St. Pölten University of Applied Sciences

/informatik & security

/fh/// st. pölten

Decidability and Computability

Theoretical Computer Science

Dipl.-Ing. Hubert Schölnast, BSc
September 28, 2021

St. Pölten University of Applied Sciences, Campus-Platz 1, 3100 St. Pölten, T: +43 (2742) 313 228, F: +43 (2742) 313 228-339, E: csc@fhstp.ac.at, I: www.fhstp.ac.at

Table of Contents

1 Definitions

1.1 The word problem

We have a grammar $G = \{V, \Sigma, P, S\}$ and a language $\mathcal{L}(G)$ that is defined by this grammar.

Σ^* is the Kleene closure of Σ, what means, that Σ^* is the set of all possible words that can be written with signs from the alphabet Σ. And it is clear, that $\mathcal{L}(G)$ is a subset of Σ^*:

$$\mathcal{L}(G) \subseteq \Sigma^*$$

And we have given a word w, i.e. an element of Σ^*:

$$w \in \Sigma^*$$

And now we want to know if this particular word w belongs to the language defined by the grammar, or not. So, we ask

$$\text{Is } w \in \mathcal{L}(G)?$$

And this question is the word problem.

1.2 Characteristic function

The word problem can be defined as a function, and because there is an individual function for each language, and because a language's function is characteristic for this language, it is called "characteristic function":

$$cf: \Sigma^* \to \{0, 1\}$$

or

$$cf: \Sigma^* \to \{\text{false}, \text{true}\}$$

- The value of $cf(w)$ is 1 (or true) if $w \in \mathcal{L}(G)$
- The value of $cf(w)$ is 0 (or false) if $w \notin \mathcal{L}(G)$

1.3 Problem

To say whether a word belongs to a language or not, which means the same as to say whether the characteristic function of a language yields 1 or 0, is a problem. And any specific word one wants to test is an instance of a problem. Thus, the term "problem" is closely related to the term "language". Each language defines its own problem. And for every problem, there is a language that corresponds to that problem.

So in computer science we can use the terms "problem" and "language" interchangeably.

And this means: Every language is a problem and every problem is a language and every problem is a subset of the Kleene closure of an alphabet Σ.

$$P = \mathcal{L}(G)$$
$$P \subseteq \Sigma^*$$

1.4 Computability

- Intuitive definition:
  A function is called intuitively computable, if there is an algorithm, that computes this function.
- Formal definition:
  A function is called Turing-computable, if there is a Turing machine, that computes the function when it is provided the parameters of the function as input, writes the result of the function on its tape, and halts.

1.4.1 Church Turing Thesis

Every intuitively computable function is Turing computable and vice versa.

Other definitions are based on predicate logic, λ-calculus, simple programming languages and others, but they are all equivalent to the definition using Turing machines.

1.5 Decidability

A problem is called decidable, if the characteristic function of the set that constitutes the problem is Turing-computable, i.e. if the Turing machine decides for every word, whether it belongs to the language or not and halts (And it is not decidable if this is not the case.)

1.5.1 decidable

A language is decidable, if its characteristic function is computable.

Decidable languages are also called μ-recursive computable languages or just "recursive languages".

A language is decidable, if there is a Turing machine, that will halt on every word form Σ^*. It then will either accept toe word or reject it.

This means the same as:

A problem is decidable, if there is an algorithm, that will halt on every instance of the problem. It then will either answer "yes" or "no".

1.5.2 recursively enumerable (semidecidable)

There are languages, for which the Turing machine halts only, if the word (the input string) belongs to the language. If the input string does not belong to the language, there is no guarantee, that the Turing machine halts.

These languages are called recursively enumerable (partially decidable or semidecidable).

A language is recursively enumerable, if there is a Turing machine, that will halt on every word that belongs to the language. In this case it will accept the word. For words that do not belong to the language, it either halts and rejects the word or it runs forever.

A problem is semidecidable, if there is an algorithm, that will halt on every instance of the problem if it's answer is "yes". For instances, where the answer should ne "no" it either answers with "no" or it runs forever.

1.5.3 undecidable

There are languages, for which no Turing machine exists that can decide whether a given input string belongs to the language or not. You will soon see why this must be the case.

If there are undecidable languages, there must be undecidable functions, too. (For example the characteristic functions of these languages.)

A language is undecidable, if any Turing machine, that is capable of processing the language, will run forever on some words of Σ^* that belong to the language and also on some words that do not belong to the language. (It also may halt in some cases and the accept or reject the word correctly.)

A problem is undecidable, if any algorithm, that is capable of processing the problem, will run forever on some instances of the problem, regardless of whether the answer should be "yes" or "no". (It also may halt in some cases and then give the correct answer.)

2 How many problems do exist?

Σ^* is an infinite set. It's cardinality is equal to the cardinality of the set of natural numbers:

$$|\Sigma^*| = |\mathbb{N}| = \aleph_0$$

Every problem P is a subset of Σ^*, and the set of all subsets of a set is this set's powerset. So, we are looking for the size of

$$\mathcal{P}(\Sigma^*)$$

If you remember the chapter about powersets, you will remember this formula that is valid for any set S:

$$|\mathcal{P}(S)| = 2^{|S|}$$

So, we have

$$|\mathcal{P}(\Sigma^*)| = 2^{|\Sigma^*|}$$
$$|\mathcal{P}(\Sigma^*)| = 2^{|\mathbb{N}|}$$
$$|\mathcal{P}(\Sigma^*)| = 2^{\aleph_0}$$

Does this help us? Not really, because we already know that $\aleph_0$ is an infinitely large number, so also $2^{\aleph_0}$ must be infinitely large. But does this mean that $2^{\aleph_0}$ is bigger than $\aleph_0$? Or are they of the same size? Infinite is infinite, or isn't it?

3 Cantor's diagonal argument

Let's keep is as simple as we can. We again use our standard minimal alphabet

$$\Sigma = \{0, 1\}$$

And then we list some elements of Σ^*:

$$\Sigma^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, 111, \cdots\}$$

A problem is a language and a language is a subset of this set. So one problem might contain these instances to which the answer should be "yes" (these are the words that belong to the language):

$$P_1 = \{\varepsilon, \quad 1, 00, \quad 10, 11, 000, \quad 100, 101, \quad 111, \cdots\}$$

Another problem might contain these words:

$$P_2 = \{ \quad 11, 000, 001, 010, \quad 101, 110, \quad \cdots\}$$

Here is a third problem:

$$P_3 = \{ \quad 11, \quad 001, 010, \quad 101, 110, 111, \cdots\}$$

etc.

Let's write this in a table together with some other problems:

Σ^*	ε	0	1	00	01	10	11	000	001	010	011	100	101	110	111	⋯
P_1	ε		1	00		10	11	000				100	101		111	⋯
P_2							11	000	001	010			101	110		⋯
P_3							11		001	010			101	110	111	⋯
P_4	ε	0		00						010	011		101	110		⋯
P_5	ε		1	00	01	10	11	000	001		011	100			111	⋯
P_6		0	1			10		000		010						⋯
P_7		0				10	11		001	010	011		101	110	111	⋯
P_8	ε		1	00				000	001			100			111	⋯
P_9		0	1		01	10	11	000			011	100				⋯
P_{10}	ε	0			01		11		001	010		100		110	111	⋯
P_{11}		0		00	01			000		010			101	110		⋯
P_{12}		0	1	00		10			001		011	100			111	⋯
P_{13}	ε		1			10	11					100	101			⋯
P_{14}	ε				01					010	011		101	110		⋯
P_{15}			1		01	10	11				011	100		110	111	⋯
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱

But we do it a little bit simpler: We make another list and we write a 0 in every cell of the new list where the old list was empty, and we write a 1 in every cell where the old list contained an element of Σ^*. And we don't need the heading (the line with the blue background). When we do so, we get this list:

P_1	1	0	1	1	0	1	1	1	0	0	0	1	1	0	1	⋯
P_2	0	0	0	0	0	0	1	1	1	1	0	0	1	1	0	⋯
P_3	0	0	0	0	0	0	1	0	1	1	0	0	1	1	1	⋯
P_4	1	1	0	1	0	0	0	0	0	1	1	0	1	1	0	⋯
P_5	1	0	1	1	1	1	1	1	1	0	1	1	0	0	1	⋯
P_6	0	1	1	0	0	1	0	1	0	1	0	0	0	0	0	⋯
P_7	0	1	0	0	0	1	1	0	1	1	1	0	1	1	1	⋯
P_8	1	0	1	1	0	0	0	1	1	0	0	1	0	0	1	⋯
P_9	0	1	1	0	1	1	1	1	0	0	1	1	0	0	0	⋯
P_{10}	1	1	0	0	1	0	1	0	1	1	0	1	0	1	1	⋯
P_{11}	0	1	0	1	1	0	0	1	0	1	0	0	1	1	0	⋯
P_{12}	0	1	1	1	0	1	0	0	1	0	1	1	0	0	1	⋯
P_{13}	1	0	1	0	0	1	1	0	0	0	0	1	1	0	0	⋯
P_{14}	1	0	0	0	1	0	0	0	0	1	1	0	1	1	0	⋯
P_{15}	0	0	1	0	1	1	1	0	0	0	1	1	0	1	1	⋯
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱

Now, we have a closer look at the elements in the diagonal:

P_1	1	0	1	1	0	1	1	1	0	0	0	1	1	0	1	⋯
P_2	0	0	0	0	0	0	1	1	1	1	0	0	1	1	0	⋯
P_3	0	0	0	0	0	0	1	0	1	1	0	0	1	1	1	⋯
P_4	1	1	0	1	0	0	0	0	0	1	1	0	1	1	0	⋯
P_5	1	0	1	1	1	1	1	1	1	0	1	1	0	0	1	⋯
P_6	0	1	1	0	0	1	0	1	0	1	0	0	0	0	0	⋯
P_7	0	1	0	0	0	1	1	0	1	1	1	0	1	1	1	⋯
P_8	1	0	1	1	0	0	0	1	1	0	0	1	0	0	1	⋯
P_9	0	1	1	0	1	1	1	1	0	0	1	1	0	0	0	⋯
P_{10}	1	1	0	0	1	0	1	0	1	1	0	1	0	1	1	⋯
P_{11}	0	1	0	1	1	0	0	1	0	1	0	0	1	1	0	⋯
P_{12}	0	1	1	1	0	1	0	0	1	0	1	1	0	0	1	⋯
P_{13}	1	0	1	0	0	1	1	0	0	0	0	1	1	0	0	⋯
P_{14}	1	0	0	0	1	0	0	0	0	1	1	0	1	1	0	⋯
P_{15}	0	0	1	0	1	1	1	0	0	0	1	1	0	1	1	⋯
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱

In our example, this is this sequence: $100111110101111\cdots$. And in this sequence, we flip all signs. So each 0 become a 1 and each 1 becomes a 0. Then we have: $011000001010000\cdots$

3.1 Here is the trick

We want to prove, that

$$|\mathcal{P}(\Sigma^*)| > |\mathbb{N}|$$

Let's assume for a moment, that this is not true. So, for a moment, we believe $|\mathcal{P}(\Sigma^*)| \leq |\mathbb{N}|$ and then we will see, where this believe will bring us.

If $|\mathcal{P}(\Sigma^*)| \leq |\mathbb{N}|$ then we can assign to every subset of Σ^* (i.e. to every Problem) a specific natural number. Then we can sort the problems by this number, and we get a list. This is exactly what we have done in the pages above.

But this also means, that this list must contain all problems, because when every problem has a number, it has it's dedicated line in the list.

And we have found a binary code that describes the problem. The code for the problem at position 1 of the list was $101101110001101\cdots$, the problem at position 2 had the code $000000111100110\cdots$ etc.

And when every problem is in the list, then also the problem with the code $011000001010000\cdots$ must be there. This is the code we from the inverted diagonal.

Let's see, where in this list we can find the problem with this code.

inv.diagonal	0	1	1	0	0	0	0	0	1	0	1	0	0	0	0	⋯
P_1	1	0	1	1	0	1	1	1	0	0	0	1	1	0	1	⋯
P_2	0	0	0	0	0	0	1	1	1	1	0	0	1	1	0	⋯
P_3	0	0	0	0	0	0	1	0	1	1	0	0	1	1	1	⋯
P_4	1	1	0	1	0	0	0	0	0	1	1	0	1	1	0	⋯
P_5	1	0	1	1	1	1	1	1	1	0	1	1	0	0	1	⋯
P_6	0	1	1	0	0	1	0	1	0	1	0	0	0	0	0	⋯
P_7	0	1	0	0	0	1	1	0	1	1	1	0	1	1	1	⋯
P_8	1	0	1	1	0	0	0	1	1	0	0	1	0	0	1	⋯
P_9	0	1	1	0	1	1	1	1	0	0	1	1	0	0	0	⋯
P_{10}	1	1	0	0	1	0	1	0	1	1	0	1	0	1	1	⋯
P_{11}	0	1	0	1	1	0	0	1	0	1	0	0	1	1	0	⋯
P_{12}	0	1	1	1	0	1	0	0	1	0	1	1	0	0	1	⋯
P_{13}	1	0	1	0	0	1	1	0	0	0	0	1	1	0	0	⋯
P_{14}	1	0	0	0	1	0	0	0	0	1	1	0	1	1	0	⋯
P_{15}	0	0	1	0	1	1	1	0	0	0	1	1	0	1	1	⋯
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱

- It cannot be the problem P_1 because the 1st item of P_1 is 1 while the 1st item of the inverted diagonal is 0.
- It cannot be the problem P_2 because the 2nd item of P_2 is 0 while the 2nd item of the inverted diagonal is 1.

And so on. You can repeat this argument for every single line in the list. The code, we generated from the diagonal, cannot be the code of the problem P_n because the n^{th} item of P_n is different from the n^{th} item of our constructed code.

This means: The list is incomplete. But it must be complete if $|\mathcal{P}(\Sigma^*)| \leq |\mathbb{N}|$. And this only can mean:

$$|\mathcal{P}(\Sigma^*)| > |\mathbb{N}|$$

quod erat demonstrandum.
And this means that the number of all problems is uncountably infinite.

$$|P| > \aleph_0$$

4 Insolvable problems

Do you remember when we computed the number of busy beavers? We can do the same for the set of all Turing machines. It is a little more complicated, because "normal" Turing machines have more variable elements than busy beavers, but it still is possible define sets of Turing machines with a certain size, and within such a group you can define a certain order (by defining an order for the signs in both alphabets), and then you can assign an individual natural number to each Turing machine, and each natural number is a number of a unique and distinctive Turing machine.

So, the set of all Turing machines is infinitely large, but it is as big as $\mathbb{N}$:

$$|TM| = |\mathbb{N}| = \aleph_0$$

And this again means, that there are more Problems than Turing machines:

$$|P| > |TM|$$

But every Turing machine can accept only exactly 1 language, which means, each Turing machine can solve only exactly 1 problem.

And this means, that there are problems, for which no Turing machine exists, that will be able to solve it.

5 The halting problem

Remember the program will_it_halt.py? Or the 40 busy beavers with 5 states that are running for more than 30 years? Wouldn't it be nice to have a program, that could read the source code of such a program or the description of a Turing machine and that the would tell you if this program or automaton would run forever or if it would halt?

For many programs you also would have to know a concrete input, because it often depends on the input if a program will halt or run forever.

So, we need a halt-tester that can read two inputs: The description of the program or Turing machine, and the input to be fed to this program. The halt-tester would analyze the code and would check what it does with the input, end then it would come to an end (it halts) and prints out

- the program you gave we will loop forever when it reads the given input
  or
- the program you gave we will halt when it reads the given input

5.1 It's impossible

But such a halt-tester is impossible.

There is a nice video that explains why: https://www.youtube.com/watch?v=92WHN-pAFCs It is 7 minutes and 51 seconds long. This here is even shorter (4:13) and also explains it very well: https://www.youtube.com/watch?v=VyHbd6sx5Po

But if you prefer reading, here is the proof:

Let's assume there is a halt-tester, and it can predict for every source code of another program whether it will halt. It reads source code and after a while it says either "it halts" or "it runs forever", and it is always correct.

Then we need an inverter. It accepts only two possible inputs: when the input is "it halts", then the inverter runs forever and will never halt. If the input for the inverter is "it runs forever", then it immediately halts.

We build a new bigger machine by joining the halt-tester and the inverter to one machine that we call "the absurd machine".

If you feed the description of any machine into the absurd machine, it hands it over to the halt-tester, and the halt-tester gives its own output to the invert, both of them are part of the absurd machine.

Now, let's see what happens, if the absurd machine reads it's own description as input. If the halt-tester says "the absurd machine halts" then the inverter will run forever, and, because it is a part of the absurd machine, also the absurd machine will run forever. So, the halt-tester was wrong. But if the halt-tester says "the absurd machine runs forever", then the inverter and therefore also the absurd machine will halt immediately. So, the halt-tester is wrong again. But it is always correct by definition. and this means, that a program like the halt-tester never can exist.

The halting problem is an undecidable problem.

5.2 Other undecidable problems

5.2.1 Post correspondence problem

Named after Emil Leon Post (1897-1954), a Polish-born American mathematician and logician.

It best can be described with an example: There are two lists of words:

element number	list 1	list 2
1	a	aba
2	ab	bb
3	baa	aa

Is there a sequence of element numbers, such that the words, arranged in this order and then concatenated, will produce the same result, no matter if you apply this sequence to list 1 or to list 2?

A solution for the given two lists is the sequence $(1, 3, 2, 3)$ when you write words from list 1 in this order you get:

$$a + baa + ab + baa = abaaabbaa$$

If you do the same with list 2 you get

$$aba + aa + bb + aa = abaaabbaa$$

So, for both lists there is common sequence that will produce the same result.
Finding such a sequence for any arbitrary pair of lists is an undecidable problem

5.2.2 Equivalence problem

The Equivalence Problem

This is the question, whether two Turing machines do accept the same language or not.

This is undecidable for context-sensitive languages and even for non-deterministic context free languages.