Foundations of Computer Science
Logic, models, and computations

Chapter: Computability

Course CSC_41012_EP

of l'Ecole Polytechnique

Olivier Bournez

bournez@lix.polytechnique.fr

Version of July 11, 2025

Computability

This chapter presents some main results of computability theory. In other words, this chapter is devoted to understanding the power of (modern today's) computers. We will prove that some problems cannot be solved using a computer: The objective is to explore the limits of computer programming.

Remark 1 *In practice, one could say that in computer science, one aims to solve problems by implementing algorithms as programs and that discussing the problems that cannot be solved by programs has only little interest. But, actually, it is very important to understand that we will not focus on problems for which no solution is known, but on something much stronger: We will focus on problems for which it is (provably) impossible to produce any algorithmic solution.*

Why focusing on understanding the problems that cannot be solved? First because understanding that a problem cannot be solved is useful: This means in particular that the problem must be simplified or modified in order to be solved. Second, because all these results are culturally very interesting and provide a perspective on programming, and on limits of computational devices, or of the automation of some tasks, such as the verification of programs.

1 Universal machines

1.1 Interpreters

A certain number of these results is the consequence of a simple fact, that has many consequences: One can program *interpreters*, that is to say programs that takes as input the description of some other program and that then simulate this program.

Example 2 *A language such as Java or Python is*[a] *interprete d: A Java program for example is compiled into some encoding that is called a* bytecode*. When one wants to start this program, the Java interpreter simulates this bytecode on the machine on which it is executed. This principle of interpretation allows a Java program to work on numerous platforms and directly on various machines: Only the interpreters depends on the machine on which the program is executed. This portability is partly what historically led to the success of the Java language (and* *remains true for interpreted languages such as Python).*

[a] The discussion here is true for early versions of Java. Now, just-in-time compilation is often used, and this discussion is only partially true.

The possibility of programming interpreters is thus practically extremely positive.

However, it also leads to mathematical proofs of numerous negative results or to paradoxical results about the impossibility of solving certain problems with a computer, even for very simple problems, as we will see shortly.

Programming an interpreter is possible in all usual programming languages, in particular it can be programmed using Turing machines.

Remark 3 *We will not talk about Java or Python in what follows, but about programs for Turing machines. Reasoning on Java (or any other language) would only complicate the discussion without changing the heart of the arguments.*

Let us start by getting convinced that one can construct an interpreter for Turing machines. In that context, an interpreter is called a *universal!Turing machine*.

1.2 Encoding Turing machines

We first need to fix a representation of programs of Turing machines. Here is a way to do so.

Remark 4 *The following encoding is only a convention. Any other encoding that would guarantee the possibility of decoding (for example in the spirit of Lemma 7) would work and would be a valid alternative.*

Definition 5 (Encoding of a Turing machine) *Let M be a Turing machine on alphabet $\Sigma = \{0, 1\}$.*

According to Definition 7.2, M is given by a 8-tuple

$$M = (Q, \Sigma, \Gamma, \mathbf{B}, \delta, q_0, q_a, q_r) :$$

- *Q is a finite set, whose elements are $Q = \{q_0, q_1, \cdots, q_{r-1}\}$, with the convention that q_0 is the initial state, and that $q_1 = q_a$, $q_2 = q_r$;*

- *Γ is a finite set, whose elements are $\Gamma = \{X_1, X_2, \cdots, X_s\}$, with the convention that X_s is the blank $\mathbf{B}$ symbol, and that X_1 is the symbol 0 of Σ, and that X_2 is the symbol 1 of Σ.*

For $m \in \{\leftarrow, |, \rightarrow\}$, define $\langle m \rangle$ as follows: $\langle \leftarrow \rangle = 1$, $\langle | \rangle = 2$, $\langle \rightarrow \rangle = 3$.

We can encode the transition function δ as follows: Suppose that the transition rule is $\delta(q_i, X_j) = (q_k, X_l, m)$: The encoding of this rule is the word $0^i 1 0^j 1 0^k 1 0^l 1 0^{\langle m \rangle}$ on alphabet $\{0, 1\}$. Observe that for any non-null integers i, j, k, l, there is no consecutive 1 in this word.

An encoding, denoted $\langle M\rangle$, of Turing machine M is a word on alphabet $\{0,1\}$ of the form

$$C_1 11 C_2 11 C_3 \cdots C_{n-1} 11 C_n,$$

where each C_i is the encoding of one of the transition rules of δ.

Remark 6 *To a given Turing machine M one can associate several encodings $\langle M\rangle$: In particular, one can permute the $(C_i)_i$, or the states, etc.*

The only interest of this particular encoding is that it can be easily decoded: One can easily find all the ingredients of the description of a Turing machine from the encoding of the machine.

For example, if one wants to determine the movement m for a given transition:

Lemma 7 (Decoding the encoding) *One can construct a Turing machine M with four tapes, such that if the encoding $\langle M'\rangle$ of a machine M' is put on its first tape, 0^i is put on its second tape, and 0^j is put on its third tape, M produces on its fourth tape the encoding $\langle m\rangle$ of the movement $m \in \{\leftarrow, |, \rightarrow\}$ such that $\delta(q_i, X_j) = (q_k, X_l, m)$ where δ is the transition function of the machine M'.*

Sketch of proof: Construct a machine that scans the encoding of M' until it finds the encoding of the associated transition, and that then reads in this encoding of this transition the desired value of m. □

We will need to encode pairs made of the encoding of a Turing machine M and of a word w on the alphabet $\{0,1\}$. One way to do this is to define the encoding of this pair, denoted $\langle\langle M\rangle, w\rangle$, by

$$\langle\langle M\rangle, w\rangle = \langle M\rangle 111 w,$$

that is to say the word obtained by concatenating the encoding of the Turing machine M, three times the symbol 1, and then the word w. Since our encoding of Turing machines never produces three consecutive 1's, the idea is that one can find in the word $\langle M\rangle 111 w$ the part which is $\langle M\rangle$ and the part which is w: In short: this encoding guarantees that one can decode unambiguously $\langle M\rangle$ and w from $\langle\langle M\rangle, w\rangle$.

1.3 Encoding pairs, triplets, etc...

We have just fixed an encoding that works for a Turing machine and a word. We now more generally fix a way to encode two words w_1 and w_2 into a unique word w. In other words, we give a way to encode an (ordered) pair of words, i.e., an element of $\Sigma^* \times \Sigma^*$, by a unique element (i.e. word) of Σ^*, that we will denote $\langle w_1, w_2\rangle$.

How to do this?

A first idea is to encode two words of Σ using a bigger alphabet, in such a way that it is possible to reconstruct the initial words.

For example, one could encode the words $w_1 \in \Sigma^*$ and $w_2 \in \Sigma^*$ by the word $w_1 \# w_2$ on alphabet $\Sigma \cup \{\#\}$. A Turing machine can then easily determine both w_1 and w_2 from the word $w_1 \# w_2$.

One can also re-encode the obtained word one letter after the other to obtain a way to encode two words on $\Sigma = \{0,1\}$ by a unique word on $\Sigma = \{0,1\}$: For example, if $w_1\#w_2$ is written $a_1 a_2 \cdots a_n$ on alphabet $\Sigma \cup \{\#\}$, we define $\langle w_1, w_2 \rangle$ as the word $e(a_1)e(a_2)\dots e(a_n)$ where $e(0) = 00$, $e(1) = 01$ and $e(\#) = 10$. This encoding is still decodable: From $\langle w_1, w_2 \rangle$, a Turing machine can decode w_1 and w_2.

From now, we will denote by $\langle w_1, w_2 \rangle$ the encoding of the pair consisting of the word w_1 and of the word w_2.

Observe that the coming results do not depend on the concrete encoding used for pairs: One can hence encode a pair made of a Turing machine and of a word as in previous section, or consider it as $\langle\langle M\rangle, w\rangle$, that is to say the encoding of a pair made of the encoding of the machine and of the word, indifferently.

1.4 Existence of a universal Turing machine

After these preliminary discussions, we now show that one can construct an interpreter, that is to say what is called a *universal!Turing machine* in the context of Turing machines.

Its existence is given by the following theorem:

Theorem 8 (Existence of a universal Turing machine) *There exists a Turing machine M_{univ} such that, on input $\langle\langle A\rangle, w\rangle$ where*

1. *$\langle A\rangle$ is the encoding of a Turing machine A;*
2. *$w \in \{0,1\}^*$;*

M_{univ} simulates the Turing machine A on input w.

Proof: One can easily see that there exists a Turing machine M_{univ} with three tapes such that if one puts:

- the encoding $\langle A\rangle$ of a Turing machine A on the first tape;
- a word w on alphabet $\Sigma = \{0,1\}$ on the second;

then M_{univ} simulates the Turing machine A on input w by using its third tape.

Indeed, the machine M_{univ} simulates transition after transition the machine A on input w on its second tape: M_{univ} uses the third tape to store 0^q where q is encoding the state of the Turing machine A at the transition that one is currently simulating: Initially, this tape contains 0, the encoding of q_0.

To simulate each transition of A, M_{univ} reads the letter X_j in front of its head on the second tape, then reads in the encoding $\langle A\rangle$ on the first tape the value of q_k, X_l and m, for the transition $\delta(q_i, X_j) = (q_k, X_l, m)$ of A, where 0^i is the content of the third tape. Then M_{univ} writes then X_l on its second tape, writes q_k on its third tape, and moves the head of the second tape according to the movement m.

To prove the result, it is then sufficient to use a Turing machine with only one tape that simulates the previous machine with three tapes, by first decoding $\langle A\rangle$ and w from the input. □

1.5 First consequences

Here is a first consequence of the existence of interpreters: The proof of Proposition 7.23, that is to say the proof that a non-deterministic Turing machine M can be simulated by a deterministic Turing machine.

Proof:[Of Propositon 7.23] The transition relation of the non-deterministic machine M is necessarily of *bounded degree of non-determinism*: That is to say, the number

$$r = max_{q\in Q, a\in\Gamma}|\{((q,a),(q',a',m))\in\delta\}|$$

is finite ($|.|$ denotes the size).

Suppose that for each pair (q,a), we number the choices among the transition relation of the non-deterministic machine M from 1 to (at most) r. At this moment, to describe the non-deterministic choices done by the machine up to time t, it is sufficient to give a sequence of t numbers between 1 and (at most) r.

We construct a (deterministic) Turing machine that simulates the machine M in the following way: For $t = 1, 2, \cdots$, it enumerates all the sequences of length t of integers between 1 and r. For each of these sequences, it simulates t steps of the machine M by making the choices given by the sequence. Doing this for all sequences simulates all potential non-deterministic choices of M. The machine stops and accepts as soon as it finds some t and a sequence such that M reaches an accepting configuration. □

2 Languages and decidable problems

Having established the existence of a universal Turing machines (interpreter), we will present a few additional definitions in this section.

In the rest of this chapter, we will only focus on problems whose answer is either *true* or *false*, which will allow us to simplify the discussion, see Figure 1.

2.1 Decision problems

Definition 9 *A* decision problem *is given by a set E, called the set of* instances, *and by a subset $E^+ \subseteq E$, called the set of the* positive instances.

The question on which we will focus is development of an algorithm (when this is possible) that decides whether a given instance is positive or not, i.e. belongs to E^+. We will formulate the decision problems systematically in the following form:

Definition 10 ($Name of the problem$**)**

***Input**: An instance (that is to say an element of E).*

***Answer**: Decide if a given property holds (that is to say if this element belongs to E^+).*

For example, we can consider the following problems:

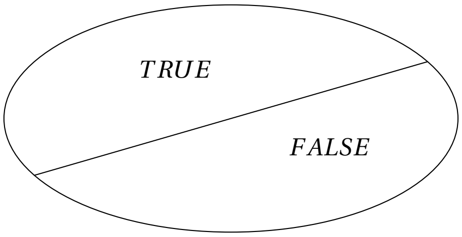

Figure 1: Decision problems: In a decision problem, we have a property that is either true or false for each instance. The objective is to distinguish the positive instances E^+ (were the answer is true) from negative instances $E \backslash E^+$ (where the property is false).

Definition 11 (PRIME NUMBER)

Input: *An integer n.*

Answer: *Decide if n is prime.*

Definition 12 (ENCODING)

Input: *A word w.*

Answer: *Decide if w is the encoding $\langle M \rangle$ of some Turing machine M.*

Definition 13 (REACH)

Input: *A triple consisting of a graph G, a vertex u and a vertex v of the graph.*

Answer: *Decide whether there exists a path between u and v in G.*

2.2 Problems versus Languages

We interchangably use the terminology *problem* or *language* in the upcoming discussions and chapters.

Remark 14 (Problems vs Languages) *This is due to the following considerations: To a decision problem we can associate a language and conversely.*

Indeed, to a decision problem we generally implicitly associate an encoding function (for example for graphs, a way to encode the graphs) that allows encoding the instances, that is to say the elements of E, by a given word on a certain alphabet Σ. One can then see E as a subset of Σ^*, where Σ is this alphabet: With a decision problem $\mathscr{P}$, we associate the language $L(\mathscr{P})$ defined as the set of words *which encode instances of E which belong to E^+:*

$$L(\mathscr{P}) = \{w \mid w \in E^+\}.$$

Conversely, we can see any language L as a decision problem, by formulating it as follows:

***Definition 15** (Problem associated to the language **L**)*

***Input**: A word w.*

***Answer**: Decide if $w \in L$.*

2.3 Decidable languages

We recall the notion of decidable language.

Definition 16 (Decidable language) *A language $L \subset \Sigma^*$ is said to be* decidable *if it is decided by some Turing machine.*

A language or a problem that is decidable is also said to be *recursive*. A language that is not decidable is said to be *undecidable*.

We write D for the class of languages or of problems that are *decidable*.

For example:

Proposition 17 *The decision problems* PRIME NUMBER, ENCODING *and* REACH *are* decidable.

The proofs consists in constructing a Turing machine that decides if its input is a prime number (respectively: the encoding of a Turing machine, or a positive instance of REACH). We leave this as an exercise in elementary programming to our readers.

Exercise 1 *(solution on page 234) Let A be the language consisting of the only string s where*

$$s = \begin{cases} 0 & \textit{if God does not exist} \\ 1 & \textit{if God exists} \end{cases}$$

Is A decidable? Why? (Hint: The answer does not depend on the religious convictions the reader).

3 Undecidability

In this section, we will prove one of the philosophically most important result in the theory of programming: The existence of problems that cannot be decided, i.e. that are *undecidable*.

3.1 First considerations

Observe first that this can be established easily.

Theorem 18 *There exist decision problems which are not* decidable.

Proof: We have seen that one can encode a Turing machine by a finite word on alphabet $\Sigma = \{0,1\}$, see Definition 5. There is consequently a countable number of Turing machines.

By contrast, there is an uncountable number of languages over the alphabet $\Sigma = \{0,1\}$: Indeed, we saw in Chapter 1 that the power set of $\mathbb{N}$ is uncountable, using Cantor diagonalization argument. Now, this must also be the case for the set of languages, as there is an easy bijection this latter set and power set of $\mathbb{N}$: just take the characteristic functions of the languages.

There are consequently more problems than those that can be solved by any Turing machine. There must thus be decision problems that are not solved by any Turing machine (and there is even an uncountable number of such problems). □

In general, a proof as the one above does not say anything about examples of undecidable problems. Are they esoteric? Are they only of interest for theoreticians?

Unfortunately, this is not the case as even some simple and natural problems turn out not be solvable by any algorithm.

3.2 Is this problematic?

For example, in one important undecidable problem, we are given a program and a specification of what this program is supposed to do (for example sorting some numbers) and one wants to check if the program matches its specification (i.e. is a correct sorting algorithm).

We could hope that this process of *verification* could be automatized, that is to say that we could design an algorithm that would test if a given program satisfies its *specification*. Unfortunately, this is impossible: The general problem of *verification* is undecidable, and can thus not be solved on a computer.

We will meet some other *undecidable* problems in this chapter. Our objective will be to make our reader feel what types of problems are undecidable, and to understand the techniques that permit to prove that a given problem cannot be solved algorithmically, i.e. cannot be solved by a computer.

3.3 A first undecidable problem

We will use a *diagonalisation argument* that is to say an argument close to Cantor's diagonalisation. Recall that Cantor's diagonalisation is used to prove that the set of subsets of $\mathbb{N}$ is uncountable, see the first chapter.

Remark 19 *Behind the previous argument on the fact that there is an uncountable number of languages on the alphabet* $\Sigma = \{0,1\}$ *is already a diagonalization argument. Here, we will do a more explicit, and more constructive diagonaliza-*

tion.

We call the following decision problem *universal!language*.

Definition 20 ($\mathrm{L}_{\mathrm{univ}}$)

Input: *The encoding $\langle M\rangle$ of a Turing machine M and a word w.*

Answer: *Decide if the machine M accepts the word w.*

Remark 21 *One can also see this problem in the following way: We are given a pair $\langle\langle M\rangle, w\rangle$, where $\langle M\rangle$ is the encoding of a Turing machine M, and w a word, and one wants to decide if the machine M accepts the word w.*

Theorem 22 *The problem* $\mathrm{L}_{\mathrm{univ}}$ *is not decidable.*

Proof: We prove the result by contradiction. Suppose that $\mathrm{L}_{\mathrm{univ}}$ is decided by some Turing machine A.

We are then able to construct a machine B working as follows:

- B takes as input a word $\langle C\rangle$ representing the encoding of a Turing machine C;
- B calls the Turing machine A on the pair $\langle\langle C\rangle, \langle C\rangle\rangle$ (that is to say on the input consisting of the encoding of the Turing machine C and the word w equal to this encoding);
- If the Turing machine A:
  - accepts this word, B rejects;
  - rejects this word, B accepts.

By construction, and from our hypothesis, B halts on every input.

We prove now that there is a contradiction, by applying the Turing machine B on the word $\langle B\rangle$, that is on the word encoding the Turing machine B.

- If B accepts $\langle B\rangle$, then if follows by definition of $\mathrm{L}_{\mathrm{univ}}$ and of A, that A accepts $\langle\langle B\rangle, \langle B\rangle\rangle$. But if A accepts this word, B is constructed such that it rejects the input $\langle B\rangle$. Contradiction.
- If B rejects $\langle B\rangle$, then it follows by definition of $\mathrm{L}_{\mathrm{univ}}$ and of A, that A rejects $\langle\langle B\rangle, \langle B\rangle\rangle$. But if A rejects this word, B accept the input $\langle B\rangle$ by construction. Contradiction.

□

3.4 Semi-decidable problems

While the problem $\mathrm{L}_{\mathrm{univ}}$ is undecidable, it is semi-decidable in the following sense:

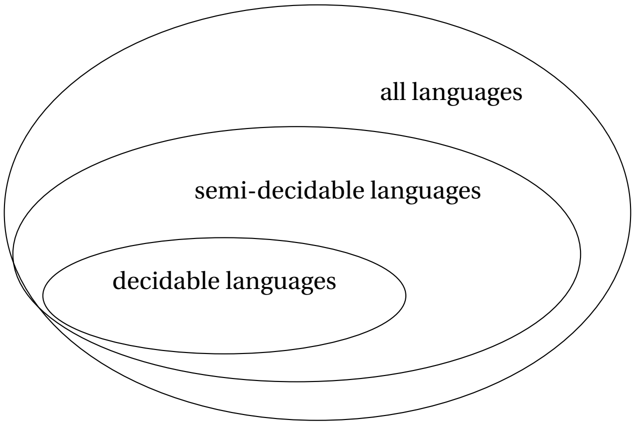

Figure 2: Inclusions between classes of languages.

Definition 23 (Semi-decidable language) *A language* $L \subset M^*$ *is said to be* semi-decidable *if it is the set of words accepted by some Turing machine* M*.*

Corollary 24 *The universal language* L_{univ} *is semi-decidable.*

Proof: To know if one must accept some input that is the encoding $\langle M \rangle$ of a Turing machine M and of a word w, it is sufficient to simulate the Turing machine M on input w. One stops the simulation and one accepts if one detects in this simulation that the Turing machine M reaches some accepting state. Otherwise, one simulates M for ever. □

A language *semi-decidable* is also called *computably enumerable* (sometimes also called *recursively enumerable*).

We write CE for the class of languages and decision problems semi-decidable: See Figure 2.

Corollary 25 $D \subsetneq CE$.

Proof: The inclusion follows directly from the definitions. Since L_{univ} is in CE and is not in D, the inclusion is strict. □

3.5 A problem that is not semi-decidable

Let us start by establish the following fundamental result:

Theorem 26 *A language is* decidable *if and only it it is* semi-decidable *and its complement is also* semi-decidable.

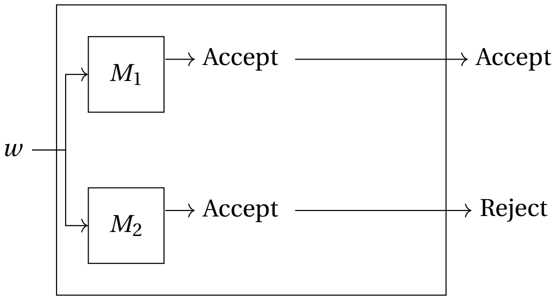

Figure 3: Illustration of the proof of Theorem 26.

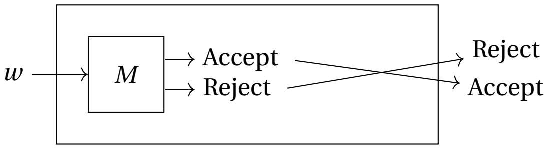

Figure 4: Construction of a Turing machine that accepts the complement of a decidable language.

> **Remark 27** *Theorem 26 explains the terminology* semi-decidable*: A language that is semi-decidable and whose complement as well, is decidable. So when a language is semi-decidable, one half of the requirements to be decidable is in a sense satisfied.*

Proof: Direction $\Leftarrow$. Suppose that L is semi-decidable as well as its complement. There exists a Turing machine M_1 which halts and accepts words of L, and a Turing machine M_2 which halts and accepts words of its complement. We construct a Turing machine M that, on a given input w, simulates in parallel[1] M_1 and M_2, (that is to say it simulates t steps of M_1 on w, and then t steps of M_2 on w, for $t = 1, 2, \cdots,$) until one of the two machines halts. See Figure 3. If M_1 halts and accepts, the Turing machine M accepts. If M_2 does so, the machine M rejects. Obviously, the Turing machine that we just described decides L.

Direction $\Rightarrow$. By definition, a decidable language is semi-decidable. By exachanging the accepting state and the rejecting state in the Turing machine, its complement is also decidable (See Figure 4) and hence is also semi-decidable. □

We now consider then the complement of the problem $\mathrm{L_{univ}}$, that we will denote $\overline{\mathrm{L_{univ}}}$.

[1] One alternative is to consider that M is a non-deterministic Turing machine that simulates in a non-deterministic way either M_1 or M_2.

Remark 28 *In other words, by definition, a word w is in $\overline{\mathrm{L}_{\mathrm{univ}}}$ if and only w is not in $\mathrm{L}_{\mathrm{univ}}$, that is to say*

- *not of the form $\langle\langle M\rangle, w\rangle$, for some Turing machine M,*
- *or of the form $\langle\langle M\rangle, w\rangle$ but with Turing machine M that does not accept input w.*

Corollary 29 *The problem $\overline{\mathrm{L}_{\mathrm{univ}}}$ is not semi-decidable.*

Proof: Otherwise, by the Theorem 26, its complement, the problem $\mathrm{L}_{\mathrm{univ}}$, would be decidable. □

3.6 On the terminology

A decidable language is also called a *recursive language*: The terminology is a reference to the notion of recursive functions, see for example the course [Dowek, 2008] which present computability through recursive functions, or Section 7.2.

The notion of *enumerable* in *computably enumerable* is explained by the following result.

Theorem 30 *A language $L \subset M^*$ is* computably enumerable *if and only if one can construct some Turing machine that outputs one after the other all the words of language L.*

Proof: Direction ⇒. Suppose that L is computably enumerable. There exists some Turing machine A which halts and which accepts the words of L.

The set of pairs (t, w), where t is some integer, and where w is a word is countable. One can even get convinced easily that is *effectively* countable: One can construct some Turing machine that produces the encoding $\langle t, w\rangle$ of all the pairs (t, w). For example, one considers a loop that for $t = 1, 2, \cdots$ for ever, considers all the words w of length or equal to t, and produces for each pair the word $\langle t, w\rangle$.

Consider a Turing machine B that for each produced pair (t, w), simulates t steps of the machine A. If the machine halts and accepts in exactly t steps, B outputs the word w. Otherwise B does not print anything for this pair.

A word of language L, is accepted by A at some particular time t. It will then be printed by B when it considers the pair (t, w). By assumption, any word w produced by B is accepted by A, and hence is a word of L.

Direction ⇐. If there is a Turing machine B which enumerates all the words of the language L, then one can construct a Turing machine A, which, given some word w, simulates B, and every time that B produces a word, compares this word to the word w. If there are equal, then A stops and accepts. Otherwise, A continues.

By construction, on some input w, A halts and accepts if w is among the words enumerated by B, that is to say if $w \in L$. If w is not among these words, by hypothesis, $w \not\in L$, and hence by construction, A will not accept w. □

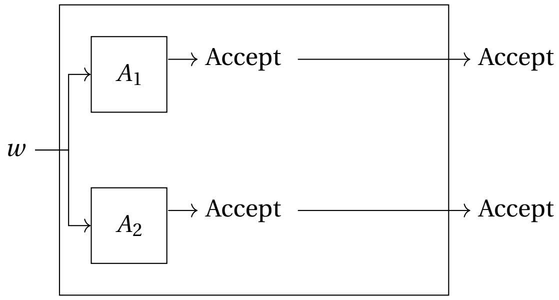

Figure 5: Construction of a Turing machine accepting $L_1 \cup L_2$.

3.7 Closure properties

Theorem 31 *The set of semi-decidable languages is closed by union and by intersection: In other words, if L_1 and L_2 are semi-decidable, then $L_1 \cup L_2$ and $L_1 \cap L_2$ are.*

Proof: Suppose that L_1 is $L(A_1)$ for some Turing machine A_1 and L_2 is $L(A_2)$ for some Turing machine A_2. Then $L_1 \cup L_2$ is $L(A)$ for the Turing machine A which simulates in parallel A_1 and A_2 and which halts and accepts as soon as one of the Turing machine A_1 or A_2 halts and accepts: See Figure 5.

$L_1 \cap L_2$ is $L(A)$ for the Turing machine A which simulates in parallel A_1 and A_2 and which halts and accepts as soon as both Turing machines A_1 and A_2 halt and accept.

□

Theorem 32 *The set of decidable languages is closed by union, intersection and complement: In other words, if L_1 and L_2 are decidable, then $L_1 \cup L_2$, $L_1 \cap L_2$, and L_1^c are.*

Proof: We have already used the closure by complement. Indeed, by exchanging the accepting state and the rejecting state of the Turing machine, the complement of a decidable set is also decidable: See Figure 4.

It remains to prove that with the hypotheses, the languages $L_1 \cup L_2$ and $L_1 \cap L_2$ are decidable. But this is clear from the previous theorem and the fact that a set is decidable if and only if it is semi-decidable as well as its complement, by using Morgan's law (the complement of a union is the intersection of the complement, and symmetrically) and the fact that complements of decidable languages are decidable. □

In particular, we deduce:

Definition 33 ($\overline{\mathrm{L_{univ}}}'$)

Input: *The encoding $\langle M \rangle$ of a Turing machine M and a word w.*

Answer: *Decide if the machine M does not accept the word w.*

Corollary 34 *The problem $\overline{\mathrm{L_{univ}}}'$ is undecidable.*

Proof: $\overline{\mathrm{L_{univ}}}$ is the union of $\overline{\mathrm{L_{univ}}}'$ and of the complement of ENCODING. If $\overline{\mathrm{L_{univ}}}'$ were decidable, then $\overline{\mathrm{L_{univ}}}$ would be decidable. □

4 Other undecidable problems

Having shown a first result to be undecidable, we will now obtain other undecidable languages.

4.1 Reductions

We know two undecidable problems, $\mathrm{L_{univ}}$ and its complement. Our aim is now to obtain some more. We will also show how to compare problems. To this end, we will introduce the notion of *reduction*.

First, we generalize the notion of *computable* from languages and decision problems to functions.

Definition 35 (Computable function) *Let Σ and Σ' be two alphabets. A (total) function $f : \Sigma^* \to \Sigma'^*$ is* computable *if there exists a Turing machine A working on alphabet $\Sigma \cup \Sigma'$, such that for all words w, A on input w, halts and accepts, with $f(w)$ written on its tape at the moment it halts.*

One can see that the composition of two computable functions is computable.

This provides a way to introduce a notion of reduction between problems: The idea is that if A reduces to B, then problem A is as least as easy as problem B, or, if one prefers, the problem B is at least as hard than problem A. See Figure 6 and Figure 7.

Definition 36 (Reduction) *Let A and B two problems of respective alphabet Σ_A and Σ_B. A* reduction from A to B *is a computable function $f : \Sigma_A^* \to \Sigma_B^*$ such that $w \in A$ if and only if $f(w) \in B$. We write $A \leq_m B$ when A reduces to B, i.e., there is a reduction from A to B.*

Reductions, as defined above, behave as we would like: A problem is at least as easy (and hard) as itself, and the relation "is at least as easy as" is transitive. In other words:

Theorem 37 *$\leq_m$ is a preorder:*

1. *$L \leq_m L$;*

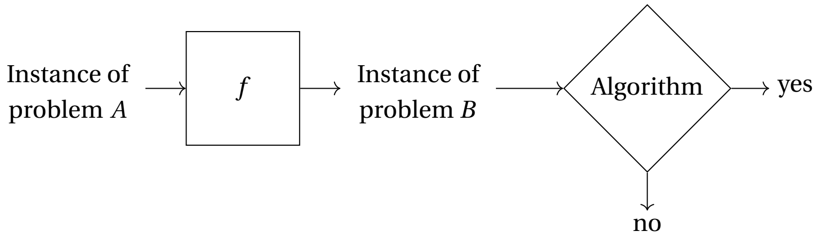

Figure 6: Reduction of problem A to problem B. If one can solve the problem B, then one can solve the problem A. The problem A is consequently at least as easy as problem B, denoted by $A \leq_m B$.

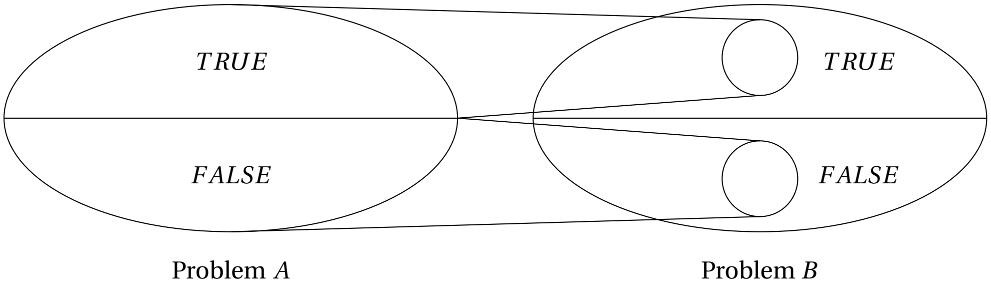

Figure 7: A reduction transforms the positive instances to positive instance,s and negative instances to negative instances.

2. $L_1 \leq_m L_2$, $L_2 \leq_m L_3$ implies $L_1 \leq_m L_3$.

Proof: Consider the identity function as function f for the first point.

For the second, suppose that $L_1 \leq_m L_2$ via the reduction f, and that $L_2 \leq_m L_3$ via the reduction g. We have $x \in L_1$ if and only if $g(f(x)) \in L_2$. It is then sufficient to see that $g \circ f$, being the composition of two computable functions is computable. □

Remark 38 *The reduction relation $\leq_m$ is not an order, since $L_1 \leq_m L_2$, $L_2 \leq_m L_1$ does not imply $L_1 = L_2$. It is actually rather natural to introduce the following concept: Two problems L_1 and L_2 are equivalent, denoted $L_1 \equiv L_2$, if $L_1 \leq_m L_2$ and $L_2 \leq_m L_1$.*

Intuitively, if a problem is at least as easy as a decidable problem, then it is decidable. We show this here formally:

Proposition 39 (Reduction) *If $A \leq_m B$, and if B is decidable then A is decidable .*

Proof: A is decided by the Turing machine that, on a given input w, computes $f(w)$, and then simulate the Turing machine that decides B on input $f(w)$. Since we have $w \in A$ if and only if $f(w) \in B$, the Turing machine behaves correctly. □

Proposition 40 (Reduction) *If $A \leq_m B$, and if A is undecidable, then B is undecidable.*

Proof: This is the contrapositive of the previous proposition. □

4.2 Some other undecidable problems

Proposition 40 provides a way to obtain the undecidability proof for many other problems.

As a first example, it is not possible to determine algorithmically if a given Turing machine halts.

Definition 41 (*$Halting\ Problem$*)

Input: *The encoding $\langle M \rangle$ of a Turing machine M and some input w.*

Answer: *Decide if M halts on input w.*

Proposition 42 *The problem $Halting\ Problem$ is undecidable.*

Proof: We construct a reduction from $\mathrm{L_{univ}}$ to the halting problem: For every pair $\langle\langle A \rangle, w\rangle$, we consider the Turing machine B defined in the following way (see Figure 8):

- B takes as input a word w;

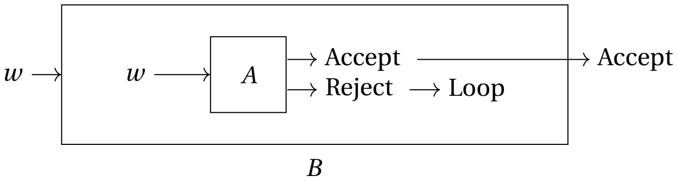

Figure 8: Illustration of the Turing machine used in the proof of Proposition 42.

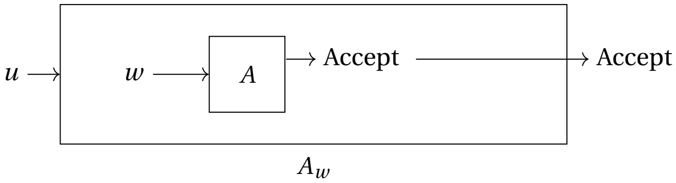

Figure 9: Illustration of the Turing machine used in the proof of Proposition 44.

- B simulates A on w;
- If A accepts w, then B accepts. If A rejects w, then B loops (possibly B simulates A forever, if A never halts).

The function f that maps $\langle\langle A\rangle, w\rangle$ to $\langle\langle B\rangle, w\rangle$ is computable. Furthermore, we have $\langle\langle A\rangle, w\rangle \in \mathrm{L}_{\mathrm{univ}}$ if and only if B halts on input w, that is to say $\langle\langle B\rangle, w\rangle \in Halting\ Problem$.
□

As another example, it is not possible to decide algorithmically (that is to say by a Turing machine, i.e. a program) if a Turing machine accepts at least one input:

Definition 43 ($L_\emptyset$)

Input: *The encoding $\langle M\rangle$ of a Turing machine M.*

Answer: *Decide if $L(M) \neq \emptyset$.*

Proposition 44 *The problem $L_\emptyset$ is undecidable.*

Proof: We design a reduction from $\mathrm{L}_{\mathrm{univ}}$ to $L_\emptyset$: For any pair $\langle\langle A\rangle, w\rangle$, we consider the Turing machine A_w defined as follows (see Figure 9):

- A_w takes as input a word u;
- A_w simulates A on w;
- If A accepts w, then A_w accepts.

The function f that maps $\langle\langle A\rangle, w\rangle$ to $\langle A_w\rangle$ is indeed computable. Furthermore, we have $\langle\langle A\rangle, w\rangle \in \mathrm{L}_{\text{univ}}$ if and only if $L(A_w) \neq \emptyset$, that is to say $\langle A_w\rangle \in L_\emptyset$: Indeed, A_w accepts either all the words (and hence the associated language is not empty) if A accepts w, or accepts no word otherwise (and hence the associated language is empty). □

Definition 45 ($L_{\neq}$)

Input: *The encoding $\langle A\rangle$ of a Turing machine A and the encoding of $\langle A'\rangle$ of a Turing machine A'.*

Answer: *Determine if $L(A) \neq L(A')$.*

Proposition 46 *The problem $L_{\neq}$ is undecidable.*

Proof:

We design a reduction from $L_\emptyset$ to $L_{\neq}$. We consider a Turing machine B that accepts the empty language: Take for example a Turing machine B that enters immediately in a non-terminating loop. The function f that maps $\langle A\rangle$ to the pair $\langle A, B\rangle$ is indeed computable. Furthermore, we have $\langle A\rangle \in L_\emptyset$ if and only if $L(A) \neq \emptyset$ if and only if $\langle A, B\rangle \in L_{\neq}$. □

4.3 Rice's theorem

The two previous results can be seen as the consequence of a very general statement that asserts that any non-trivial property of algorithms is undecidable.

A property of semi-decidable languages is said to be non-trivial if it is not always true or always false for Turing machines: That is to say, there is at least a Turing machine M_1 such that $L(M_1)$ satisfies P and a Turing machine M_2 such that $L(M_2)$ does not satisfy P.

Theorem 47 (Rice's theorem) *Any non-trivial property of semi-decidable languages is undecidable.*

Then the following decision problem L_P:

Input: *The encoding $\langle M\rangle$ of a Turing machine M;*

Answer: *Decide if $L(M)$ satisfies property P;*

is undecidable.

Remark 48 *Observe that if a property P is trivial in the above sense, L_P is trivially decidable: Construct a Turing machine that does not even read its input and accepts (respectively: rejects).*

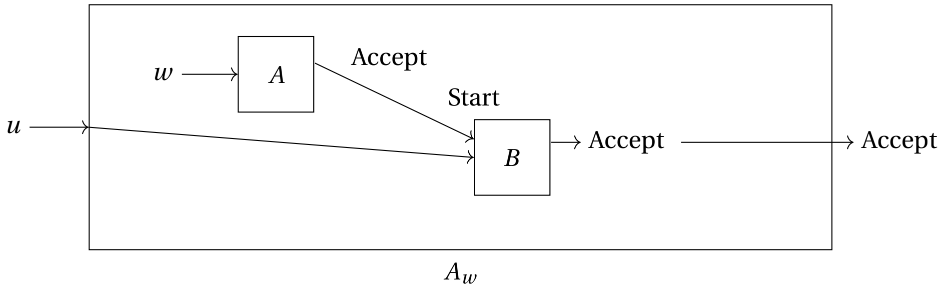

Figure 10: Illustration of the Turing machine used in the proof of Rice Theorem.

Proof: We need to prove that the decision problem L_P is undecidable.

Replacing P by its negation if needed, one can assume that the empty word does not satisfy the property P (proving the undecidability of L_p is equivalent to proving the undecidability of its complement). Since P is non-trivial, there exists at least one Turing machine B whose accepted language $L(B)$ satisfies P.

We design a reduction from L_{univ} to the language L_P. Given a pair $\langle\langle A\rangle, w\rangle$, we consider a Turing machine A_w defined as follows (see Figure 10):

- A_w takes as input a word u;
- On word u, A_w simulates A on word w;
- If A accepts w, then A_w simulates B on the word u: A_w accepts if and only if B accepts u.

In other words, A_w accepts, if and only if A accepts w and if B accepts u. If w is accepted by A, then $L(A_w)$ equals $L(B)$, and hence satisfies property P. If w is not accepted by A, then $L(A_w) = \emptyset$, and hence does not satisfy property P.

The function f that maps $\langle\langle A\rangle, w\rangle$ to $\langle A_w\rangle$ is obviously computable. □

Exercise 2 *(solution on page 235) Prove that the set of encodings of Turing machines which accepts all the words which are palindromes (possibly accepting other words) is undecidable.*

4.4 The drama of verification

From Rice's theorem, we directly get the following:

Corollary 49 *It is not possible to design an algorithm that takes as input a program, its specification, and that determines if the program satisfies the specification.*

The above is true, even if the specification is fixed to a property P (as soon as the property P is not trivial) by Rice's theorem.

From what we have seen before, this turns out to be true even for very rudimentary systems. For example:

Corollary 50 *It is not possible to design an algorithm that takes as input the description of a system, its specification, and that determines if the system satisfies its specification.*

The above is true, even if the specification is fixed to a property P (as soon as the property P is not trivial), and even for systems as simple as two counter machines by Rice's theorem, and the simulation results of the previous chapter.

4.5 Notion of completeness

We now introduce a notion of completeness.

Definition 51 (CE-**completeness)** *A problem A is called* CE*-complete, if:*

1. *it is computably enumerable;*
2. *for every other computably enumerable problem B we have $B \leq_m A$.*

In other words, a CE-complete problem is maximal for $\leq_m$ among the problems of class CE.

Theorem 52 *The problem* $\mathrm{L}_{\mathrm{univ}}$ *is* CE*-complete.*

Proof: $\mathrm{L}_{\mathrm{univ}}$ is semi-decidable. Now, let L be a semi-decidable language. By definition, there exists a Turing machine A which recognizes L. Consider the function f which maps w to the word $\langle\langle A\rangle, w\rangle$. We have that $w \in L$ if and only if $f(w) \in \mathrm{L}_{\mathrm{univ}}$, and hence we obtain $L \leq_m \mathrm{L}_{\mathrm{univ}}$. □

5 Natural undecidable problems

One can object that the previous problems, relative to algorithms are "artificial" in the sense that they are talking about properties of algorithms, the algorithms have been in turn defined by the theory of computability.

It is difficult to define formally what one would like to call a "*natural problem*", but one can say that a problem that has been discussed before the invention of computability theory is (more) natural.

5.1 Hilbert's tenth problem

This is clearly the case of Hilbert's tenth problem, identified by David Hilbert as one of the most interesting problems for the 20th century in 1900: Can we determine if a given polynomial equation with integer coefficients has an integer solution?

Definition 53 ($Hilbert's\ 10th\ problem$)

Input: *A polynomial $P \in \mathbb{N}[X_1, \cdots, X_n]$ with integer coefficients.*

Answer: *Decide if P has an integer root*

Theorem 54 *The problem $Hilbert's\ 10th\ problem$ is undecidable.*

The proof of this result, due to Matiyasevich [Matiyasevich, 1970] (extending statements from Davis, Putnam and Robinson) is beyond the ambition of this document.

5.2 The Post correspondence problem

The proof of the undecidability of Post correspondence problem is easier, even if we will not give it here. One can consider this problem as a *natural problem*, in the sense that it is not making a (direct) reference to the notion of algorithms, or to Turing machines.

Definition 55 (Post correspondence problem)

Input: *A sequence $(u_1, v_1), \cdots, (u_n, v_n)$ of pairs of words on alphabet Σ.*

Answer: *Decide if this sequence admits a* correspondence*, that is to say a sequence of indexes $i_1, i_2, \cdots, i_m$ of $\{1, 2, \cdots, n\}$ such that*

$$u_{i_1} u_{i_2} \cdots u_{i_m} = v_{i_1} v_{i_2} \cdots v_{i_m}.$$

Theorem 56 *The problem* Post correspondence problem *is undecidable.*

5.3 Decidability/Undecidability of theories in logic

We have already presented in Chapter 6, the axioms of Robinson arithmetic and the axioms of Peano: One expects these axioms to be true on the integers, that is to say in the *standard model of the integers* where the base set is the integers, and where $+$ is interpreted by addition, $*$ by multiplication, and $s(x)$ is interpreted by the function that maps x to $x+1$.

Given some closed formula F on the signature that contains the symbols of arithmetic, F is either true or false on the the integers (that is to say in the standard model of the integers). Let's call *theory!of the arithmetic* the set $Th(\mathbb{N})$ of closed formulas F which are true on the integers.

The constructions of the previous chapter proves the following result:

Theorem 57 *$Th(\mathbb{N})$ is not decidable.*

Proof: We prove in the following chapter that $Th(\mathbb{N})$ is not computably enumerable. It is then sufficient to observe that a decidable set is computably enumerable to obtain a contradiction with assuming $Th(\mathbb{N})$ decidable. □

We can prove (we will not do it) that if one considers the set of formulas written without using the multiplication symbol, then the associated theory is decidable.

Theorem 58 *One can decide if a closed formula F on signature $(\mathbf{0}, s, +, =)$ (i.e. the one from Peano without the multiplication symbol) is satisfied on the integers.*

We also obtain the following results:

Theorem 59 *Let F be a closed formula on the signature of Peano axioms. The decision problem that consists, given F, to determine whether it can be proved from the axioms of Peano is undecidable..*

Proof: Given some pair $\langle\langle M\rangle, w\rangle$, where M is a machine and w a word, we show in the next chapter how to produce some closed formula γ on the signature of arithmetic such that

$$\langle\langle M\rangle, w\rangle \in \overline{\mathrm{HP}} \Leftrightarrow \gamma \in Th(\mathbb{N}),$$

where $\overline{\mathrm{HP}}$ is the complement of the halting problem of Turing machines.

But, doing so, one can check that the reasoning that is done for that can be formalized with Peano arithmetic and can be deduced from Peano axioms.

We consequently have actually $\langle\langle M\rangle, w\rangle \in \overline{\mathrm{HP}}$ if and only if γ can be proved from Peano axioms.

This provides a reduction from the complement of the universal problem of Turing machines to our problem: The transformation that maps $\langle\langle M\rangle, w\rangle$ to γ is indeed easily computable. Our problem is hence undecidable, since the first is. □

One can prove.

Theorem 60 *One can decide if some closed formula F on the signature $(\mathbf{0}, s, +, =$ $)$ (i.e. the one from Peano without the multiplication symbol) is provable from Peano axioms.*

6 Fixpoint problems

The results of this section are very subtle, but extremely powerful.

Let us start by a simple version, that will help understanding the proofs.

Proposition 61 *There exists a Turing machine A^* that produces its own algorithm: It outputs $\langle A^*\rangle$.*

In other words, it is possible for a program to write its own code.

This is true in any programming language which is equivalent to Turing machines.

In *UNIX* shell for example, the following program

```
x=’y=‘echo . | tr . "\47" ‘; echo "x=$y$x$y;$x"’; y=‘echo . | tr .
 "\47"‘; echo "x=$y$x$y;$x"
```

produces

```
x=’y=‘echo . | tr . "\47" ‘; echo "x=$y$x$y;$x"’; y=‘echo . | tr .
 "\47"‘; echo "x=$y$x$y;$x"
```

which is a command, once executed, prints its own code.

Such programs are sometimes called *quines*, in honor of philosopher Willard van Orman Quine, who discussed the existence of self-reproducing programs.

Proof: We consider Turing machines which halt on every input. For two such machines A and A', we write AA' for the Turing machine that is obtained by composing in a sequential manner A and A'. Formally, AA' is the Turing machine that runs first the program of A and then when A halts with its tape set to w, runs the program of A' on the input w.

We construct the following machines:

1. Given a word w, the Turing machine $Print_w$ halts with the result w;

2. For a given input w of the form $w = \langle X \rangle$, where X is a Turing machine, the Turing machine B produces as output the encoding of the Turing machine $Print_w X$, that is to say the encoding of the Turing machine obtained by composing $Print_w$ and X.

We consider then the Turing machine A^* given by $Print_{\langle B \rangle}B$, that is to say the sequential composition of the machines $Print_{\langle B \rangle}$ and B.

Let us unfold the result of this machine: The Turing machine $Print_{\langle B \rangle}$ produces as output $\langle B \rangle$. The sequential composition with B produces then the encoding of $Print_{\langle B \rangle}B$, which is indeed the encoding of the Turing machine A^*. □

The recursion theorem allows self references in programing languages. Its proof consists of extending the ideas behind the proof of the previous result.

Theorem 62 (Recursion theorem) *Let $t : M^* \times M^* \to M^*$ be a computable function. Then there exists a Turing machine R which computes a function $r : M^* \to M^*$ such that for all words w*

$$r(w) = t(\langle R \rangle, w).$$

Tje statement of the Recursion Theorem is rather technical, but its use is simple. To obtain a Turing machine that obtains its own description, and use it to compute, we simply need a Turing machine T which computes some function t as in the statement, that takes as input some supplementary entry which contains the description of the Turing machine. Then the recursion theorem produces a new machine R which operates as T but with the description of $\langle R\rangle$ encoded in its code.

Proof: We use basically the same idea as before. Let T be a Turing machine that computes a function t: T takes as input a pair $\langle u,w\rangle$ and produces as output $t(u,w)$.

We then consider the following machines:

1. Given a word w, the Turing machine $Print_w$ takes as input a word u and halts with the result $\langle w,u\rangle$;

2. For a given input w' of the form $\langle\langle X\rangle,w\rangle$, the Turing machine B:

   (a) computes $\langle\langle Print_{\langle X\rangle}X\rangle,w\rangle$, where $Print_{\langle X\rangle}X$ denotes the Turing machine which composes $Print_{\langle X\rangle}$ with X;

   (b) and then gives the control to the Turing machine T.

We consider then the Turing machine R given by $Print_{\langle B\rangle}B$, that is to say the Turing machine obtained by composing $Print_{\langle B\rangle}$ with B.

Let us unfold the result $r(w)$ of this machine R on an input w: On the input w, the Turing machine $Print_{\langle B\rangle}$ produces as output $\langle\langle B\rangle,w\rangle$. The composition with B produces then the encoding of $\langle\langle Print_{\langle B\rangle}B\rangle,w\rangle$, and gives the control to T. The latter produces then $t(\langle Print_{\langle B\rangle}B\rangle,w)=t(\langle R\rangle,w)=r(w)$. □

We directly obtain the followign result:

Theorem 63 (Kleene fixed point theorem) *Let f be a computable function that to every word $\langle A\rangle$ encoding a Turing machine associates a word $\langle A'\rangle=f(\langle A\rangle)$ encoding a Turing machine. For conciseness, write $A'=f(A)$ in that case.*

Then there exists a Turing machine A^* such that $L(A^*)=L(f(A^*))$.

Proof: Consider a function $t: M^*\times M^*\to M^*$ such that $t(\langle A\rangle,x)$ is the result of the simulation of the Turing machine $f(A)$ on input x. By the previous theorem, there exists a Turing machine R which computes a function r such that $r(w)=t(\langle R\rangle,w)$. By construction $A^*=R$ and $f(A^*)=f(R)$ have hence the same value on w for all w. □

Remark 64 *One can interpret the previous results in relation with computer viruses. Indeed a virus is a program that aims at propagating, that is to say at self-reproducing, without being detected. The principle of the previous proof of the recursion theorem is a way to self-reproduce, by duplicating its own code.*

7 A few remarks

7.1 Computing on other domains

We have introduced the notion of decidable sets, computably enumerable, for the subsets of Σ^*, and the notion of (total) computable function $f:\Sigma^*\to\Sigma^*$.

One may want to work on other domains than words over some given alphabet, for example on the integers. In that case, on can simply consider encodings of integers by their binary expansion to come back to the case of a word on the alphabet $\{0,1\}$.

Remark 65 *One could also encode an integer for example in unary, i.e., n by a^n for a letter a on some alphabet Σ with $a \in \Sigma$. This would not change the notion of computable function.*

In the general case, to work on some domain E, one fixes some encoding of the elements of E in some alphabet Σ: One then says for example that a subset $S \subset E$ is computably enumerable (respectively decidable) if the subset of encodings of elements of E is computably enumerable (resp. decidable).

Similarly, a (total) function $f: E \to F$ is called computable if the function from the encoding $e \in E$ to the encoding of $f(e) \in F$ is computable.

Example 66 *We can encode $\vec{n} = (n_1, \dots, n_k) \in \mathbb{N}^k$ by*

$$\langle \vec{n} \rangle = a^{n_1+1} b a^{n_2+1} b \cdots a^{n_k+1}$$

on the alphabet $\Sigma = \{a, b\}$. A function $f: \mathbb{N}^k \to \mathbb{N}$ is called computable if it is computable with respect to this encoding.

The obtained notions of computable functions, semi-decidability, etc. do not depend on the encodings, for the usual encodings (actually technically as soon as one can go from one encoding to the other encoding in a computational way, a property that holds for all “natural” encodings of objects, and in particular for all encodings considered in this document).

7.2 Algebraic vision of computability

The notions of computability are sometimes introduced in an algebraic manner, by talking about functions over the integers.

In particular, one can introduce the notion of computable partial function, which extends the notion of computable functions to the case of non-total functions.

Definition 67 (Partial computable function) *Let $f: E \to F$ be a function, possibly partial.*

The function $f: E \to F$ is computable *if there exists a Turing machine A such that for any word w encoding an element $e \in E$ in the domain of f, the machine A on the input w, halts and accepts with the encoding of $f(e)$ written on its tape at the moment when it halts.*

Of course, this notion matches to the previous notion for the case of total functions.

One can characterize in a purely algebraic way the notion of computable functions:

Definition 68 (Recursive functions) *A (possibly partial) function $f : \mathbb{N}^n \to \mathbb{N}$ is* recursive *if it is either the constant* 0*, or one of the functions:*

- $\mathsf{Zero} : x \mapsto 0$ *the function 0;*
- $\mathsf{Succ} : x \mapsto x+1$ *the successor function;*
- $\mathsf{Proj}_n^i : (x_1, \dots, x_n) \mapsto x_i$ *the projection functions for* $1 \leq i \leq n$*;*
- $\mathsf{Comp}_m(g, h_1, \dots, h_m) : (x_1, \dots, x_n) \mapsto g(h_1(x_1, \dots, x_n), \dots, h_m(x_1, \dots, x_n))$ *the composition of the recursive functions* $g, h_1, \dots, h_m$*;*
- $\mathsf{Rec}(g, h)$ *the function defined by recurrence as*

  $$\begin{cases} f(0, x_2, \dots, x_n) = g(x_2, \dots, x_n), \\ f(x_1+1, x_2, \dots, x_n) = h(f(x_1, \dots, x_n), x_1, \dots, x_n), \end{cases}$$

  where g and h are recursive.
- $\mathsf{Min}(g)$ *the function that to* $(x_2, \dots, x_n)$ *associates the least* $y \in \mathbb{N}$ *such that*

  $$g(y, x_2, \dots, x_n) = 1$$

  if there is one (and which is not defined otherwise) where g is recursive.

A primitive recursive *function is a function that can be defined without using the schema* Min.

One can prove the following result:

Theorem 69 *A function $f : \mathbb{N}^n \to \mathbb{N}$ is recursive if and only if it is computable by some Turing machine.*

The notion of decidability or semi-decidability can then also be defined in an algebraic way:

Theorem 70 *A subset $S \subset \mathbb{N}$ is decidable if the characteristic function of S, i.e., the (total) function* $\chi : n \mapsto \begin{cases} 1 & \text{if } n \in S \\ 0 & \text{if } n \notin S \end{cases}$ *is recursive.*

Theorem 71 *A subset $S \subset \mathbb{N}$ is semi-decidable if the (partial) function*

$$n \mapsto \begin{cases} 1 & \text{if } n \in S \\ \text{undefined} & \text{if } n \notin S \end{cases}$$ *is recursive.*

Exercise 3 *(solution on page 235) Prove these theorems.*

8 Exercises

***Exercise 1** *(solution on page 235) [Generalized halting problem] Let A be a decidable subset of the set of encodings of Turing machines, such that all machines of A always halt.*

Then A is incomplete: There exists some (unary) total function $f : \mathbb{N} \to \mathbb{N}$ computable that is not represented by any Turing machine of A.

Explain why this result implies the halting problem.

Exercise 4 *(solution on page 235) Let $E \subset \mathbb{N}$ be a computably enumerable set enumerated by some computable function f strictly increasing. Prove that E is decidable.*

Exercise 5 *(solution on page 236) Deduce that any infinite computably enumerable set of $\mathbb{N}$ contains some infinite decidable subset.*

Exercise 6 *(solution on page 236) Let $E \subset \mathbb{N}$ be a decidable set. Prove that it can be enumerated by some computable function f strictly increasing.*

Exercise 7 *(solution on page 236) Let $A \subset \mathbb{N}^2$ be a decidable set of pairs of integers.*

Write $\exists A$ for the (first) projection of A, that is to say the subset of $\mathbb{N}$ defined by

$$\exists A = \{x | \exists y \in \mathbb{N} \textit{ such that } (x, y) \in A\}.$$

1. *Prove that the projection of a decidable set is computably enumerable.*

2. *Prove that any computably enumerable set is the projection of some decidable set.*

Exercise 8 *A real number a is called* computable *if there exist computable functions F and G from $\mathbb{N}$ to $\mathbb{N}$ such that for any $n > 0$ we have $G(n) > 0$ and*

$$\left| |a| - \frac{F(n)}{G(n)} \right| \leq \frac{1}{n}.$$

1. *Prove that any rational number is computable.*
2. *Prove that $\sqrt{2}, \pi, e$ are computable.*
3. *Prove that any real number $0 < a < 1$ is computable if and only there exists a computable radix 10 expansion of a, that is to say a computable function $H : \mathbb{N} \to \mathbb{N}$ such that for all $n > 0$ we have $0 \leq H(n) \leq 9$ and*
$$|a| = \sum_{n=0}^{\infty} \frac{H(n)}{10^n}.$$
4. *Prove that the set of computable reals is a countable sub-field of $\mathbb{R}$, such that any polynomial of odd degree has a root.*
5. *Give an example of a non-computable real.*

Exercise 9 *A "useless" internal state of a Turing machine is a state $q \in Q$ in which the machine never enters on any input. Formulate the problem to decide whether a given Turing machine has a useless state as a decision problem, and prove that it is undecidable.*

Exercise 10 *(solution on page 236) Consider the following decision problem: A Turing machine A is given, and one wants to determine*

1. *if $L(A)$ contains at least two distinct words*
2. *if $L(A)$ is empty*

Is the problem decidable? semi-decidable?

9 Bibliographic notes

Suggested readings To go further on the notions of this chapter, we suggest reading the books [Sipser, 1997] and[Hopcroft & Ullman, 2000] in English, or [Wolper, 2001], [Stern, 1994] [Carton, 2008] in French.

The book [Sipser, 1997] is very pedagogical.

Bibliography This chapter contains some standard results from computability. We used essentially their presentation in [Wolper, 2001], [Carton, 2008], [Jones, 1997], [Kozen, 1997], [Hopcroft & Ullman, 2000], as well as in [Sipser, 1997].

Index

Bibliography

[Carton, 2008] Carton, O. (2008). *Langages formels, calculabilité et complexité.*

[Dowek, 2008] Dowek, G. (2008). *Les démonstrations et les algorithmes.* Polycopié du cours de l'Ecole Polytechnique.

[Hopcroft & Ullman, 2000] Hopcroft, J. E. & Ullman, J. D. (2000). *Introduction to Automata Theory, Languages and Computation, Second Edition* (2nd ed.). Addison-Wesley.

[Jones, 1997] Jones, N. D. (1997). *Computability and complexity - from a programming perspective.* Foundations of computing series. MIT Press. `https://doi.org/10.7551/mitpress/2003.001.0001`

[Kozen, 1997] Kozen, D. (1997). *Automata and computability.* Springer Verlag. `https://doi.org/10.1007/978-1-4612-1844-9`

[Matiyasevich, 1970] Matiyasevich, Y. (1970). Enumerable sets are diophantine. *Doklady Akademii Nauk SSSR*, 191(2), 279–282.

[Sipser, 1997] Sipser, M. (1997). *Introduction to the Theory of Computation.* PWS Publishing Company.

[Stern, 1994] Stern, J. (1994). Fondements mathématiques de l'informatique. *Ediscience International, Paris.*

[Steyaert,] Steyaert, J.-M. Théorie des automates, langages formels, calculabilité. Cours de l'Ecole Polytechnique, Cours de l'Ecole Polytechnique.

[Wolper, 2001] Wolper, P. (2001). *Introduction à la calculabilité: cours et exercices corrigés.* Dunod.