

# COMPLEXITY THEORY

## Lecture 4: Undecidability and Recursion

Markus Krötzsch

Knowledge-Based Systems

TU Dresden, 22 Oct 2024

More recent versions of this slide deck might be available.  
For the most current version of this course, see  
[https://iccl.inf.tu-dresden.de/web/Complexity\\_Theory/en](https://iccl.inf.tu-dresden.de/web/Complexity_Theory/en)

# Undecidability so far

We have seen several undecidable problems for TMs:

- The **Halting Problem**: recognise TM-word pairs where the TM halts
- The **Non-Halting Problem**: recognise TM-word pairs where the TM does not halt
- The  **$\varepsilon$ -Halting Problem**: recognise TMs that halt on the empty input

Many further TM-related problems are undecidable ...

... but we can use a shortcut to proving many of them:

**Theorem 4.1 (Rice's Theorem, informal):** Any interesting property related to the language recognised by a given TM is undecidable.

# Rice's Theorem

We can make this formal as follows:

**Definition 4.2:** Let  $\mathcal{P}$  be a set of languages. A language  $\mathbf{L}$  has the property  $\mathcal{P}$  if  $\mathbf{L} \in \mathcal{P}$ . Property  $\mathcal{P}$  is a **non-trivial** property of recognisable languages if there are TM-recognisable languages that have it and others that do not have it.

**Theorem 4.1 (Rice's Theorem):** If  $\mathcal{P}$  is a non-trivial property of recognisable languages, then the following problem is undecidable:

$$\mathcal{P}\text{-ness} = \{\langle \mathcal{M} \rangle \mid \mathbf{L}(\mathcal{M}) \in \mathcal{P}\}$$

# Proof of Rice's Theorem

**Theorem 4.1 (Rice's Theorem):** If  $\mathcal{P}$  is a non-trivial property of recognisable languages, then the following problem is undecidable:

$$\mathcal{P}\text{-ness} = \{\langle \mathcal{M} \rangle \mid \mathbf{L}(\mathcal{M}) \in \mathcal{P}\}$$

**Proof:** We reduce  $\varepsilon$ -Halting to  $\mathcal{P}$ -ness.

- Assume w.l.o.g. that  $\emptyset \notin \mathcal{P}$  (otherwise do the proof for  $\overline{\mathcal{P}}$ )
- Let  $\mathcal{M}_{\mathbf{L}}$  be some TM that recognises a language  $\mathbf{L} \in \mathcal{P}$
- Given any TM  $\mathcal{M}$ , compute a TM  $\mathcal{M}^*$  that behaves as follows:  
On input  $w \in \Sigma^*$ :
  - (1) Simulate  $\mathcal{M}$  on input  $\varepsilon$
  - (2) If  $\mathcal{M}$  halts, simulate  $\mathcal{M}_{\mathbf{L}}$  on  $w$
- Then  $\mathbf{L}(\mathcal{M}^*) = \mathbf{L} \in \mathcal{P}$  if  $\mathcal{M}$  halts on  $\varepsilon$ , and  
 $\mathbf{L}(\mathcal{M}^*) = \emptyset \notin \mathcal{P}$  if  $\mathcal{M}$  does not halt on  $\varepsilon$

For the required Turing reduction, we construct a TM that:

(Step 1) checks if the input is a TM encoding  $\langle \mathcal{M} \rangle$  and rejects otherwise,

(Step 2) returns the result of the check  $\langle \mathcal{M}^* \rangle \in \mathcal{P}\text{-ness}$ . This would decide  $\varepsilon$ -Halting.  $\square$

# Using Rice's Theorem

Here are some simple results that Rice gives us:

**Corollary 4.3:** Given an arbitrary TM  $\mathcal{M}$ , it is undecidable whether the language recognised by  $\mathcal{M}$  has any of the following properties:

- emptiness
- finiteness
- decidability
- regularity
- context-freedom
- contains any given word  $w$  (word problem for TMs)

**Attention:** There are of course many non-trivial properties of TMs that can be decided, and which do not relate to their language:

**Example 4.4:** It is decidable if a TM has at least three states.

# Semi-decidability and Co-semi-decidability

We can distinguish the following two cases:

- (1)  $L$  is Turing-recognisable:  $L$  is semi-decidable
- (2)  $\bar{L}$  is Turing-recognisable:  $L$  is co-semi-decidable

We have seen examples for both:

**Theorem 4.5:** The Halting Problem is semi-decidable.

**Proof:** Use the universal TM to simulate an input TM, and accept if it halts.  $\square$

**Corollary 4.6:** The Non-Halting Problem is co-semi-decidable.

# Semi-decidable + Co-semi-decidable = Decidable

An easy but important observation:

**Theorem 4.7:** If  $L$  is semi-decidable and co-semi-decidable, then  $L$  is decidable.

**Proof:** On input  $w$ , simulate, in parallel, a recogniser for  $L$  and a recogniser for  $\bar{L}$ . At least one of them eventually must halt, so we can decide if  $w \in L$ .  $\square$

We thus obtain an example of a problem that is not Turing-recognisable.

**Corollary 4.8:** The Non-Halting Problem is not Turing-recognisable.

# Turing reductions and semi-decidability

## Observation:

- If  $\mathbf{Q}$  is decidable and  $\mathbf{P} \leq_T \mathbf{Q}$ , then  $\mathbf{P}$  is decidable (Theorem 3.17)
- **But:** if  $\mathbf{Q}$  is semi-decidable and  $\mathbf{P} \leq_T \mathbf{Q}$ , then  $\mathbf{P}$  may or may not be semi-decidable

**Reason:** An oracle for Halting is as good as an oracle for Non-Halting, since we are free to complement the answer in an oracle machine.

This is a general insight: complementing oracles has no effect

To preserve (co-)semi-decidability, one needs a more restricted form of reduction:

**Definition 4.9:** A language  $\mathbf{P}$  is **many-one reducible** to a language  $\mathbf{Q}$ , written  $\mathbf{P} \leq_m \mathbf{Q}$  if there exists a total computable function  $f : \Sigma^* \rightarrow \Sigma^*$  such that, for all  $w \in \Sigma^*$ :

$$w \in \mathbf{P} \quad \text{if and only if} \quad f(w) \in \mathbf{Q}.$$

This is sometimes called a **mapping-reduction** or an **m-reduction**.



# Properties of Many-One-Reductions

Many-one reductions are special kinds of Turing reductions:

**Theorem 4.10:** If  $\mathbf{P} \leq_m \mathbf{Q}$  then  $\mathbf{P} \leq_T \mathbf{Q}$ .

**Proof:** We obtain an OTM with oracle  $\mathbf{Q}$  that recognises  $\mathbf{P}$  as follows:

- On input  $w$ , compute  $f(w)$
- Call the oracle and return its result (yes = accept; no = reject) □

An easy consequence of Theorem 3.17 therefore is:

**Corollary 4.11:** If  $\mathbf{P} \leq_m \mathbf{Q}$  and  $\mathbf{Q}$  is decidable, then  $\mathbf{P}$  is decidable.

However, now we also have the following:

**Theorem 4.12:** If  $\mathbf{P} \leq_m \mathbf{Q}$  and  $\mathbf{Q}$  is semi-decidable, then  $\mathbf{P}$  is semi-decidable.

**Proof:** Given a TM that recognises  $\mathbf{Q}$ , we obtain a TM that recognises  $\mathbf{P}$  as follows:

- On input  $w$ , compute  $f(w)$
- Simulate the TM for  $\mathbf{Q}$  and return the result (if any) □

## Example: Many-one Reduction

Some of our previous Turing-reductions can easily be described as many-one, e.g., Halting can be many-one reduced to  $\varepsilon$ -Halting. Here is another example:

**Definition 4.13:** Two TMs  $\mathcal{M}$  and  $\mathcal{N}$  are **equivalent** if  $\mathbf{L}(\mathcal{M}) = \mathbf{L}(\mathcal{N})$ .

**Theorem 4.14:** Equivalence of Turing machines is undecidable.

(Note that we could also get this from Rice's Theorem, but we want to try out our new machinery.)

**Proof:** We define  $f$  such that  $w \in \varepsilon$ -Halting iff  $f(w) \in$  Equivalence.

Let  $\mathcal{M}_a$  be a TM that accepts all inputs.

For a TM  $\mathcal{M}$ , we define the following TM  $\mathcal{M}^*$ :

- Simulate  $\mathcal{M}$  on the empty input.
- If  $\mathcal{M}$  halts, accept.

Then  $\mathcal{M}^*$  is equivalent to  $\mathcal{M}_a$  iff  $\mathcal{M}$  halts on the empty input. We define  $f$ :

$$f(w) = \begin{cases} \langle \mathcal{M}^*, \mathcal{M}_a \rangle & \text{if } w = \langle \mathcal{M} \rangle \\ \varepsilon & \text{(an invalid input) if } w \text{ is no encoded TM} \end{cases} \quad \square$$

# Equivalence is Hard

We can show a somewhat stronger result:

**Theorem 4.15:** Equivalence of Turing machines is neither semi-decidable nor co-semi-decidable.

**Proof:** We have already shown  $\varepsilon\text{-Halting} \leq_m \text{Equivalence}$ . Since we know that  $\varepsilon\text{-Halting}$  is not co-semi-decidable (similar to Halting), we conclude that Equivalence is neither.

However, we can also show that  $\overline{\varepsilon\text{-Halting}} \leq_m \text{Equivalence}$ .

- Note that the TM  $\mathcal{M}^*$  defined on the previous slide either accepts all inputs (if  $\mathcal{M}$  halts on  $\varepsilon$ ) or none (if it doesn't)
- Equivalence to  $\mathcal{M}_a$  corresponds to  $\varepsilon\text{-Halting}$
- On the other hand, equivalence to a TM  $\mathcal{M}_\emptyset$ , which rejects all inputs, corresponds to  $\varepsilon\text{-non-Halting}$

We can therefore use the reduction  $f$ :

$$f(w) = \begin{cases} \langle \mathcal{M}^*, \mathcal{M}_\emptyset \rangle & \text{if } w = \langle \mathcal{M} \rangle \\ \langle \mathcal{M}_\emptyset, \mathcal{M}_\emptyset \rangle & \text{(equivalent TMs) if } w \text{ is no encoded TM (since then } w \in \overline{\varepsilon\text{-Halting}} \end{cases}$$



# Recursion

# A Paradox

## A Paradox in the Study of Life:

- (1) Living things are machines.
- (2) Living things can reproduce.
- (3) Machines cannot reproduce.

## Rationale:

- (1) Viewpoint of modern biology.
- (2) Evident.
- (3) If a machine  $A$  produces a machine  $B$ , then  $A$  must be more complex than  $B$ . For example, a car-producing factory is **more complex** than the cars it produces, as it contains the design of the cars and, **in addition**, the design of all manufacturing robots, among others. Since no machine is more complex than itself, a machine cannot reproduce itself.

# Resolving the Paradox

## A Paradox in the Study of Life:

- (1) Living things are machines.
- (2) Living things can reproduce.
- (3) Machines cannot reproduce.

**Question:** How to resolve this paradox?

**Answer:** Assertion (3) is wrong.

In particular, the underlying argument of “more information” and “greater complexity” needed by the producing machine is flawed: there are TMs that reproduce themselves

# Quines

Reproduction of TMs is closely related to the task of creating a program that prints its own source code:

**Definition 4.16:** A **quine** is a program that, when started without any input, will print out its own source code, and then stop.

Can Quines be created? How?

**Example 4.17 (A quine in English):** Print this sentence.

However, we cannot turn this into a program, since “this sentence” does often not correspond to available programming constructs.

**Example 4.18 (Another quine in English):** Print the following sentence twice, the second time in quotes. "Print the following sentence twice, the second time in quotes."

## Some Real Quines

**Example 4.19 (A classic C quine):** `main()char *c="main()char *c=%c%s%c;printf(c,34,c,34);" ;printf(c,34,c,34);`

**Example 4.20 (The shortest C quine, by Szymon Rusinkiewicz):**

**Example 4.21 (A Python quine by Frank Stajano):**

```
l='l=%s;print l%'l'' ;print l%'l'
```

**Note:** A variation are **ouroboros quines** that print out another program that prints out the original again. More steps are possible. See, e.g., <https://github.com/mame/quine-relay> for one with 100 steps.

Other variations exist (see Wikipedia).



## Towards a TM Quine

We define a TM SELF that ignores its input and prints out a description of itself.  
(A TM quine, where “source code” is interpreted as “encoding of the TM”)

The following small result is helpful:

**Lemma 4.22:** There is a computable function  $q : \Sigma^* \rightarrow \Sigma^*$  such that, for each  $w \in \Sigma^*$ , the word  $q(w)$  is (the encoding of) a TM that prints  $w$  and halts.

**Proof:** For any word  $w$ , let  $\mathcal{P}_w$  be a TM that replaces the tape contents with the word  $w$  (clearly, this can easily be found for any  $w$ ).

Now  $q$  is simply computed by a TM that, given  $w$  as input, constructs  $\mathcal{P}_w$  and then computes and outputs  $\langle \mathcal{P}_w \rangle$ . □

**Intuition:** If we were using another programming language, the TM  $\mathcal{P}_w$  might be, e.g., `print(w)`, and the function we seek would simply turn input string  $w$  into output string `"print(w)"`.

# Defining the TM SELF

Like other quines, SELF consists of two parts:

- A Compute the “source code”  $\langle B \rangle$  of a suitable program  $B$
- B Use  $\langle B \rangle$  to print out:
  - (1) source code  $\langle A \rangle$  that computes  $\langle B \rangle$  and (2) the source code  $\langle B \rangle$  itself

We know how to implement part A: use the TM  $\mathcal{P}_{\langle B \rangle}$

(however, to actually do this, we need to know  $B$  first)

$B$  in turn can work as follows:

Given some input string  $\langle M \rangle$ :

- compute  $q(\langle M \rangle)$
- concatenate the TMs given by  $q(\langle M \rangle)$  and  $\langle M \rangle$   
(take a disjoint union of states where any halting state of  $q(\langle M \rangle)$  gets a transition to the starting state of  $\langle M \rangle$ )
- output the encoding of the resulting machine

Then part  $B$  does not depend on  $A$ , so we can really define  $A$  as  $\mathcal{P}_{\langle B \rangle}$

# Summary: SELF TM

So how did we construct our TM quine now?

**Step 1:** We define some TM  $B$  that behaves as follows:

Given some input string  $\langle M \rangle$ :

- compute  $q(\langle M \rangle)$
- concatenate the TMs given by  $q(\langle M \rangle)$  and  $\langle M \rangle$   
(take a disjoint union of states where any halting state of  $q(\langle M \rangle)$  gets a transition to the starting state of  $\langle M \rangle$ )
- output the encoding of the resulting machine

**Step 2:** We define SELF to be the TM constructed by  $B$  on input  $\langle B \rangle$

**Exercise:** Use this recipe to create a quine in your favourite programming language (or just use Python). What is the equivalent of “TM concatenation” here? Also note that the function  $q$  is often more complicated than one might think, due to character escaping.

# The Recursion Theorem

Going further, we can allow any TM to access its own description during the computation:

**Theorem 4.23 (Recursion Theorem):** Let  $t : \Sigma^* \times \Sigma^* \rightarrow \Sigma^*$  be a function computed by some TM  $\mathcal{T}$  (assuming a suitable encoding of pairs of words over  $\Sigma^*$ ). Then there is a TM  $\mathcal{R}$  that computes a function  $r : \Sigma^* \rightarrow \Sigma^*$  such that

$$r(w) = t(\langle \mathcal{R} \rangle, w)$$

for every  $w \in \Sigma^*$ .

**Intuition:** To make a TM that can use its own description, we first devise a TM  $\mathcal{T}$  (to compute  $t$ ) that receives the description of a machine as extra input. The theorem yields a TM  $\mathcal{R}$  that operates like  $\mathcal{T}$  does but with  $\mathcal{R}$ 's description filled in automatically.

# The Recursion Theorem: Proof

**Theorem 4.23 (Recursion Theorem):** Let  $t : \Sigma^* \times \Sigma^* \rightarrow \Sigma^*$  be a function computed by some TM  $\mathcal{T}$  (assuming a suitable encoding of pairs of words over  $\Sigma^*$ ). Then there is a TM  $\mathcal{R}$  that computes a function  $r : \Sigma^* \rightarrow \Sigma^*$  such that  $r(w) = t(\langle \mathcal{R} \rangle, w)$  for every  $w \in \Sigma^*$ .

**Proof:** The proof is similar to the construction of SELF, using a TM with three parts  $A$ ,  $B$  and  $T$ :

- $A$ : print  $\langle BT \rangle$  (like  $\mathcal{P}_{\langle BT \rangle}$  but without deleting the input)

we use  $BT$  to denote the concatenation of the TM parts  $B$  and  $T$  in one TM

- $B$ : on an input of form  $w\langle M \rangle$ , replace  $\langle M \rangle$  by an encoding of the concatenation of  $q'(\langle M \rangle)$  and  $\langle M \rangle$

where  $q'(v)$  is like  $q$  but returns a TM that adds  $v$  at the end of the tape

- $T$ : run  $\mathcal{T}$  on an input of form  $w\langle N \rangle$

We assume here that our TM encoding can be written next to the input  $w$  without risk of confusion. Then  $\mathcal{R}$  is the TM obtained as the concatenation of  $A$ ,  $B$ , and  $T$ .

We can get an encoding of this TM by running  $B$  on input  $\varepsilon\langle BT \rangle$ .

□

# Using the Recursion Theorem

By the Recursion Theorem, we can now use instructions like “obtain own description  $\langle \mathcal{M} \rangle$ ” in our informal descriptions of TMs.

**Example 4.24:** We can describe a TM quine in the style of our previous SELF as follows:

On any input:

- Obtain own description  $\langle \mathcal{M} \rangle$
- Print  $\langle \mathcal{M} \rangle$

We can construct such a TM by applying the Recursion Theorem to the TM  $\mathcal{T}$  described as follows:

On input  $\langle w, \mathcal{M} \rangle$ , print  $\langle \mathcal{M} \rangle$

The Recursion Theorem turns this into a TM  $\mathcal{R}$  that is a quine.

# Halting is Undecidable: Proof by Introspection

We can also use the Recursion Theorem for alternative proofs:

**Theorem 3.11** The Halting Problem  $\mathbf{P}_{\text{Halt}}$  is undecidable.

**Proof:** By contradiction: Suppose there is a decider  $\mathcal{H}$  for the Halting Problem

We construct a TM  $\mathcal{M}$  that, on input  $w$ , acts as follows:

- (1) Obtain own description  $\langle \mathcal{M} \rangle$
- (2) Simulate  $\mathcal{H}$  on input  $\langle \mathcal{M} \rangle \#\# \langle w \rangle$ , that is, check if  $\mathcal{M}$  halts on  $w$
- (3) If yes, enter an infinite loop;  
if no, halt and accept

Then  $\mathcal{M}$  halts on  $w$  if and only if it doesn't – contradiction. □

# Minimal TMs

**Definition 4.25:** A TM  $\mathcal{M}$  is called **minimal** if there is no TM equivalent to  $\mathcal{M}$  that has a shorter description. The problem of deciding if a TM is minimal is:

$$\mathbf{MIN}_{\text{TM}} = \{\langle \mathcal{M} \rangle \mid \mathcal{M} \text{ is a minimal TM}\}$$

**Theorem 4.26:**  $\mathbf{MIN}_{\text{TM}}$  is not Turing-recognisable.

**Proof:** Assume there is some TM  $\mathcal{E}$  enumerating  $\mathbf{MIN}_{\text{TM}}$ .

We define a TM  $\mathcal{C}$  that processes an input  $w$  as follows:

- (1) Obtain own description  $\langle \mathcal{C} \rangle$
- (2) Simulate  $\mathcal{E}$  until some TM  $\mathcal{D}$  is printed such that  $\langle \mathcal{D} \rangle$  is longer than  $\langle \mathcal{C} \rangle$
- (3) Simulate  $\mathcal{D}$  on  $w$

Then  $\mathcal{C}$  is equivalent to  $\mathcal{D}$ , but it has a shorter description, contradicting the assumption that  $\mathcal{D}$  is minimal. □



# Summary and Outlook

Most properties related to the computation of TMs are undecidable

Many-one reductions establish a closer relationship between two problems than Turing reductions

There are non-semi-decidable problems

Turing machines can work with their own description

## **What's next?**

- Defining complexity classes
- Time complexity
- Non-deterministic time