TDDD14/TDDD85 Lecture 15: Undecidability

Victor Lagerkvist (based on slides by Christer Bäckström and Gustav Nordh)

In this lecture we investigate the limits of Turing machines, and thus the limits of computation itself, and prove that there exist languages which cannot be recognised by any Turing machine.

Background and Intuition

Recall that a language is *Turing-recognizable* if there exists a Turing machine recognising the language (which may or may not loop for strings not included in the language) and that a language is *Turing-decidable*, or simply *decidable* if there exists a Turing machine which recognises the language and which halts on every input. Since Turing machines are so powerful, could it be the case that *every* language is Turing-recognisable, or even decidable? Or could there exist languages which no Turing machine can recognise (or decide)? On the one hand, for the previous classes of automata we have always been able to come with classes of languages which the automata in question could not recognise. For example, by proving that $\{0^n1^n \mid n \geq 0\}$ is not regular, or that $\{0^n1^n2^n \mid n \geq 0\}$ is not context-free. On the other hand, it is not evident that this can be done in the context of Turing machines, and it is certainly not evident if it (for example) is possible to generalise $\{0^n1^n2^n \mid n \geq 0\}$ to a language which is not Turing-recognisable. Although we will successfully show the existence of languages which are not Turing-recognisable, we will see that the resulting languages are rather different from what we are accustomed to.

A Remark Concerning Encodings

Before we turn to the main topic of this lecture we need to make a few remarks about encodings. Most of the problems that we will encounter can be seen as *meta problems*, meaning that they ask questions about various properties of Turing machines. For example, can we construct a Turing machine which given a Turing machine and a string, decides if the string is accepted by the given Turing machine? To accomplish this the Turing machine could e.g. attempt to simulate the given Turing machine on the given input string.

However, a Turing machine, according to the definition from the previous lecture, always takes a string (over some fixed alphabet) as argument. So what do we really mean by "taking a Turing machine

as argument"? The answer, as it turns out, is actually not so complicated.

Example 1. *Assume that you want to write a program simulating a Turing machine, similarly to how one can write a program simulating a finite automaton, or a pushdown automaton. To represent a Turing machine one could e.g. use a class consisting of a finite set of states, a finite alphabet, the finite transition function, and so on. But in the end, any instantiation of this class is nothing else than a bitstring, a sequence of* 0 *and* 1*. A sequence like this is roughly what we mean by an "encoding" in this context.*

If M is a Turing machine then we will write $\langle M \rangle$ to denote a suitable string encoding. We will not explicitly mention the alphabet or any details of the actual encoding, since the details are not important. Hence, just think of $\langle M \rangle$ as a (probably) long string which can be used to reconstruct M. If M is a Turing machine and w an input string to this Turing machine then we will occasionally also write $\langle M, w \rangle$ to denote a suitable encoding of this pair. Again, the precise details are not important.

The Theoretical Limits of Computation: Undecidability

We first remark that there is no analogue to the pumping lemma in the context of Turing machines. To see this, note that while a Turing machine has a fixed number of states, the number of configurations cannot be bounded, and we thus cannot guarantee that accepted strings follow some sort of "pattern". And although we have not yet discussed the "grammatical counterpart" to Turing machines, it is not possible to use this description to come up with a pumping lemma either. Hence, we have to investigate alternative proofs, and begin with the following theorem which uses a counting argument to conclude that not every language is Turing recognisable.

Theorem 1. *There exist languages that are not Turing-recognizable*

Proof. (Sketch)

The basic idea is to show that while there exists an infinite number of Turing machines, it is possible to design an algorithm which enumerates Turing machines $M_1, M_2, M_3, \ldots$ such that *every* Turing machine appears in this list. This turns out to be surprisingly simple. Assume that we fix the size of the alphabet, the tape alphabet and the number of states to some constant. Then there only exists a finite number of Turing machines which we can enumerate by enumerating *all* possible transition functions. Hence, if we enumerate Turing machines in this manner, and gradually increase the size of these constants, then every Turing machine will eventually appear in this

list. In mathematical terminology the set of Turing machines is *countable*. Other examples of countable sets are the set of natural numbers and the set of prime numbers, which can also be enumerated in a straightforward manner.

In contrast, the set of all languages is *not* countable, i.e., is *uncountable*. Assume that we have an alphabet Σ of size at least two. Does it seem possible to enumerate *all* languages over Σ, i.e., subsets of Σ^*, similar to how we could enumerate Turing machines? The idea is then to perform a proof by contradiction: assume that we are given an infinite sequence $\Sigma_1, \Sigma_2, \Sigma_3, \ldots$ of all languages over Σ, i.e., each language is equal to Σ_i for some $i \geq 1$. One can then use a technique known as *diagonalization* to produce a language which is distinct from each element Σ_i in this sequence. This is similar to how the set of real numbers can be proven to be uncountable. See Corollary 4.18 in Sipser [1] for details.

But if we have an uncountable number of languages but only a countable number of Turing machines, then it cannot be the case that *every* language is recognised by some Turing machine. Hence, there exists languages that are not Turing-recognisable. □

[1] M. Sipser. *Introduction to the Theory of Computation*. Course Technology, Boston, MA, third edition, 2013

This theorem should mainly be viewed as a theoretical result, but it highlights something of importance: given a language, we cannot necessarily define a Turing machine which recognises the language. But can we do better? Can we pinpoint some concrete languages which are not Turing-recognisable? This can indeed be done, but we will first turn to the problem of finding languages that are not decidable, i.e., are *undecidable*.

Example 2. *Assume that you have just finished writing a nice meta-interpreter for your favourite programming language. This interpreter takes a program, represented as a string, and an input string, as arguments, and returns the resulting of interpreting the given program with the given input string. Naturally, after this feat of engineering you immediately start looking for applications for your shiny new toy. Would it be possible to answer some meta questions about programs? For example, could we determine whether a program terminates or not? This would be a rather nice application since it would make debugging much easier. Hence, you decide to extend the meta-interpreter so that it returns* 1 *if a given program halts with respect to the given input string, and* 0 *if it loops. The easy part is of course if the program terminates: you then simply run the interpreter with the given input, and once it terminates you return* 1*. To handle the case when the program does not seem to terminate you implement some "clever" loop-detection scheme and are completely satisfied with the test programs that you tried. To make it user-friendly you call the function* halt *and you let it take a single argument consisting of the program in question (which*

contains some type of main function containing some test data).

However, how can you be sure that halt *actually works? Could it be possible to come up with a counter example? Consider the following. We define a new function (in the same programming language)* halt′ *which takes a program P as argument and*

1. *calls* halt(P),
2. *if the result is* 0 *then it returns* 1*, and if the result is* 1 *then it loops.*

Hence, the new program does exactly the opposite of halt*. But now, what happens if we call* halt′ *with* itself *as input? That is, a string encoding of the the source code of* halt′*. We have the following possibilities.*

1. *Case* 1*:* halt *returned* 0*, meaning that* halt′ *loops. But according to the definition of* halt′ *it should return* 1*, not loop.*
2. *Case* 2*:* halt *returned* 1*, i.e., that* halt′ *does not loop. But then* halt′ *will loop, which is exactly the opposite of what* halt *reported.*

Since neither outcome is possible we conclude that the claimed properties of halt *cannot be possible, and that it cannot correctly deduce whether every given program halts or not.*

Deciding whether a program (or a Turing machine) halts on a given input is typically called the *Halting problem*. The formal proof of the undecidability of the halting problem closely follows the intuition outlined above. However, before we turn to the halting problem we begin by proving undecidability of a related problem with respect to Turing machines: given a Turing machine M and a string w, does M accept w? Thus, we are interested in whether the following language is decidable.

Definition 1. *Let* $A_{\text{TM}} = \{\langle M, w\rangle \mid M \text{ accepts } w\}$.

Hence, each element in A_{TM} is a Turing machine (or rather, an encoding of it) and a string which the Turing machine accepts. For example, if we let M be a Turing machine from the previous lecture recognising the language $\{0^n1^n2^n \mid n \geq 0\}$, then A_{TM} would contain $\langle M, \varepsilon\rangle$, $\langle M, 012\rangle$, $\langle M, 001122\rangle$, and so on. Hence, to decide A_{TM} we for *every* Turing machine M would need to determine *all* strings that M accepts. Indeed, we can prove that to Turing machine can accomplish this and that the language is undecidable.

Theorem 2. A_{TM} *is undecidable.*

Proof. The basic idea is similar to the construction in Example 2. Assume, with the aim of reaching a contradiction, that A_{TM} is decided

by some Turing machine H. Hence, for any Turing machine M and string w, the machine H can always decide if M accepts w.

Now, define a Turing machine D which takes a Turing machine $\langle M \rangle$ as input and:

1. Runs H with $\langle M, \langle M \rangle \rangle$ as input, and
2. accepts if it is rejected and rejects if it is accepted.

$\langle M, \langle M \rangle \rangle$ looks weirder than it actually is. All that we are asking is whether the Turing machine M, when given *itself* as input (in a suitable string encoding) accepts or rejects. Think about a compiler for a programming language written in the same language: why should it not be able to compile itself?

Hence, the machine D asks H whether the machine M accepts $\langle M \rangle$, and returns the opposite answer. Now comes the fun part. What happens if we run D with $\langle D \rangle$ as input? Then D asks H if D accepts $\langle D \rangle$. But if H answers "yes", then D rejects, meaning that the answer was false. Similarly, if H answers "no", then D accepts, also meaning that the answer was false. Both these cases lead to a contradiction, and we conclude that our original assumption was false. Hence, A_{TM} is undecidable. □

We finish this section by showing the existence of a language which is not Turing-recognisable. Let $\overline{A_{\text{TM}}} = \{\langle M, w \rangle \mid \langle M, w \rangle \notin A_{\text{TM}}\}$ be the complement of A_{TM}, i.e., it contains a pair $\langle M, w \rangle$ if M does not accept w.

Theorem 3. *$\overline{A_{\text{TM}}}$ is not Turing-recognisable.*

Proof. Assume there exists a Turing machine M' which recognises $\overline{A_{\text{TM}}}$. We will show that M' can be used to decide A_{TM}, which contradicts Theorem 2. Hence, let M be a Turing machine and let w be an input string. We construct a Turing machine which:

1. simulates M with w as input, and
2. simulates M' with $\langle M, w \rangle$ as input,

but does so simultaneously, in the sense that it first runs M a fixed number of steps, and then runs M' with a fixed number of steps, and then goes back and forth between the simulations. But then this machine decides A_{TM} since if w is accepted then this will eventually be discovered by the simulation of M, and if w is not accepted then this will eventually be discovered by the simulation of M'. □

Mapping Reductions

We have discovered the existence of undecidable languages, but do not have a general recipe for proving that a language is undecidable (or not Turing-recognisable). While we cannot obtain something as simple as the pumping lemma for regular or context-free languages,

we in this section introduce a fairly powerful method for proving undecidability based on the notion of a *reduction*. The idea is as follows. Given a language B which we suspect might be undecidable, take a language A that we have already established to be undecidable (e.g., A_{TM} from the preceding section), and attempt to translate A into B so that the decidability of B would imply the decidability of A. Formally, we then want to find a function $f\colon A \to B$ taking a string $w \in A$ as argument and returning a string $f(w) \in B$. However, since we are thinking in terms of computability and decidability, the function f has to be sufficiently simple so that it can be computed by a Turing machine.

Definition 2. *A function $f\colon \Sigma^* \to \Sigma^*$ is said to be* computable *if some Turing machine M on every input w halts with $f(w)$ on its tape.*

The formal definition of a reduction, often called a *mapping reduction*, is then fairly straightforward.

Definition 3. *The language A is* mapping reducible *to the language B if there is a computable function $f\colon \Sigma^* \to \Sigma^*$ such that for every w*

$$w \in A \Leftrightarrow f(w) \in B.$$

Hence, a mapping reduction f is a "translation" between two languages such that a string w belongs to the first language if and only if the result of the translation, $f(w)$, belongs to the second language.

1. *The function f is called a* reduction *from A to B.*
2. *If A is mapping reducible to B then we write $A \leq_m B$.*

It is then not so difficult to prove that mapping reductions can be used to relate languages when it comes to decidability.

Theorem 4. *Let A and B be two languages. If B is decidable and $A \leq_m B$, then A is decidable.*

Proof. Let M_B be a Turing machine that decides B and let f the reduction from A to B. Given input w we:

1. compute $f(w)$,
2. run M_B on $f(w)$, accept if M_B accepts $f(w)$, and reject if M_B rejects $f(w)$.

□

But since our main focus at the moment is to show undecidability we are more interested in the contrapositive form ($\neg y \to \neg x$ instead of $x \to y$) of Theorem 4.

Corollary 1. *Let A and B be two languages. If A is undecidable and $A \leq_m B$, then B is undecidable.*

Using the exact same arguments one can also prove (1) that if B is Turing-recognisable and $A \leq_m B$, then A is Turing-recognisable, and (2) that if A is not Turing-recognisable and $A \leq_m B$ then B is not Turing-recognisable. However, let us concentrate on undecidability, and consider an example of how we can use a mapping reduction to prove that a language is undecidable.

Definition 4. *Let* $H_{\text{TM}} = \{\langle M, w\rangle \mid M \text{ halts with input } w\}$.

This language, or rather, the corresponding computational problem of determining whether a Turing machine halts or not, is typically called the *Halting problem*. We could prove that it is undecidable using similar arguments to those in Theorem 2, but it is easier to prove undecidibility by using the already established fact that A_{TM} is undecidable.

Theorem 5. *H_{TM} is undecidable.*

Proof. We begin by showing the idea behind the reduction and then show how it can be phrased as a mapping reduction. Assume that we have a machine M_H which decides H_{TM}. We want to use this machine to decide A_{TM}. Hence, let M be an arbitrary Turing machine together with an input string w. Merely running M_H with $\langle M, w\rangle$ as input is not good enough since if it halts we do not know whether M accepts or rejects w. Instead, we construct a new Turing machine M' which simulates M over a given string, accepts if it accepts, and *loops* if it rejects. Hence, the machine M' almost agrees M, but instead of rejecting a string it chooses to loop. If we then use H_{TM} with the input $\langle M', w\rangle$, then (1) if H_{TM} accepts then M' halted, meaning that M accepted w, and (2) if H_{TM} rejects then M' looped, meaning that M rejected the string w. But since A_{TM} is undecidable this leads to a contradiction, and we conclude that H_{TM} cannot be decidable.

Let us now see how the above can be expressed as a mapping reduction. We now need to show that there exists a (computable) function $f\colon A_{\text{TM}} \to H_{\text{TM}}$ such that $\langle M, w\rangle \in A_{\text{TM}}$ if and only if $f(\langle M, w\rangle) \in H_{\text{TM}}$. We describe f as follows.

1. Let $\langle M, w\rangle$ be an input string.
2. Construct a Turing machine M' which for a given input string simulates M and accepts if M accepts, and loops if M rejects.
3. Output $\langle M', w\rangle$.

For correctness, assume first that $\langle M, w\rangle \in A_{\text{TM}}$. Hence, M accepts w. But due to the definition of the new machine M' it follows that M' accepts w and halts. Hence, $f(\langle M, w\rangle) = \langle M', w\rangle \in H_{\text{TM}}$.

For the other direction, assume that $f(\langle M, w\rangle) = \langle M', w\rangle \in H_{\text{TM}}$. Then M' halts under the input string w. But this means that M accepts w, and that $\langle M, w\rangle \in A_{\text{TM}}$. □

For more examples of mapping reductions, see Ch. 5.3. in Sipser [2].

[2] M. Sipser. *Introduction to the Theory of Computation*. Course Technology, Boston, MA, third edition, 2013

Summary

We proved the existence of undecidable and non-Turing-recognisable languages. Hence, not even Turing machines can compute everything. This should not be viewed as a shortcoming of Turing machines, but rather that not all properties are inherently computable.

Food for Thought

1. The *nth busy beaver number* is the largest number of ones which can be written by a halting Turing machine over $\Sigma = \{0, 1\}$ with exactly n states (with a blank input tape). Determine the first and second busy beaver numbers.
2. Can you come up with any non-trivial meta questions in compiler design which risk being undecidable? For example, given two functions, can we decide whether they always give the same output?

References

M. Sipser. *Introduction to the Theory of Computation*. Course Technology, Boston, MA, third edition, 2013.