3. COMPUTING

Mihai-Valentin DUMITRU
mihai.dumitru2201@upb.ro

October 2024

In the introductory lecture, we discussed what "problems" are and classified them into two classes:

- *"decision problems"* – those which admit a "yes" or "no" answer
- *"function problems"* – those whose answer is some mathematical object other than a boolean, such as a number, a set, a graph etc.

In the second lecture, we introduced the concept of *Turing Machines* and presented some particular examples of machines that solve problems. The answer to a decision problem was given by transitioning to one of the final states Y (a "yes" answer) or N (a "no" answer), while the answer to a function problem was given by encoding the answer in the tape contents and transitioning to the final state H.

In all cases mentioned above, the computation ends by reaching a final state. But consider the following machine:

	0	1	$\square$
q_1	$q_1, 0, -$	$q_1, 1, -$	$q_1, \square, -$

The machine starts in the initial state q_1. Regardless of what symbol it reads, it remains in the same state, keeping the head stationary. Its computation will never end: it can never reach any of the three final states.

This should evoke the more familiar concept of `while (1) {}` from a language like C. The machine performs work: it goes through successive discrete stages, each time transitioning to a state, writing a symbol on the tape and moving the tape head. It just so happens that the next state is always the same as the current one, the written symbol is always the same as the read one and the movement direction of the head is to keep it stationary, so there is no interesting effect for any transition.

However, if the machine above seems too odd, here's one that moves the head and changes the contents of the tape:

	0	1	$\square$
q_1	$q_1, 1, \rightarrow$	$q_1, 1, \rightarrow$	$q_1, 1, \rightarrow$

Regardless of the input, the machine will go on forever, writing one more 1 on the tape at each step. By analogy with C, you can think of it as: `while (1) { putchar("1"); }`.

Here is one more example of a machine that goes on forever, but also changes state:

	0	1	$\square$
q_1	$q_1, 1, \rightarrow$	$q_1, 0, \rightarrow$	$q_2, \square, \leftarrow$
q_2	$q_2, 1, \leftarrow$	$q_2, 0, \leftarrow$	$q_1, \square, \rightarrow$

This machine keeps going left-to-right, then right-to-left through the input, flipping symbols.

The three examples above always run forever, regardless of the input. But we can easily conceive of a machine that halts for some inputs and doesn't for others:

	0	1	$\square$
q_1	$q_1, 0, H$	$q_2, 1, \rightarrow$	$q_2, 1, \rightarrow$
q_2	$q_2, 1, \rightarrow$	$q_2, 1, \rightarrow$	$q_2, 1, \rightarrow$

This one halts in one transition for any input that begins with a 0, or goes on to write a never-ending sequence of 1s for any input that begins with 1, as well as for the empty input.

The key takeaway here is that, during its run, a machine can either make a finite number of transitions until it reaches a final state, or make an infinite number of transitions, never reaching a final state. In the first case, we say that the machine *"halts"*, in the other that it *"doesn't halt"*.

We will now formalize the notion of a machine's *"run"*.

1 Well-behaved machines and tape contents

To keep our definitions short and sweet, we will adopt the following convention: *a Turing machine will always maintain the contents of the tape as a string of non-blank symbols surrounded on both sides by infinite sequences of blanks.*

This means that a machine isn't allowed to overwrite any non-blank symbol with a blank; except for the first and last symbols.

A machine is also not allowed to "leave a gap": go over some blanks to the left or to the right of the content string, then insert a non-blank symbol there.

Finally, because of the above restrictions, there is no point to advance more than one blank symbol in any direction: if a Turing machine moves left onto a blank symbol, it can then only move back right, or stay and keep transitioning until it overwrites the blank symbol with a non-blank symbol (similarly for moving right onto a blank symbol).

We will use the term *"tape contents"* for the contiguous non-blank portion of the tape.

A well-behaved machine cannot end its computation on a blank symbol, if there are any tape contents; only if the entire tape is filled with blanks. This will simplify a later definition about what it means to compute an output.

From now on, we shall discuss only such "well-behaved" machines.

2 Turing Machine configurations

At any given *step* of computation, the following is true:

- the tape contains a finite string of non-blank cells
- the head is scanning exactly one cell
- the machine is in some state $\mathbf{q}$

These three elements together (tape contents, head position, state) uniquely characterize a computation step – together, they form a *"configuration"*.

Definition 3.1. A *configuration* of a Turing Machine $M = (Q, \Sigma, \Gamma, B, q_1, \delta)$ is a triplet $(\gamma_L, \gamma_R, \mathbf{q})$ where:

- γ_L is a string of non-blank symbols ($\gamma_L \in \Gamma'^*$) with the contents of the tape, *to the left of the head.*
  The string is *empty* if there are no non-blank symbols to the left of the head. For example, this is how all machines start.
  Note that the character currently read by the head **is not** part of this string.
- γ_R is a string of symbols ($\gamma_R \in \Gamma^*$) with the contents of the tape, *to the right of the head.*
  All the symbols of γ_R, except for the first one, are non-blank. The first symbol is a blank only in the following two situations:
  - the head of the machine is currently one position left of the tape contents
  - the head of the machine is currently one position right of the tape contents; in this case $\gamma_R = B$

  Note that the character currently read by the head **is** part of this string.
- $\mathbf{q} \in Q'$ is the current state.

2.1 Configuration types

Definition 3.2. The *initial configuration of a Turing machine* $M = (Q, \Sigma, \Gamma, B, q_1, \delta)$ *on input* $w \in \Sigma^*$ is:

- (ε, w, q_1), for $w \neq \varepsilon$
- (ε, B, q_1), for $w = \varepsilon$

We shall call it $C_0^M(w)$.

Definition 3.3. A *halting configuration of Turing machine* $M = (Q, \Sigma, \Gamma, B, q_1, \delta)$ *on input* $w \in \Sigma^*$ is (ϕ, μ, q_f), where $\phi \in \Gamma'^*, \mu \in \Gamma^*, q_f \in \{Y, N, H\}$.
Specifically, an *accepting configuration* is one where $q_f = Y$ and a *rejecting configuration* is one where $q_f = N$.

2.2 Configurations example

Recall the machine `isEven` from the previous lecture and let's consider its run over input `10010`.

The first configuration, at the very beginning, before doing any computational step is:

$$(\varepsilon, 10010, q_1)$$

The third element of the tuple tells us that the machine is in state q_1. There are no symbols to the left of the head, hence the first element of the tuple is just the empty string, ε. The whole input is to the right of the head, except for the first symbol which is right under it. To figure out the symbol currently read by the head, we need to inspect the first symbol of the second element of the tuple; here, it is 1.

Looking at the transition table of `isEven`, we know that $\delta(q_1, 1) = (q_1, 1, \rightarrow)$. So the next configuration is:

$$(1, 0010, q_1)$$

The state is still q_1, but the head moved one position right, so the first 1 symbol of the input is now to the left of the head, which is positioned over a 0.

The machine then goes through the following configurations, until it transitions into a halting state.

$$(10, 010, q_1)$$
$$(100, 10, q_1)$$
$$(1001, 0, q_1)$$
$$(10010, \square, q_1)$$
$$(1001, 0, q_2)$$
$$(1001, 0, Y)$$

3 Going from one configuration to another

Definition 3.4. Let $M = (Q, \Sigma, \Gamma, B, q_1, \delta)$ be a Turing Machine.

We say that $C \vdash_M C'$ (read "from configuration C, machine M goes to configuration C' in one step") if, in a single transition, the machine goes from configuration C to configuration C'.

Depending on the transition function δ, the following situations are possible (for ease of reading, we shall use bolded Latin letters, like $\mathbf{p}$ to represent *states*, monospaced Latin letters from the beginning of the alphabet, like $\mathtt{a}$, to represent *tape symbols* and Greek letters to represent *strings of tape symbols*):

1. $\delta(\mathbf{p}, \mathtt{a}) = (\mathbf{q}, \mathtt{b}, -)$:
   1.1. $(\phi, \mathtt{a}\mu, \mathbf{p}) \vdash_M (\phi, \mathtt{b}\mu, \mathbf{q})$
2. $\delta(\mathbf{p}, \mathtt{a}) = (\mathbf{q}, \mathtt{b}, \rightarrow)$:
   2.1. $(\phi, \mathtt{ac}\mu, \mathbf{p}) \vdash_M (\phi\mathtt{b}, c\mu, \mathbf{q})$
   2.2. $(\phi, \mathtt{a}, \mathbf{p}) \vdash_M (\phi\mathtt{b}, \square, \mathbf{q})$
3. $\delta(\mathbf{p}, \mathtt{a}) = (\mathbf{q}, \mathtt{b}, \leftarrow)$:
   3.1. $(\phi c, \mathtt{a}\mu, \mathbf{p}) \vdash_M (\phi, \mathtt{cb}\mu, \mathbf{q})$
   3.2. $(\varepsilon, \mathtt{a}\mu, \mathbf{p}) \vdash_M (\varepsilon, \square\mathtt{b}\mu, \mathbf{q})$

Where:

- $\phi, \mu \in \Gamma'^*$ (possibly-empty strings of non-blank tape symbols)
- $\mathtt{a}, \mathtt{b}, \mathtt{c} \in \Gamma$
- $\mathbf{q}, \mathbf{s} \in Q$

We abuse notation here and use a specialized version of concatenation where concatenating a blank to the end of the empty string results in the empty string: $\varepsilon\square = \varepsilon$.

Combined with our specification of "well-behaved machines", this convention keeps our rules succinct (consider the specifications for the transitions that move the head right).

Remember that our machines are *well-behaved* and never advance more than one blank symbol outside the bounds of the tape contents.

3.1 Zero or more steps

We now define the operator $\vdash_M^*$ to be the *reflexive and transitive closure* of $\vdash_M$. This means that we extend this operator to yield a configuration "in 0 or more steps", exhibiting the two properties:

- **reflexivity**: For any configuration C: $\quad C \vdash_M^* C$
- **transitivity**: For any configurations C_1, C_2, C_3: $\quad (C_1 \vdash_M^* C_2) \wedge (C_2 \vdash_M^* C_3) \implies C_1 \vdash_M^* C_3$

We now give formal definitions for "the result" of running a machine M on an input w.

4 How Turing Machines run

In all definitions in this section:

- M is a Turing Machine $(Q, \Sigma, \Gamma, B, q_1, \delta)$
- $w, v \in \Sigma^*$
- $\phi \in \Gamma'^*$
- $\mu \in \Gamma^*$ (only its first symbol may be a B)

Definition 3.5. $M[w] \rightarrow TRUE \stackrel{\text{def}}{=} C_0^M(w) \vdash_M^* (\phi, \mu, Y)$

Read this as: *"M accepts input w"*.

Note 3.1. In many Computability textbooks, you may see this denoted by $M(w) = TRUE$, but we want to emphasize here that M is not a *function* that *maps* w to a value, but a **machine** that **performs computation** starting from w.

Definition 3.6. $M[w] \to FALSE \stackrel{\text{def}}{=} C_0^M(w) \vdash_M^* (\phi, \mu, N)$

Read this as: *"M rejects input w"*.

Definition 3.7. $M[w] \to v \stackrel{\text{def}}{=} C_0^M(w) \vdash_M^* (\phi, \mu, H)$ and:

- $\phi\mu = v$, if $v \neq \varepsilon$
- $\phi = \varepsilon, \mu = B$, if $v = \varepsilon$

Read this as: *"on input w, M computes the output v"*.

This definition tells us that M computes a string v, if it eventually transitions into state H with the tape contents (from the first non-blank, to the last non-blank) consisting of that string.

Remember that a well-behaved machine cannot end its computation on a blank symbol, unless the tape is completely blank. The second bullet covers that case.

Definition 3.8. $M[w]$ *halts* $\stackrel{\text{def}}{=} M[w] \to X \vee M[w] \to \text{TRUE} \vee M[w] \to \text{FALSE}$
where $X \in \Gamma'^*$

This is a generalization of the previous three definitions. If, on input w, M accepts or rejects or computes an output v, we can say generically that *"M halts on input w"*. Otherwise we say that *"M doesn't halt on input w"*.

Going forwards, it will be useful for us to have a short notation for this case:

Definition 3.9. $M[w] \to \perp \stackrel{\text{def}}{=} M[w]$ doesn't halt

(The symbol $\perp$ is called a *"bottom"* and its usage here is a bastardization of its meaning in *Type Theory*.)

5 Accepting a problem

Definition 3.10. Let $f : \Sigma^* \to \{FALSE, TRUE\}$ be a decision problem and $M = (Q, \Sigma, \Gamma, B, q_1, \delta)$ a Turing Machine:
M accepts $f \stackrel{\text{def}}{=} \forall w \in \Sigma^*, f(w) = TRUE \Leftrightarrow M[w] \to TRUE$

Note that the definition doesn't say anything about the case in which $f(w) = FALSE$. The machine could transition to final state N, or H, or it could loop forever, it doesn't matter.

Definition 3.11. f is acceptable $\stackrel{\text{def}}{=} \exists M$ s.t. M accepts f

We can now define our first important set of decision problems:

Definition 3.12. $RE \stackrel{\text{def}}{=} \{f \mid f \text{ is acceptable } \}$

The name "RE" stands for "recursively-enumerable", which is another term for "acceptable". We use it here (instead of something like A) because it's the classical name for this set.

6 Deciding a problem

Definition 3.13. Let $f : \Sigma^* \to \{FALSE, TRUE\}$ be a decision problem and $M = (Q, \Sigma, \Gamma, B, q_1, \delta)$ a Turing Machine:
M decides $f \stackrel{\text{def}}{=} \forall w \in \Sigma^*, (f(w) = TRUE \Leftrightarrow M[w] \to TRUE) \wedge (f(w) = FALSE \Leftrightarrow M[w] \to FALSE)$

Definition 3.14. f is decidable $\stackrel{\text{def}}{=} \exists M$ s.t. M decides f

Note that for a problem to be *decidable*, there has to exist a Turing machine M which *always halts* and gives the correct answer, whatever the input. M matches our intuition of what a *"decision algorithm"* is: a *finite mechanistic procedure* that *always* provide an answer to a yes/no question, in *a finite amount of time.*

Definition 3.15. $R \stackrel{\text{def}}{=} \{f \mid f \text{ is decidable } \}$

The name "R" stands for "recursive", which is another term for "decidable". Like RE, it's a classic name.

At this point in the lecture, the concept of acceptance might seem artificial and meaningless. But we shall soon prove (and provide examples) that there are decision problems for which there exist machines that can *accept* them but there exist no machines that can *decide* them. Thus, acceptance is a notion worthy of study.

What's more, we will also prove (and provide examples) that there are decision problems which are not acceptable: no Turing Machine exists that can accept them.

7 Computing a problem

We move on from decision problems for a bit, and generalize the concept of "deciding".

Definition 3.16. Let $f : \Sigma_1^* \to \Sigma_2^*$ be a function problem and $M = (Q, \Sigma_1, \Gamma, B, q_1, \delta)$ a Turing Machine:
M computes $f \stackrel{\text{def}}{=} \forall w \in \Sigma_1^*, M[w] \to f(w)$

Definition 3.17. f is computable $\stackrel{\text{def}}{=} \exists M$ s.t. M computes f

Note that, per our definitions, decidability is a special case of computability, applicable only to decision problems.

Similarly to what we said about acceptable and decidable problems, we will soon prove that there are some function problems that are not computable.

Note 3.2. From now on, when talking generically about functions, we will use the informal term "*solve*" to mean either "accept", "decide" or "compute", appropriately, based on context.

8 The Church-Turing thesis

We now formulate a very important (and possibly non-intuitive) claim that is essential to the relevance of our approach to Computability Theory: the Church-Turing thesis. One possible formulation of the thesis is:

"The problems that are solvable by a mathematician with pen and paper are precisely those computable by a Turing Machine."

We can rephrase it in a way more relevant to our goals:

"The problems that are solvable by an algorithm are those that are computable by a Turing Machine."

This is a philosophical statement about reality, that asserts a correspondence between the intuitive concept of "algorithm" and the formal concept of "Turing machine". Because "algorithm" (or "solvable by an algorithm") is not a formally defined mathematical object, the Church-Turing thesis **is not** a mathematical statement; thus it can be neither proven or disproven within the realm of mathematics.

However, it can be analyzed, debated, doubted, accepted etc. At this point, you might find little reason to believe this assertion that Turing Machine can solve any problem solvable by an algorithm, but there are some solid arguments in support of it.

8.1 Reasons to believe

8.1.1 Different approaches, same result

At the beginning of the twentieth century, what algorithms are and what can actually be solved with an algorithm became an important subject. This led to the idea of an "effective method", a recipe of steps to carry out a problem. The characteristic of an effective method are:

- It consists of a finite number of instructions, each expressible with a finite amount of symbols.
- When carried out, it will produce a result in a finite number of steps.
- It can be carried out by a human with just pencil and paper.
- It demands no insight, intuition or ingenuity.

A problem that can be solved by an effective method is "effectively calculable". Note that this is an intuitive, informal concept.

There are at least three contemporaneous attempts at formalizing this notion:

- recursive functions
- the Lambda calculus
- Turing Machines

These three widely different models all started as an attempt to rigorously capture which problems are "effectively calculable". They were all shown to be equivalent to each other; that is, they identify precisely the same set of problems.

Since then, many other models of computations (e.g. counter machines, RAM machines, Markov algorithms) were designed for which it was proven that the set of problems *computable in that model* is R.

8.1.2 No known counterexamples

One way to reject the hypothesis would be to provide a counterexample of a problem that can be solved algorithmically, but for which no Turing Machine that decides it exists.

The examples from the previous lecture, as well as the exercises in the first lab, should convince you that there's quite a wide range of problems that these machines can decide. The literature on computability theory is filled with countless other examples of Turing Machines deciding various functions, as well as methods for proving that a Turing Machine which decides a function exists (i.e. even without explicitly presenting such a machine). †

9 References and further reading

The "recursive functions" mentioned in subsubsection 8.1.1 were formulated by Kurt Gödel (1906-1978) during a series of lectures at Princeton. The lecture notes can be found, for example, in "Kurt Gödel: Collected Works: Volume I: Publications 1929-1936" [1].

The Lambda calculus was developed by Alonzo Church (1903-1995) and his students. In his 1936 paper "An Unsolvable Problem of Elementary Number Theory" [2], Church set out to formalize the notion of "effective-calculability" and prove the existence of undecidable problems. He also showed the equivalence between lambda calculus and Gödel's recursive functions.

†There is a branch of Computability Theory that deals with *"hypercomputation"* – computational models which can compute functions that Turing Machines cannot. However, while the models and results are mathematically sound, the computation they describe could not be carried out by a human with pen and paper. For example, some models require performing an infinite number of steps in a finite amount of time.

Alan Turing described his machines in 1936, in "On computable numbers, with an application to the Entscheidungsproblem" [3] (which was only published in 1937). Because Church's paper had been published one year before, Turing added an appendix to his own, proving the equivalence between lambda-definable functions and Turing-computable functions.

The Stanford Encyclopedia of Philosophy has an entry on the Church-Turing thesis [4]: its historical context, what it claims and what it doesn't, as well as a more detailed section on why to accept it. [‡]

Bibliography

[1] Kurt Gödel. *Kurt Gödel: Collected Works: Volume I: Publications 1929-1936*. Vol. 1. Oxford University Press, New York, 1986, pp. 338–371.

[2] Alonzo Church. "An unsolvable problem of elementary number theory". In: *American Journal of Mathematics* 58.2 (1936), pp. 345–363.

[3] Alan Mathison Turing et al. "On computable numbers, with an application to the Entscheidungsproblem". In: *J. of Math* 58.345-363 (1936), p. 5.

[4] B. Jack Copeland. "The Church-Turing Thesis". In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Summer 2020. Metaphysics Research Lab, Stanford University, 2020.

[‡] `https://plato.stanford.edu/entries/church-turing/`