CENG 280 - Chapter 5: Undecidability

Burak Metehan Tunçel - June 2022

1 The Church Turing Thesis

The main question was "What can be computed?" (And, more intriguingly, "What cannot be computed?"). Various and diverse mathematical models of computational processes that accomplish concrete computational tasks (in particular, *decide*, *semidecide*, or *generate languages*, and *compute functions*) are introduced. By trying to formalize our intuitions on which numerical functions can be considered computable, we defined a class of functions that turned out to be precisely the recursive ones.

All this suggests that a natural upper limit on what a computational device can be designed to do has been reached; that search for the ultimate and most general mathematical notion of a computational process, of an *algorithm*, has been concluded successfully and the *Turing machine* is the right answer. However, we have also seen that not all Turing machines deserve to be called "*algorithms*": We argued that Turing machines that semi decide languages, and thus reject by never halting, are not useful computational devices, whereas Turing machines that decide languages and compute functions (and therefore halt at all inputs) are. Our notion of an algorithm must exclude Thring machines that may not halt on some inputs.

We therefore propose to adopt the Turing machine that halts on all inputs as the precise formal notion corresponding to the intuitive notion of an "algorithm". Nothing will be considered an algorithm if it cannot be rendered as a Turing machine that is guaranteed to halt on all inputs, and all such machines will be rightfully called algorithms.

This principle is known as the **Church-Turing thesis**. It is a thesis, not a theorem, because it is not a mathematical result: *It simply asserts that a certain informal concept (algorithm) corresponds to a certain mathematical object (Turing machine).* Not being a mathematical statement, the Church-Thring thesis cannot be proved. It is theoretically possible, however, that the Church-Turing thesis could be disproved at some future date, if someone were to propose an alternative model of computation that was publicly acceptable as a plausible and reasonable model of computation, and yet was provably capable of carrying out computations that cannot be carried out by any Turing machine. No one considers this likely.

According to the Church-Turing thesis, computational tasks that cannot be performed by Turing machines are *impossible*, *hopeless*, *undecidable*.

Briefly

Informally, an algorithm is a rule for solving a problem in a finite number of steps. **Church - Turing thesis** states that an algorithm corresponds to a Turing machine that *halts on all inputs.* **CHURCH:** A function of positive integers is effectively computable only if recursive. **TURING:** Turing machine can do anything that could be described purely mathematical.

2 Universal Turing Machines

We shall argue that Turing machines are also software. That is, we shall show that there is a certain "generic" Turing machine that can be programmed, about the same way that a general-purpose computer can, to solve *any* problem that can be solved by Turing machines. The "program" that makes this generic machine behave like a specific machine M will have to be *a description of* M. In other words, we shall be thinking of the formalism of Turing machines as a *programming language*, in which we can write programs. Programs written in this language can then be *interpreted* by a *universal Turing machine* (that is to say, another program in the same language).

That a program written in a language can interpret any program in the same language is not a very novel idea (it is the basis of the classical method for "bootstrapping" language processors). But to continue with our project in this book we must make this point precise in the context of Turing machines.

To begin, we must present a general way of specifying Turing machines, so that their descriptions can be used as input to other Thring machines. That is, we must define a language whose strings are all legal representations of Turing machines. One problem manifests itself already:

> No matter how large an alphabet we choose for this representation, there will be Turing machines that have more states and more tape symbols.

Evidently, we must encode the states and tape symbols as strings over a fixed alphabet. We adopt the following convention:

- A string representing a Turing machine state is of the form $\{q\}\{0,1\}^*$; that is, the letter q followed by a binary string.
- Similarly, a tape symbol is always represented as a string in $\{a\}\{0,1\}^*$

Let $M = (K, \Sigma, \delta, s, H)$ be a Turing machine, and let i and j be the smallest integers such that $2^i \geq |K|$, and $2^j \geq |\Sigma| + 2$. Then each state in K will be represented as a q followed by a binary string of length i; each symbol in Σ will be likewise represented as the letter a followed by a string of j bits. The head directions $\leftarrow$ and $\rightarrow$ will also be treated as "honorary tape symbols" (they were the reason for the "+2" term in the definition of j). We fix the representations of the special symbols $\sqcup$, $\triangleright$, $\leftarrow$, and $\rightarrow$ to be the lexicographically four smallest symbols, respectively:

- $\sqcup$ will always be represented as $a0^j$,
- $\triangleright$ as $a0^{j-1}1$,
- $\leftarrow$ as $aO^{j-2}10$, and
- $\rightarrow$ as $a0^{j-2}11$.

The start state will always be represented as the lexicographically first state, $q0^i$. Notice that we require the use of leading zeros in the strings that follow the symbols a and q, to bring the total length to the required level.

Represent TMs as strings

- min i such that $2^i \geq |K|$
- States are represented as

  $$q\{\textit{binary string whose length is } i\}$$

  for example: $q00$, $q01$, $q10$, $q11$ (increase the number of bits w.r. to the number of states)
- $q0^i$ is *fixed* representation of the start state
- min j such that $2^j \geq |\Sigma| + 2$ (for left/right), a + j bits
- Symbols are represented as

  $$a\{\textit{binary string whose length is } j\}$$

  for example: $a0$, 1 (increase the number of bits w.r. to the number of states)
- Fixed symbols:
  - $\sqcup$: $a0^{j-2}00 = a0^j$,
  - $\triangleright$: $a0^{j-2}01 = a0^{j-1}1$,
  - $\leftarrow$: $a0^{j-2}10$,
  - $\rightarrow$: $a0^{j-2}11$.

We shall denote the representation of the whole Turing machine M as "M". "M", consists of the transition table δ. That is, it is a sequence of strings of the form $(q,\ a,\ p,\ b)$, with q and p representations of states and a, b of symbols, separated by commas and included in parentheses. We adopt the convention that the quadruples are listed in *increasing lexicographic order*, starting with $\delta(s,\ \sqcup)$. The set of halting states H will be determined indirectly, by the absence of its states as first components in any quadruple of "M". If M decides a language, and thus $H = \{y,\ n\}$, we will adopt the convention that y is the lexicographically smallest of the two halt states.

This way, any Turing machine can be represented. We shall use the same method to represent *strings* in the alphabet of the Turing machine. Any string $w \in \Sigma^*$ will have a unique representation, also denoted "w", namely, the juxtaposition of the representations of its symbols.

Example 2.1

Consider the Turing machine $M = (K, \Sigma,\ \delta,\ s, \{h\})$, where $K = \{s,\ q,\ h\}$, $\Sigma = \{\sqcup,\ \triangleright,\ a\}$, and δ is given in this table.

state	symbol	δ
s	a	$(q,\ \sqcup)$
s	$\sqcup$	$(h,\ \sqcup)$
s	$\triangleright$	$(s,\ \rightarrow)$
q	a	$(s,\ a)$
q	$\sqcup$	$(s,\ \rightarrow)$
q	$\triangleright$	$(q,\ \rightarrow)$

Since there are three states in K and three symbols in Σ, we have $i = 2$ and $j = 3$. These are the smallest integers such that $2^i \geq 3$ and $2^j \geq 3 + 2$. The states and symbols are represented as follows:

state / symbol	representation
s	$q00$
q	$q01$
h	$q11$
$\sqcup$	$a000$
$\triangleright$	$a001$
$\leftarrow$	$a010$
$\rightarrow$	$a011$
a	$a100$

Thus, the representation of the string $\triangleright aa\sqcup a$ is

$$\text{“}\triangleright aa\sqcup a\text{”} = a001a100a100a100a000a100$$

The representation "M" of the Turing machine M is the following string:

$$\begin{aligned}\text{“}M\text{”}=&(q00,\ a100,\ q01,\ a000),\ (q00,\ a000,\ q11,\ a000),\\ &(q00,\ a001,\ q00,\ a011),\ (q01,\ a100,\ q00,\ a011),\\ &(q01,\ a000,\ q00,\ a011),\ (q01,\ a001,\ q01,\ 011).\end{aligned}$$

Now we are ready to discuss a *universal Turing machine* U, which uses the encodings of other machines as programs to direct its operation. Intuitively, U takes two arguments, *a description of a machine* M, "M", and *a description of an input string* w, "w". We want U to have the following property:

> U halts on input "M" "w" if and only if M halts on input w.

To use the functional notation for Turing machines we developed in the last chapter,

$$U(\text{“}M\text{”}\ \text{“}w\text{”}) = \text{“}M(w)\text{”}$$

We actually describe not the single-tape machine U, but a closely related 3-tape machine U' (then U will be the single-tape Turing machine that simulates U').

Specifically, U' uses its three tapes as follows:

- Initially $\triangleright\sqcup$"M" "w" is on the first tape of U'
- U' copies "M" to the second tape, shifts "w" to the left ($\triangleright\sqcup$"w")
- U' writes $q0^i$ to its third tape (initial state)
- Then U' simulates M as follows:
  - U' scans its second tape until it finds a quadruple (*a transition*) whose first element matches the string in the third tape (e.g. *the state*) and second element matches the string on the first tape (e.g. *the symbol under the reading head of* M)
  - If it finds such a quadruple, updates the string on the third tape (*the state of* M), and performs the action on the first tape (*move the head, or write a symbol*)
  - If it cannot find a matching quadruple or the state is in H, then halt.

3 The Halting Problem

The question is that "Can we write a program that tells whether an arbitrary Turing machine (program) P halts on its input X?"

$$\texttt{halts}(P,\ X)\texttt{:}\begin{cases}\texttt{YES} & \text{if } P \text{ halts on } X\\ \texttt{NO} & \text{if } P \text{ does not halts on } X\end{cases}$$

Consider the following example:

```
diagonal(X)
  a: if halts(X, X) then goto a else halt
```

Notice what `diagonal`(X) does: If your halts program decides that program X would halt if presented with itself as input, then `diagonal`(X) loops forever; otherwise it halts.

And now comes the unanswerable question:

> Does `diagonal(diagonal)` halt?

It halts if and only if the call `halts(diagonal, diagonal)` returns "no"; in other words, *it halts if and only if it does not halt.* This is a contradiction:

> We must conclude that the only hypothesis that started us on this path is false, that program `halts(`P`, `X`)` does not exist.

That is to say, there can be *no program, no algorithm* for solving the problem halts would solve: to tell whether arbitrary programs would halt or loop. Thus, **the halting problem is undecidable**.

We are ready to define *a language that is not recursive*, and *prove that it is not.* Let

$H = \{$"M" "w" $\mid$ Turing machine M halts on input string $w\}$

Notice first that H is *recursively enumerable*: It is precisely the *language semidecided by our universal Turing machine* U. Indeed, on input "M" "w", U halts precisely when the input is in H.

Furthermore, if H is recursive, then *every* recursively enumerable language is recursive. In other words, H holds the key to the question we asked in Section 4.2, *whether all recursively enumerable languages are also Turing decidable*:

> The answer is positive if and only if H is recursive.

For suppose that H is indeed decided by some Turing machine M_0. Then given any particular Turing machine M semideciding a language $L(M)$, we could design a Turing machine M' that decides $L(M)$ as follows:

First, M' transforms its input tape from $\triangleright\sqcup w\sqcup$ to $\triangleright$"M" "w"$\sqcup$ and then simulates M_0 on this input. By hypothesis, M_0 will correctly decide whether or not M accepts w.

However we can show, by formalizing the argument for `halts(`P`, `X`)` above, that H is *not recursive.* First, if H were recursive, then

$$H_1 = \{\text{"}M\text{"} \mid \text{Turing machine } M \text{ halts on input string "}M\text{"}\}$$

would also be recursive. (H_1 stands for the `halts(`X`, `X`)` part of the diagonal program.) Uf there were a Turing mahcine M_0 that could decide H, then a Turing machine M_1 to decide H_1 would only need to transform its input string $\triangleright$"M"$\sqcup$ to $\triangleright$"M" "M"$\sqcup$ and then yield control to M_0. Therefore, it suffices to show that H_1 is not recursive.

Second, if H_1, were recursive, then its complement would also be recursive:

$$\overline{H_1} = \{w \mid \text{either } w \text{ is not the encoding of a Turing machine or it is the encoding of "}M\text{" of a Turing machine } M \text{ that does not halt on "}M\text{"}\}$$

This is so because the class of recursive languages is closed under complement. Incidentally, $\overline{H_1}$ is the *diagonal* language, the analog of our `diagonal` program, and the last act of the proof.

But $\overline{H_1}$ cannot even be recursively enumerable (let alone recursive). For suppose M^* were a Turing machine that semidecided $\overline{H_1}$. Is "M^*" in $\overline{H_1}$? By definition of $\overline{H_1}$, "M^*" $\in \overline{H_1}$ if and only if M^* does not accept input string "M^*". But M^* is supposed to semidecide $\overline{H_1}$, so "M^*" $\in \overline{H_1}$ if and only if M^* accepts "M^*". We have concluded that M^* accepts "M^*" if and only if M^* does not accept "M^*". This is absurd, so the assumption that M_0 exists must have been in error.

Theorem 3.1

The language H is not recursive; therefore, the class of recursive languages is a strict subset of the class of recursively enumerable languages.

Theorem 3.2

The class of recursively enumerable languages is not closed under complement.