $\phi^{\sim}_{lc,i}$.

Proof. Analogous to the previous proof but depends on the result above which is admitted. ■