3 – Computability

Luc Lapointe
luc.lapointe@ens-paris-saclay.fr
home.lmf.cnrs.fr/LucLapointe/

Abstract

The *theory of computability* aims to understand which computations are or will eventually be within the reach of our computers, and which will never be, even by our most powerful supercomputers.

1. Some variations on the Turing machine model

1.1. Many-tape Turing machines

Turing machines you have seen in previous lecture have one single tape. There exist some Turing machines with more than one tape.

Definition: Many-tape Turing machine

Let $k \in \mathbb{N}$. A **k-tape Turing machine** is a Turing machine with k memory tapes.

They have k different independent pointers, one for each cell. They all start at the left of their tape.

At the start of the computation, the input of the machine is only written on the first tape, not on the others.

Each transition reads one letter for each tape, write one letter on each tape, and moves pointer for each tape.

It happens that although this model seems more powerful, it actually is not.

Theorem: Many-tape machines are not more powerful

Let $k \in \mathbb{N}$ and L a language recognized by a k-tape Turing machine. Then there exist a 1-tape Turing machine recognizing L.

This theorem allows to show more easily that some languages are computable.

Exercise

Show that the language of palindromes on $\Sigma = \{0, 1\}$ is a computable language.

1.2. Non-deterministic machines

Just as finite automata, Turing machines can either be deterministic or not.

Definition: Non-deterministic Turing machine

A **non-deterministic Turing machine** is just the same as a Turing machine, except for its transition functions which is typed

$$\delta : Q \times \Sigma \to \mathcal{P}(\Sigma \times \{\leftarrow, \rightarrow, \downarrow\} \times Q).$$

It must ensure:

$$\forall q \in Q, \quad \delta(q, \$) \subseteq \{\$\} \times \{\rightarrow\} \times Q$$

Just as finite automata, the language of a non-deterministic Turing machine is the set of words with at least one possible computation accepting them.

Here we skip the formal definition of a non-deterministic Turing machine computation and accepted language, which is too heavy to be relevant.

Just as finite automata, non-deterministic Turing machines are not more powerful.

Theorem: Non-deterministic machines are not more powerful

Let L a language recognized by a non-deterministic Turing machine. Then there exist a deterministic Turing machine recognizing L.

Non-deterministic machines however compute things more efficiently. This will be the topic of next lecture.

2. Computable functions

Turing machines can not only recognize languages, they can also compute functions.

Computing a function works just the same as recognizing a word, except that the output is not only `accept` or `reject`, but rather what is written on the memory tape when the computation ends. Wether it ends on the `accept` or `reject` state does not matter.

What functions? We will only consider here functions from integers to integers. Theoretical computability of real numbers also exists, but it is harder to understand.

Encoding inputs and outputs To translate an integer $n \in \mathbb{N}$ into a sequence of characters, an *encoding* is needed. You are already familiar with decimal encoding: the integer 17 is also a sequence of characters, with digit `1` followed by digit `7`. Real-life machines read binary encodings: 17 is encoded by the sequence of bits `10001`. When it comes to Turing machines, you can use whatever encoding you like, although binary encoding often results in more concise machines.

Example The Turing machine in Figure 1 computes the function

$$n \mapsto n+1$$

where both its input and output are written in binary, with strong bit on the left.

States with an arrow aim at moving the machine pointer to the right, and the "cr" state computes carries.

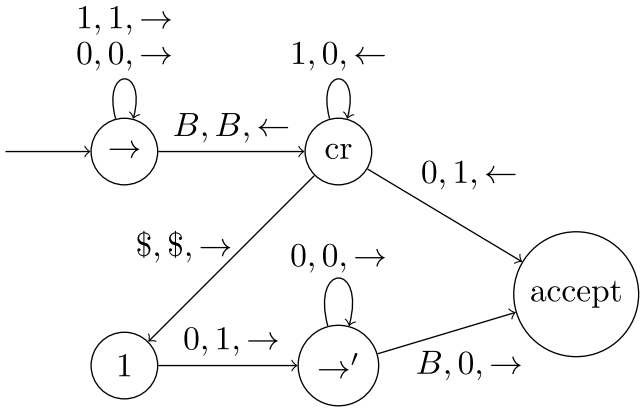

Figure 1: A Turing machine $\mathcal{M}_1$.

Definition: Computable function

Let $k \in \mathbb{N}$ an integer, $f : \mathbb{N}^k \to \mathbb{N}$ a function, and $b : \mathbb{N} \to \{0,1\}^*$ the usual binary encoding of integers. We say that f is **computable** if there exist a Turing machine $\mathcal{M}$ with a special # character in its alphabet and such that such that for all $(n_1, n_2, ..., n_k) \in \mathbb{N}^k$:

- When starting with $b(n_1)\#b(n_2)\#...\#b(n_k)$ on the tape, $\mathcal{M}$ ends its computation;
- When its computation ends, the tape contains only $b(f(n_1, ..., n_k))$.

In general, functions that can be computed by a program in an usual programming language are computable function. This is partly formalized by following theorem.

Theorem: Usual functions are computable

The following functions are computable:

- For all $n \in \mathbb{N}$, the function that maps nothing to n. They are often called "constants" rather than functions.
- Addition, subtraction, multiplication.
- Quotient or remainder of an Euclidean division.
- Exponential function. Logarithm, rounded to closest integer.
- A composition of computable functions is a computable function.

Just as for languages, different Turing machine models have the same computable functions.

Theorem: Same computable functions

Let $n \in \mathbb{N}$ and $f : \mathbb{N}^k \to \mathbb{N}$ a function.

$$\begin{aligned} & \text{There exist a many-tape Turing machine computing } f \\ \Leftrightarrow\ & \text{There exist a non-deterministic machine computing } f \\ \Leftrightarrow\ & \text{There exist a Turing machine computing } f \end{aligned}$$

3. From languages to problems

We call *problems* questions with an input/output format, such as the following:

Input: A map M, two places x and y on this map.
Question: What is the shortest path from x to y on this map?

or

Input: A program P.
Question: Is P buggy?

We would like to build a formal definition that models such questions. This can be achieved through *languages* and *encodings*.

3.1. Alphabets + encodings = (almost) everything

Here, *encoding* data means: describing complex data with a simpler alphabet.

- One example of simple encoding is the encoding of $\mathbb{N}$ with $\Sigma = \{0, 1, ..., 9\}$ through decimal writing of integers.
- Another example is encoding programs source code through everyday alphabets $\Sigma = \{a, ..., z\} \cup \{0, ..., 9\}$ along with some special characters such as `;` or `#`.
- On a more general point of view, every data inside modern computers, including pictures, music, text... is encoded with the binary alphabet $\Sigma = \{0, 1\}$.

Not every word is an encoding When using encodings, some combinations of letters of an alphabet can have a meaning, and some other not. e.g. both

```
if true then return x
```

and

```
x nruter neht eurt fi
```

are written with letters of everyday alphabet, but only one is the encoding of a program, and the other does not have any meaning. This does not mean *at all* that using everyday letters is a bad way to encode programs.

3.2. Problems as languages of encodings

Some problems only accept binary answers, as *True* or *False*, or *Yes* or *No*. Example given:

Input: A program P.
Question: Is P buggy?

They are called *Decision problems*.

Definition: Decision problem

A **decision problem** is a problem with only two possible answers. Without loss of generality, we consider they are *True* or *False*.

Decision problems can be simply modelled as *languages* of encodings, such as:

The language of encodings of buggy programs.

Definition: Decision problem, version 2

A **decision problem** can be defined as a language P on some alphabet Σ^*.
Words of P are called **positive instances**, words on Σ^* are called **instances**.

Note that there is no mention of encodings in this definition. Encodings are indeed not needed to give a formal meaning of *problems*. They are just a tool that makes this definition able to model real-life problems.

Instances is an other word for *input*, and *positive instances* for *answers*.

Example The problem

Input: An integer $n \in \mathbb{N}$.
Question: Is n an even number?

Can be modelled by the set

$$\{0, 1, ..., 9\}^* \cdot \{0, 2, 4, 6, 8\}$$

Definition: Decidable decision problem

A decision problem is **decidable** if it is the language of a Turing machine.

3.3. Problems as functions of encodings

On some other problems, outputs and inputs are different objects. Example given:

Input: A map M, two places x and y on this map.
Question: What is the shortest path from x to y on this map?

They are called *Optimization problems*.

Definition: Optimization problem

An **optimization problem** is a problem with more than two different possible answers. It is often a quantity that must be maximized or minimized.

Optimization problems can be modelled as *functions* on encodings. Example given:

f : Encodings of maps and two places $\to$ Encodings of paths

$M, x, y \mapsto$ Encoding of shortest path from x to y in M

Definition: Optimization problem, version 2

An **optimization problem** can be defined as a function P from some alphabet Σ_I^* to some other alphabet Σ_O^*. Inputs of P are called **instances**.

Definition: Decidable optimization problem

An optimization problem is **decidable** if it is a computable function.

4. Undecidability

Turing machines model computers with unbounded memory. They can also take any time they want before answering the question they are asked, therefore reaching some sort of unlimited computation power. But yet, not every problem is decidable.

Idea of the proof Remember that in essence, a problem is *decidable* if there exist a machine answering it. Finding our very first undecidable problem will be done with a proof by contradiction.

The key of the proof is to use the hypothetical machine answering to the problem on itself. This is called a *diagonal argument*, and can only be achieved if the problem we are dealing with can take (encodings of) machines as inputs.

We will achieve this proof with the so-called *Halting problem*.

4.1. The Halting problem is undecidable

Definition: Halting problem

The following problem is called **Halting problem**.

Input: A machine $\mathcal{M}$, an input x.
Question: Does $\mathcal{M}$ end its computation on x?

Consider there exist a machine $\mathcal{M}_{\text{Halt}}$ whose language is exactly the (encoding of) machines $\mathcal{M}$ and inputs x such that $\mathcal{M}$ ends its computation on x.

Now consider the following `trouble` program that takes Turing machines as inputs:

```
trouble(M):
    if M end its computation on M:
        loop forever
    else:
        end computation
```

As $\mathcal{M}_{\text{Halt}}$ can decide wether a machine $\mathcal{M}$ end its computation on or not on some given input, this program can be modelled by a Turing machine $\mathcal{M}_{\text{trouble}}$. Now, does $\mathcal{M}_{\text{trouble}}(\mathcal{M}_{\text{trouble}})$ end its computation?

- If $\mathcal{M}_{\text{trouble}}(\mathcal{M}_{\text{trouble}})$ end its computation, then according to the `if` test it should loop forever, thus never ending its computation. Contradiction $\lightning$
- If $\mathcal{M}_{\text{trouble}}(\mathcal{M}_{\text{trouble}})$ does not end its computation, then according to the `if` test it should jump to the `else` case, thus ending its computation. Contradiction $\lightning$

This concludes the proof.

Theorem

The Halting problem is undecidable.

An important note on undecidability Knowing that a problem is undecidable means: we do not have any automated method that works on every possible inputs. It does **not** mean that we have no answers on any input. We can still find local proofs on some instances. One example here is that we know some Turing machines end their computations, even though the halting problem is undecidable. You will find some other similar examples in Section 4.2.

A second important note on undecidability Undecidability relies ***a lot*** on ***inputs*** of the problem. Consider for example a fixed Turing machine $\mathcal{M}$, a fixed word x, and the following fake halting problem:

Input: Nothing.
Question: Does $\mathcal{M}$ end its computation on x?

This function is either the function with 0 inputs that always answer *True*, or the function with 0 inputs that always answer *False*. In either case it is computable, so this fake halting problem is decidable.

Exercise: Look carefully

Two of the following problems are decidable, and the decidability of the third one is a famous open problem. Find them.

Input: An integer $n \in \mathbb{N}$.
Question: Do the Navier-Stokes equations always have smooth solutions?

Input: An integer $n \in \mathbb{N}$.
Question: Do π have a sequence of n consecutive 1 in its decimal writing?

Input: An integer $n \in \mathbb{N}$.
Question: Do π have a sequence of n consecutive 1 surrounded by characters different than 1 in its decimal writing?

Hint: No knowledge on π is necessary.

4.2. Some undecidable problems

Here is a short list of useful but undecidable problems. Many more are known, but most of them have quite technical statements.

Automatic proving Deciding wether a mathematical property is true or not by simply giving it to an automated prover would be a tremendous breakthrough in all mathematical fields, and probably most scientific fields.

Input: A mathematical property P.
Question: Is P true?

Grammar ambiguity, Grammar equality We say that a grammar is *ambiguous* if one given word can be generated by two different trees.

Input: A grammar G.
Question: Is G ambiguous?

Input: Two grammars G_1 and G_2.
Question: $\mathcal{L}(G_1) = \mathcal{L}(G_2)$?

Those questions are interesting having an automatic answer when conceiving new grammars for existing or new programming languages. A non ambiguous grammar is considered better, and editing a working grammar of a programming language should generate the same programs.

Game of life See Conway's game of life rules on the internet if you do not know them yet.

Input: An initial game of life position.
Question: Will this position appear again later?

4.3. Rice's Theorem

Rice's theorem is a bulldozer theorem that annihilates any hope of automated provers of program properties. Here is a first intuitive statement of the theorem.

Rice's Theorem

All non-trivial semantic properties of programs are undecidable.

A *trivial property* of programs is a property that is either true on all programs or false on all programs.

A *semantic* property is a property that describes the result of the program.

A more mathematically precise version of the theorem is given by the following definitions.

Definition: Property of programs

A **property of programs** P is a subclass of all Turing machines $\mathcal{M}$. We say that machines in P verify the property, and machines outside of it do not.

Definition: Semantic property

A property of programs is a *semantic property* if, for any pair of Turing machines $\mathcal{M}_1$ and $\mathcal{M}_2$,

$$\mathcal{L}(\mathcal{M}_1) = \mathcal{L}(\mathcal{M}_2) \implies (\mathcal{M}_1 \in P \Leftrightarrow \mathcal{M}_2 \in P)$$

In other words, if two Turing machines have the same languages, if one does (not) verify the property, then so does the other.

Example "This Turing machine ends in 10 steps" is *not* a semantic property. "This Turing machine accepts the empty word" is.

Definition: Non-trivial property

A property of programs P is a **non-trivial property** if there exist $\mathcal{M}_1$ and $\mathcal{M}_2$ such that $\mathcal{M}_1 \in P$ and $\mathcal{M}_2 \notin P$.

Rice's Theorem consequences A major consequence of this theorem is that, as bugs of programs are a semantic property, there can be no automated bug-finder that works on all programs.

Improvise. Adapt. Overcome. Current research results circumvent this theorem by designing automated bug-finders on limited programs, or by designing some that might miss bugs.

5. Reductions

Reductions are a key tool in the theory of computability. They consist in re-encoding one problem into the other, and allow to transfer some (un)decidability results from one problem to another.

Why talking about reductions? Most undecidability proofs are achieved through a reduction from another undecidable problem. Also, this tool is used a lot not only in the theory of computability, but also in the theory of complexity, which will be the topic of next lecture.

We define reductions only on decision problems[1]. This already covers a wide range of problems, as it covers every question of the form:

Does *this object* have *this property*?

[1]Remember, they are problems whose output is either *True* or *False*.

Definition: Reduction

Let $\mathcal{P}_1$ and $\mathcal{P}_2$ two decision problems. A **reduction from $\mathcal{P}_1$ to $\mathcal{P}_2$** is a **computable** function f that maps inputs of $\mathcal{P}_1$ into inputs of $\mathcal{P}_2$, and such that

$$x \text{ is a positive instance of } \mathcal{P}_1 \Leftrightarrow f(x) \text{ is a positive instance of } \mathcal{P}_2$$

Notation $\mathcal{P}_1 \preccurlyeq \mathcal{P}_2$ means that there exist a reduction from $\mathcal{P}_1$ to $\mathcal{P}_2$. We also say that $\mathcal{P}_1$ *reduces to* $\mathcal{P}_2$, or that $\mathcal{P}_2$ *is harder than* $\mathcal{P}_1$.

What does a reduction look like? Here is a reduction from the halting problem to a variant of it, that looks simpler to decide. We call this new problem Halt_ε.

Input: A machine $\mathcal{M}$.
Question: Does $\mathcal{M}$ end its computation on ε?

Building the reduction We want to build a *computable* function that maps inputs of the halting problem to inputs of this new problem. So our function r should map pairs $(\mathcal{M}, x)$ to machines $\mathcal{M}'$. Furthermore, $\mathcal{M}$ should end on x if and only if $\mathcal{M}'$ ends on ε.

A key detail here is that $\mathcal{M}'$ *depends on* $\mathcal{M}$ *and* x. So there can be one new machine $\mathcal{M}'$ for *each* pair $\mathcal{M}$ and x.

We suggest here a machine described by the pseudo-code in Listing 1.

```
M'(y):
    erase input y and write B characters instead
    move pointer to the left of the tape
    write x on the left of the tape
    move pointer to the left of the tape
    mimic the machine M
```

Listing 1: Pseudo-code of the machine $\mathcal{M}' = r(\mathcal{M}, x)$.

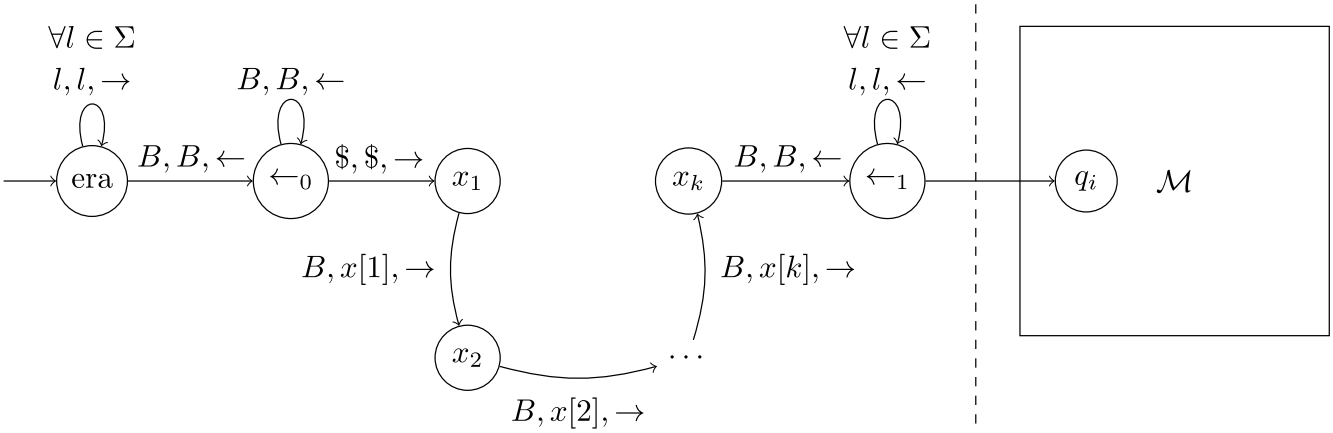

Figure 2: Drawing of the machine $\mathcal{M}' = r(\mathcal{M}, x)$.

Is it computable? We want to check wether a function that maps (the encoding of) a Turing machine $\mathcal{M}$ and an input x to the machine $\mathcal{M}'$ described in Figure 2 can be computed by a Turing machine or not. We call R the machine that computes this reduction r. **Attention**, the machine computing the reduction is *not the same* as machines $\mathcal{M}$ and $\mathcal{M}'$!

- The first four lines in Listing 1 are described by the part of the machine left of the dashed line in Figure 2. In particular, it does not depend on the input y of $\mathcal{M}'$ – but it depends on input x of R.

  So as a first step in writing (the encoding of) $\mathcal{M}'$, the machine R can simply write (the encoding of) the left part of the drawing. This is computable, as this is "only" a very long word depending on its input x.
- The last line of $\mathcal{M}'$ can be achieved by simply acting as $\mathcal{M}$. The machine $\mathcal{M}'$, after ending its computation of the first four lines, can simply connect to the initial state of $\mathcal{M}$.

  Copying $\mathcal{M}$ right after the beginning of $\mathcal{M}'$ can also be computed by R, as $\mathcal{M}$ is an input of R, so R should simply copy its input.

This concludes that r is indeed computable. ■

Equivalence of positive instances Let $\mathcal{M}, x$ such that $\mathcal{M}$ ends its computation on x. Then $\mathcal{M}'$ also ends its computation on ε, as the only part of $\mathcal{M}'$ that could not finish is the execution of $\mathcal{M}$ on x.

Conversely, if $\mathcal{M}$ does not end its computation on x, then M' should also never end its computation on ε. ■

We can consequently conclude that there exist a reduction from the halting problem to Halt_ε.

Exercise: An easier reduction

Describe a reduction from Halt_ε to the halting problem.

It is the following theorem that justifies the use of the notation $\preccurlyeq$ and that when $\mathcal{P}_1 \preccurlyeq \mathcal{P}_2$, we say that $\mathcal{P}_2$ is *harder* than $\mathcal{P}_1$.

Theorem: Use of reductions

Let $\mathcal{P}_1$ and $\mathcal{P}_2$ two decision problems such that there exist a reduction f from $\mathcal{P}_1$ to $\mathcal{P}_2$. Then:
- If $\mathcal{P}_2$ is decidable, $\mathcal{P}_1$ also is.
- If $\mathcal{P}_1$ us undecidable, $\mathcal{P}_2$ also is.

Proof The second mark is the contraposition of the first one. As such, proving only the first one is enough.

We prove the first mark by building a *computable* function *that solves* $\mathcal{P}_1$.

Let `f` a reduction from $\mathcal{P}_1$ to $\mathcal{P}_2$. Assume that there exist a computable function `solveP2` that answers `True` on instances of $\mathcal{P}_2$ if and only if they are positive instances of $\mathcal{P}_2$. Consider the following program to solve $\mathcal{P}_1$:

```
solveP1(x):
    return solveP2(f(x))
```

The reduction `f` is by definition a computable function, as well as `solveP2`. `solveP1` is thus also a computable function. Moreover, it indeed solves problem $\mathcal{P}_1$:

$$
\begin{aligned}
& x \text{ is a positive instance of } \mathcal{P}_1 \\
\Leftrightarrow\ & f(x) \text{ is a positive instance of } \mathcal{P}_2 \\
\Leftrightarrow\ & \texttt{solveP2} \text{ answers } \texttt{true} \text{ on } \texttt{f(x)} \\
\Leftrightarrow\ & \texttt{solveP1} \text{ answers } \texttt{true} \text{ on } \texttt{x}
\end{aligned}
$$

First equivalence is because f is a reduction from $\mathcal{P}_1$ to $\mathcal{P}_2$. Second equivalence is because `solveP2` solves $\mathcal{P}_2$. Third equivalence comes from the construction of `solveP1`. ■

Example The problem Halt_ε is undecidable, because the halting problem also is, and there exist a reduction from it to Halt_ε.