Non-computable functions[*]

In this section we argue that there exist problems for which no algorithm solves all instances in a finite time. The goal is to train to recognize such problems. If in the process of solving a more general problem, one is faced with such an intrinsically hard problem, it is a sign that the goals of the general problem need refinement. Maybe it is enough to only consider instances of the problem that satisfy some additional requirement? Perhaps then, one can obtain a solution that provably works. And perhaps it is also useful in practice. . . ?

At the moment, we still study computability theory. This means that we do not worry about time and space efficiency of the algorithms. We just wonder whether there exists an algorithm. Computability theory is interesting for its negative results, if there is no algorithm at all, there is definitely no practical algorithm that works for all instances. We introduce techniques such as diagonalization and reduction. These techniques will also be used in the part about computational complexity.

1 Diagonalization and the Halting problem

The simplest and historically first use of the diagonalization argument was to show that there exist uncountable sets. We first explain what we mean by this.

There exist two types of infinite sets. A set is *countable* if there exists a list that contains all the elements of the set (in any order, possibly with repetitions). The most obvious example of a countably infinite set is the set of natural numbers: they all appear in the list $1, 2, 3, 4, \dots$ Many other sets are countable:

Integer numbers	0	-1	1	-2	2	-3	3	$\dots$
Pairs of natural numbers		$(1,1)$	$(1,2)$	$(2,1)$	$(3,1)$	$(2,2)$	$(1,3)$	$\dots$
Rational numbers	0	1/1	1/2	2/1	3/1	2/2	1/3	$\dots$
Bitstrings	ε	0	1	00	10	01	11	$\dots$

Figure 1: Countable sets

For example, it is easy to believe that every bitstring x appears in the list of bitstrings; we can even specify its position by the formula $2^n + \sum_{i=1}^{n} x_i 2^{i-1}$, where $x = x_1 \dots x_n$. Now, one might wonder whether this is possible for real numbers . . .

Exercise 1. If A and B are countable, show that $A \cup B$, $A \times B$, A^k for all k and A^* are all countable. Show that a subset of a countable set is also countable.

All sets whose elements can be encoded as bitstrings, are countable because the set of bitstrings is countable. For example, there are countably many computable functions, because for a

[*]Theoretical Computer Science 2016-2017, National Research University Higher School of Economics, Version of December 9, 2016.

universal function φ, the one-argument functions are all in the list $\varphi_1, \varphi_2, \ldots$ For similar reasons there are countably many decidable sets.

It turns out that not all sets are countable and to prove the following theorem, we use diagonalization.

Theorem 1. *The set of infinite binary sequences is not countable.*

The main idea is illustrated by a funny story for children (also used in bars late in the night). Imagine there is a village. The people living in the village are conveniently called *villagers*. In the village works a barber for which the following rule holds:

The barber razes every villager that does not raze himself.

Does the barber raze himself? If he razes himself, he should not raise himself, and if he does not raise himself, he should raise himself. A contradiction. The contradiction can only be avoided if we assume that the barber is not a villager.

$$\begin{array}{r|l}
1 & \mathbf{1}\ 0\ 1\ 1\ 1\ldots \\
2 & 0\ \mathbf{1}\ 0\ 0\ 1\ldots \\
3 & 0\ 1\ \mathbf{1}\ 1\ 0\ldots \\
4 & 0\ 0\ 0\ \mathbf{0}\ 1\ldots \\
5 & 0\ 1\ 1\ 0\ \mathbf{0}\ldots \\
 & \\
\text{inverse} & \mathbf{0}\ \mathbf{0}\ \mathbf{0}\ \mathbf{1}\ \mathbf{1}\ldots \\
\text{diagonal} &
\end{array}$$

Figure 2: The method of diagonalization

Proof. We need to show that for every list there is a sequence that is not in the list. Consider the sequence containing the inverse of the 1st bit of the 1st sequence, the inverse of the 2nd bit of the 2nd sequence, etc., see figure 2. Suppose that this sequence is in the list. Let i be the index of its row. By construction, the ith bit must be equal to its inverse, and this is impossible. Hence, this sequence does not belong to the list. The theorem is proven. $\square$

With a similar argument we can show that also the set of real numbers in the interval $[0, 1]$ is not countable. Any list of real numbers in $[0, 1]$ defines a list of binary representations. Some real numbers have two binary representations, for example: $0.0111\cdots = 0.1000\ldots$. We assume that the list contains both representations. Theorem 1 gives us a sequence that does not belong to the list, and hence, the corresponding real number can not appear in the list of real numbers.

With each binary function $f : \mathbb{N} \to \{0, 1\}$ we can associate an infinite bitsequence $f(1)f(2)f(3)\ldots$. This function is one-to-one. Because there are countably many Turing machines, there exists a list of binary sequences that correspond to computable functions. Theorem 1 gives us a sequence that does not belong to this list, and hence, gives us a function that is not computable.

Theorem 2. *There exists a binary function $f : \mathbb{N} \to \{0, 1\}$ that is not computable.*

In fact, most binary functions are not computable.

Theorem 3. *If a function $f : \mathbb{N} \to \{0,1\}$ is determined by selecting $f(1)$, $f(2)$, $\ldots$ independently and uniformly randomly, then f is computable with probability zero.*

Proof. We need to show that for each $\ell \in \mathbb{N}$, the probability that f is computable is at most $2^{-\ell}$. Let $f_1, f_2, \ldots$ be a list of all computable functions. With probability at most $2^{-\ell-1}$, f equals f_1 on the values $1, \ldots, \ell+1$. With probability at most $2^{-\ell-2}$, f equals f_2 on the values $1, \ldots, \ell+2$. And so on. By the union bound, f equals one of the functions $f_1, f_2, \ldots$ with probability at most $2^{-\ell-1} + 2^{-\ell-2} + \cdots = 2^{-\ell}$. □

Exercise 2. Prove that there exists a language $L \subseteq \{0,1\}$ that is not decidable. Argue that most languages over $\{0,1\}$ are not decidable.

Exercise 3. Let $\varphi : \{0,1\}^* \times \mathbb{N} \to \mathbb{N}$ be a computable function. Show that there exists a computable function f such that for all x: $f \neq \varphi(x, \cdot)$.

Imagine there exists a programming language and a code checker. This checker inspects some code in this language and verifies whether it computes a total one-argument function. Imagine this verifier is computable and correct for all possible codes. Argue that there exists a computable function that can not be implemented in this language. (Hint: generate a list of all one-argument functions that can be implemented by applying the checker to all possible codes in parallel.)

We now present a more interesting set that is not decidable. For a Turing machine M, let $\langle M \rangle$ represent a reasonable encoding[1] of M. If we sort all codes of the machines lexicographically, we obtain a computable sequence of machines $M_1, M_2, \ldots$

Theorem 4. *The following set is undecidable*

$$\mathrm{H}_{\mathrm{TM}} = \{\langle M, w \rangle : M \textit{ halts on input } w\}$$

Proof. [2] Let us look at the table of all (M, x) on which M $\underline{\text{h}}$alts or M $\underline{\text{r}}$uns forever.

	ε	0	1	00	$\ldots$
M_1	$\underline{\mathtt{h}}$	r	r	h	$\ldots$
M_2	h	$\underline{\mathtt{r}}$	h	h	$\ldots$
M_3	r	r	$\underline{\mathtt{h}}$	h	$\ldots$
M_4	r	h	h	$\underline{\mathtt{r}}$	$\ldots$
$\ldots$					
inverse diagonal	$\underline{\mathtt{r}}$	$\underline{\mathtt{h}}$	$\underline{\mathtt{r}}$	$\underline{\mathtt{h}}$	$\ldots$

If a sequence of letters `h` and `r` describes the behaviour of a Turing machine, then there must be a row in the table that coincides with this sequence. Assume that the table above is computable. Then one can construct a machine that behaves like the inverse diagonal. But this sequence can not be a row of the table, a contradiction; the assumption must be false: the table is not computable. In other words, the set of pairs (M, w) such that M halts on w is not decidable. □

[1] With this we mean that the set of all strings that are the code of some machine is decidable, and that from $\langle M \rangle, w, t$ we can compute the state of M on input w after t computation steps.

[2] A funny visual animation of this proof can be found here:
`http://www.youtube.com/watch?v=92WHN-pAFCs&t=100`

Exercise 4. An extension of a partial function g is a partial function which equals g on all arguments for which g is defined. Show that there exists a partial computable binary function that has no computable extension. (Hint: choose g such that $g(\langle M\rangle) = 1 - M(\langle M\rangle)$ if $M(\langle M\rangle) \in \{0,1\}$.)

2 Recognizable sets

The halting set is not decidable, but its elements are recognizable: if an element belongs to the set, a machine can verify this. On the other hand, there might not be a way to certify that an element is outside the set.

Definition 5. *A set is* recognizable[3] *if there exists a recognizer that accepts precisely the strings of the set.*

Observe that the halting set is recognizable.

Exercise 5. Show that if a set is recognized by a deterministic recognizer, then it is the domain of some partial computable function.

Exercise 6. Show that a set is recognized by a deterministic recognizer if and only if it is recognized by a nondeterministic recognizer. Do you do depth-first or breadth-first search? Do both work?

Exercise 7. show that the following are equivalent for a nonempty set A:

1. A is recognizable,
2. A is the image of a computable function,
3. There exists a computable predicate (i.e., a function $f : \mathbb{N} \times \mathbb{N} \to \{\texttt{True, False}\}$), such that $A \in n \Leftrightarrow \exists t : f(t,n)$.

The following exercise shows that computable sets can also be characterized by images of computable functions.

Exercise 8. Associate bitstrings with natural numbers. Show that a set of natural numbers is decidable if and only if it is the image of an increasing function.

Exercise 9. Show that:

1. The union and intersection of two recognizable sets are recognizable.
2. The join $A \oplus B$ of two sets in $\{0,1\}^*$ is given by $\{0x : x \in A\} \cup \{1x : x \in B\}$. Show that the join of two recognizable sets is also recognizable.
3. If a set $A \subseteq \{0,1\}^*$ and its complement $\{0,1\}^* \setminus A$ are recognizable, then A is decidable.
4. There exists a set that is not recognizable for which the complement is also not recognizable.
5. Adapt the proof of Theorem 3 to show that for most binary functions f, the set $\{n : f(n) = 1\}$ is not recognizable.

[3] Also called *enumerble*

3 Mapping reducibility

This section is inspired by chapter 5 in Sipser's book. On page 171 we find a good informal description.

> A *reduction* is a way of converting one problem to another problem such that a solution for the second problem can be used to solve the first problem.
>
> Such reductions come up in everyday life, even if we don't usually refer to them this way. For example, suppose that you want to find your way around in a new city. You know that this would be easy if you had a map. Thus you can reduce the problem of finding your way around to the problem of obtaining a map of the city.
>
> Reducibility always involves two problems, which we call A and B. If A reduces to B, we can use a solution for B to solve A. So in our example, A is the problem of finding your way around the city, and B is the problem of obtaining a map.

3.1 Implicit examples

For a Turing machine M, let $\langle M \rangle$ be a reasonable[4] encoding of M. In this section we also use $\langle M, w \rangle$ to encode a pair of a Turing machine M and a string w, or $\langle M, M' \rangle$ for a pair of two machines. We only use deterministic Turing machines.

Theorem 6. $\mathrm{E_{TM}} = \{\langle M \rangle : M \text{ accepts no strings}\}$ *is undecidable.*

Proof. We show that if this set were decidable, then also $\mathrm{H_{TM}}$ would be decidable. This contradicts Theorem 4, hence $\mathrm{E_{TM}}$ can not be decidable. For this we use Turing machines $P_{M,w}$ that have a machine M and a string w hardwired in it. On input x, machine $P_{M,w}$

- simulates M on input w,
- if M halts then it accepts x (otherwise, it keeps simulating M forever).

Thus $P_{M,w}$ either accepts no strings or all strings, depending on whether M halts on input w. Assume $\mathrm{E_{TM}}$ is decidable and let R be a decider of $\mathrm{E_{TM}}$. The following algorithm defines a decider for $\mathrm{H_{TM}}$:

```
Data: ⟨M, w⟩, where M is a Turing machine and w a string.
Result: Accepts if M halts on input w, rejects otherwise.
    • Compute a description of ⟨$P_{M,w}$⟩.
    • Check whether ⟨$P_{M,w}$⟩ ∈ $\mathrm{E_{TM}}$ by running R.
    • Accept if R accepts, reject if R rejects.
```

This algorithm always halts because R is a decider. Recall that $P_{M,w} \in \mathrm{E_{TM}}$ if and only M does not halt on input w. Thus, the algorithm above accepts if and only if M halts on input w. The algorithm above is a decider for $\mathrm{H_{TM}}$. On the other hand, by Theorem 4, $\mathrm{H_{TM}}$ has no decider, a contradiction. Our assumption must be false, it is, $\mathrm{E_{TM}}$ can not be computable. □

[4] This means that there exists a computable function that maps $\langle M \rangle$, some input strings $(u, v, \ldots, z)$ and $t \in \mathbb{N}$ to a description of the state the machine after t computation steps. Also, the set $\{\langle M \rangle : M \text{ is a Turing machine}\}$ should be decidable.

Theorem 7. *Let $H_M = \{w : M \text{ halts on input } w\}$. The following set is undecidable*

$$\textsc{DomIsReg}_{\mathrm{TM}} = \{\langle M\rangle : H_M \text{ is a regular set}\}.$$

Proof. We use the same technique as the previous theorem, we show that if $\textsc{DomIsReg}_{\mathrm{TM}}$ were decidable then also H_{TM} is decidable. Let NR be a set that is not regular, for example 0^n1^n. We use machines $P_{M,w}$ that have a machine M and a string w hardwired in them. This time on input x, the machine $P_{M,w}$

- Checks whether $x \in \mathrm{NR}$, and accepts if this is true.
- Otherwise, it simulates M on input w.
- If M halts, then $P_{M,w}$ accepts x, otherwise $P_{M,w}$ runs forever.

Assume $\textsc{DomIsReg}_{\mathrm{TM}}$ is decidable. Let R be a decider of this set. Consider a machine that executes the following algorithm.

```
Data: $\langle M, w\rangle$ where $M$ is a Turing machine and $w$ a string.
Result: Accepts if $M$ halts on input $w$, rejects otherwise.
    • Compute a description of $P_{M,w}$.
    • Check whether $\langle P_{M,w}\rangle \in \textsc{DomIsReg}_{\mathrm{TM}}$ by running $R$.
    • Reject if $R$ accepts, accept if $R$ rejects.
```

Clearly, this machine is a decider. Moreover, it accepts if and only if $P_{M,w}$ is regular; and this is true if and only if $\langle M, w\rangle \in \mathrm{H}_{\mathrm{TM}}$. Hence, this machine decides H_{TM}. But such a decider can not exist, and our assumption must be false. $\textsc{DomIsReg}_{\mathrm{TM}}$ is not decidable. $\square$

In one of the last exercises of the previous lecture we defined linear bounded automata. We showed that the set $\mathrm{A}_{\mathrm{LBA}}$ is decidable. We show now that testing whether the language of an LBA is empty, can not be done in a computable way.

Theorem 8. $\mathrm{E}_{\mathrm{LBA}} = \{\langle M\rangle : M \text{ accepts no strings}\}$ *is undecidable.*

Proof. We use the same technique. Let $P_{M,w}$ be an LBA that has a machine M and a string w hardwired in it. On input x, machine $P_{M,w}$

- Erases x from the tape, and simulates M on input w. If this simulation requires space beyond the size of x, then $P_{M,w}$ rejects.
- Otherwise, if the simulation halts, $P_{M,w}$ accepts.

Note that if M halts on input w, then for large x the simulation will have enough space and will eventually halt. If this is not the case, $P_{M,w}$ accepts no strings. Thus $\langle P_{M,w}\rangle \in \mathrm{E}_{\mathrm{LBA}}$ if and only if $\langle M, w\rangle \notin \mathrm{H}_{\mathrm{TM}}$.

Assume $\mathrm{E}_{\mathrm{LBA}}$ is decidable. Let R be a decider of $\mathrm{E}_{\mathrm{LBA}}$. Consider a machine that executes the following algorithm.

Data: $\langle M, w\rangle$ where M is a Turing machine and w a string.
Result: Accepts if M halts on input w, rejects otherwise.

- Compute a description of $P_{M,w}$.
- Check whether $\langle P_{M,w}\rangle \in \mathrm{E}_{\mathrm{LBA}}$ by running R.
- *Reject* if R accepts, *accept* if R rejects.

This machine decides H_{TM}, a contradiction. □

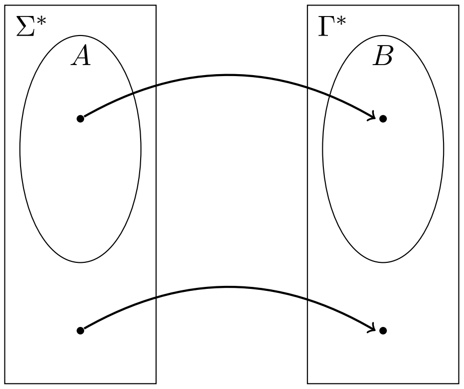
3.2 Definition

The three proofs in the previous section have a common structure: each time we code instances from the set H_{TM} in the elements of the set that is undecidable. We "reduce" H_{TM} to the other sets.

Definition 9. *Language A over an alphabet Σ* mapping reduces *to a language B over an alphabet Γ, notation $A \leq_m B$ if there exists a computable function $f : \Sigma^* \to \Gamma^*$ such that for all $w \in \Sigma^*$*

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

The function f is called the reduction.



Theorem 10.
If $A \leq_m B$ and B is decidable, then A is decidable.
If $A \leq_m B$ and B is recognizable, then A is recognizable.

Exercise 10. Proof the theorem above.

This gives us a method to show that a set is undecidable. If $A \leq_m B$ and A is undecidable, then B is undecidable. This is what we did in the three examples of the previous section. We reduced H_{TM} respectively to E_{TM}, $\textsc{DomIsReg}_{\mathrm{TM}}$ and the complement of $\mathrm{E}_{\mathrm{LBA}}$

3.3 m-complete sets

The m-complete sets are the recognizable sets that are in some sense the most difficult to recognize.

Definition 11. *A set is* m-complete *if it is recognizable and all recognizable sets reduce to it.*

Let $\mathrm{A_{TM}} = \{\langle M, w\rangle : M \text{ accepts } w\}$.

Theorem 12. *The set* $\mathrm{A_{TM}}$ *is m-complete.*

Proof. First we show that $\mathrm{A_{TM}}$ is recognizable. Indeed, there exists a machine that on input $\langle M, w\rangle$ simulates M on input w and accepts if the simulated machine accepts.

Every recognizable language can be reduced to this machine. Indeed, if U recognizes a language L, then the function f_U for which $f_U(w) = \langle U, w\rangle$ is a reduction from L to $\mathrm{A_{TM}}$. □

Exercise 11. Prove the statement or provide a counter example.

1. If A is m-complete and $A \le_m B$ for some recognizable B, is B m-complete?
2. If A and B are m-complete, is $A \cap B$ also m-complete?
3. Recall the definition of $\oplus$ form exercise 9.
   If A is m-complete and B is recognizable, is $A \oplus B$ m-complete?

Theorem 13. *The set* $\mathrm{H_{TM}}$ *is m-complete.*

Proof. Observe that $\mathrm{H_{TM}}$ is recognizable. We show that $\mathrm{A_{TM}} \le_m \mathrm{H_{TM}}$ and the theorem follows from exercise 11.1. For a Turing machine M, let P_M be a machine that on input w simulates $M(w)$ and

- if $M(w)$ halts and accepts, then P_M halts,
- if $M(w)$ halts and rejects, then P_M goes to an infinite loop,
- if $M(w)$ does not halt, then P_M simulates M indefinitely.

There is a computable function f that for every M maps $\langle M, w\rangle$ to $\langle P_M, w\rangle$. This function is a reduction from $\mathrm{A_{TM}} \le_m \mathrm{H_{TM}}$. □

All the enumerable sets we have seen so far are either computable or m-complete. This seems also to be true for all natural examples of sets encountered outside computability and mathematical logic. One might wonder whether this is always the case. It has been shown that there exist recognizable sets that are neither decidable nor m-complete.

3.4 Rice's theorem

Given a non-trivial property of a function, Rice theorem says that by looking at a description of a Turing machine it is impossible to tell whether it computes a function with this property or not. In other words, for each non-trivial property of input-output relations, and for each method that inspects Turing machines, there are always machines for which the method is unable to decide whether the machine has the property. With a *trivial* property we mean that either no Turing machine has has the property or all machines have it.

Theorem 14. *Let F be a set of one argument functions.*

$$I = \{\langle M\rangle : M \textit{ is a Turing machine that computes a function in } F\}$$

is either empty, equals the set of all descriptions of Turing machines, or is undecidable.

Proof. First assume that the everywhere-undefined-function does not belong to F. If F contains no partial computable functions, then I is empty and nothing needs to be proven. Otherwise, let h be a partial computable function in F. Let $P_{M,w}$ be a Turing machine that evaluates a function

$$x \to \begin{cases} h(x) & \text{if } M \text{ halts on input } w, \\ \text{undefined} & \text{otherwise.} \end{cases}$$

There is a computable function f that maps $\langle M, w\rangle$ to $\langle P_{M,w}\rangle$. f is a reduction from H_{TM} to I. Hence, I is undecidable.

The case where the everywhere undefined function does belong to F is proven in a similar way. If F contains all partial computable functions, then nothing needs to be shown. Otherwise, we choosing h to be outside F. Now the function above is a reduction from H_{TM} to the complement of I. Hence, I is undecidable. □

Observe that the statement of the theorem is symmetric: after replacing F by its complement, we obtain a theorem that is equivalent. On the other hand, the proof above has two cases and in each case this asymmetry has disappeared. Below we give a proof that preserves this symmetry.

Second proof. In exercise 4 it is shown that there exists a partial computable binary function g without computable extension. Assume that there are partial computable functions h_{in} in F and h_{out} outside F. If at least one of these functions does not exist, then nothing needs to be shown. Let P_r be a machine that on input x computes

$$\begin{cases} h_{\text{in}}(x) & \text{if } g(r) = 1 \\ h_{\text{out}}(x) & \text{if } g(r) = 0 \\ \text{undefined} & \text{otherwise.} \end{cases}$$

We show that if I were decidable, then we can construct a computable extension $\tilde{g}$ of g.

$$\tilde{g}(n) = \begin{cases} 1 & \text{if } \langle P_r\rangle \in I \\ 0 & \text{if } \langle P_r\rangle \notin I. \end{cases}$$

Observe that $\tilde{g}$ is computable and is an extension of g. On the other hand, by choice of g, such a computable function can not exist. A contradiction. The theorem is proven. □

Rice's theorem implies that for each partial computable function, there are infinitely many other Turing machines that compute the same function. Indeed, every finite set is decidable, and for some f we can apply Rice's with $F = \{f\}$. We can associate Turing machines with programs for some universal programming language. Then we conclude that in each such language, there are infinitely many programs that compute the same function.

4 Undecidable problems

By now, it should not be difficult to prove that the set of Turing machines that halt on empty input is m-complete, in other words,

$$\mathrm{H}_{\varepsilon} = \{\langle M\rangle : M \text{ halts on input } \varepsilon\}$$

is m-complete and hence undecidable.
In this section we show that several problems are undecidable. Usually, we do this by reduction from H_ε.

It is funny to note that also LaTeX can simulate Turing machines, see here for an implementation of a Turing machine:
`http://en.literateprograms.org/Turing_machine_simulator_(LaTeX)` Hence, the set of halting latex programs is undecidable.

4.1 Integer polynomials that have integer roots

Definition 15. *A set $S \subseteq \mathbb{N}$ is* Diophantine *if there exists a polynomial with integer coefficients $P(n, x_1, \ldots, x_e)$ such that*

$$n \in S \quad \Longleftrightarrow \quad \exists x_1, \ldots, x_e \in \mathbb{N}\, [P(n, x_1, \ldots, x_e) = 0]\,.$$

Clearly, a Diophantine set is recognizable. In 1970, Y. Matiyasevich showed that the reverse is also true: every recognizable set is Diophantine.

In 1900, David Hilbert published 23 mathematical problems for the 20th century. The 10th problem asks for a procedure to decide whether a polynomial with integer coefficients has a solution. Matiyasevich's result implies that such a procedure can not exist. Indeed, let $P(n, x_1, \ldots, x_e)$ be a polynomial that corresponds to a recognizable set S that is not decidable. If a procedure for Hilbert's problem existed, then we could decide S by applying the procedure for a fixed value of $\overline{n}$ in $P(\overline{n}, x_1, \ldots, x_e)$. A contradiction.

4.2 FRACTRAN

Exercise 12. Show that a program for any register machine can be rewritten as a program that only uses the following three types of commands:

- $a := a + 1$
- `If` $a = 0$ `then goto line` m `else` $a := a - 1$ `and goto line` n
- `Stop`

The following programming language is called FRACTRAN and was invented by John Conway. Every program consists of a number n and a list of positive rational numbers $f_1, f_2, \ldots, f_e$. The program is executed in steps. At each step, n is replaced by the left most value of $f_1 n, f_2 n, \ldots, f_e n$ that is integer. If none of these numbers are integer, the computation halts. For example, the program given by $n = 5$ and the fractions $[2/3, 6/5, 1/2]$ halts and the subsequent values of n are: 5, 6, 4, 2, 1.

Exercise 13. Show that the set of halting FRACTRAN programs is undecidable. Tip: Encode the variables $a, b, c, \ldots, g$ of a register machine as $2^a 3^b 5^c \ldots p^g$. Associate with each line number ℓ, a prime number q_ℓ different from the prime numbers used to store $a, b, c, \ldots, g$. In each computation step, let n be $q_\ell 2^a 3^b 5^c \ldots p^g$.

4.3 Tag systems

Another simple mechanism capable of computing is a k-tag system. For each $k \geq 1$, a *k-tag system* has as input, a word over an alphabet A. Its operation is described by a partial function $\delta : A \to A^*$ which transforms the input string w step by step. If at some point, w has length less than k or $\delta(w_1)$ is undefined for the first letter w_1 of w, the computation halts (and w is the output). Otherwise, the first k letters of w are deleted, $\delta(w_1)$ is appended at the end of the word and the process is iterated.

For example, consider a 2-tag system over $\{0, 1\}$ with $\delta(0) = 011$ and $\delta(1) = 1$. On input 01111 we obtain:

$$\to \cancel{01}111\mathbf{011} \to \cancel{11}1011\mathbf{1} \to \cancel{10}111\mathbf{1} \to \cancel{11}11\mathbf{1} \to \cancel{11}1\mathbf{1} \to \cancel{11}\mathbf{1}.$$

Exercise 14. Show that for every 1-tag system over A, the set of strings in A^* for which the computation halts is decidable.

Theorem 16. *The following set is undecidable*

$$\{\langle M, w\rangle : M \textit{ is a tag-machine that halts on input } w\}.$$

Proof. We showed already that register machines can simulate Turing machines. We now show that 2-tag machines can simulate register machines. In this way, we show that there is a reduction from H_ε to the set of the theorem.

It suffices to implement the three commands of exercise 12. Let $a, b, \ldots, e$ be the variables of the register machine. For each line ℓ of the program, the tag system has variables $\texttt{a}_\ell, \ldots, \texttt{e}_\ell$ and $\texttt{A}_\ell, \ldots, \texttt{E}_\ell$ (for the lines with an `If` instruction, we need additional symbols). The alphabet also contains a fixed symbol $*$.

The tag-machine is started with the word

$$\texttt{a}_\ell{*}\texttt{A}_\ell{*}\texttt{b}_\ell{*}\texttt{B}_\ell{*}\ldots\texttt{e}_\ell{*}\texttt{E}_\ell{*}$$

In the simulation of a computation step of the register machine at line ℓ, the tag machine is initially in a state

$$\overbrace{\texttt{a}_\ell?\ldots\texttt{a}_\ell?}^{2^a \text{ times}}\,\texttt{A}_\ell?\,\overbrace{\texttt{b}_\ell?\ldots\texttt{b}_\ell?}^{2^b \text{ times}}\,\texttt{B}_\ell? \quad \ldots \quad \overbrace{\texttt{e}_\ell?\ldots\texttt{e}_\ell?}^{2^e \text{ times}}\,\texttt{E}_\ell?$$

At the places of the question marks can be any symbol; these symbols do not influence the computation because they are deleted before they can be scanned. Note that the start state corresponds to a state of the register machine where all variables are empty.

Assume line ℓ is an instruction $b := b + 1$, then for all $\texttt{x} \in \{\texttt{a}, \texttt{A}, \ldots, \texttt{e}, \texttt{E}\} \setminus \{\texttt{b}\}$, we choose

$$\delta(\texttt{x}_\ell) = \texttt{x}_{\ell+1}*$$

and for $\texttt{b}_\ell$ we choose

$$\delta(\texttt{b}_\ell) = \texttt{b}_{\ell+1}{*}\texttt{b}_{\ell+1}*$$

After replacing all symbols with index ℓ, the word of the tag machine is in the correct state to start the execution of line $\ell + 1$.

If a line ℓ contains `Stop`, then δ is undefined for all symbols $\texttt{x}_\ell$. It remains to explain how an `If`-instruction is implemented, and this is the hardest part. Consider a command

ℓ. If $b = 0$ `then goto line` m `else` $b := b - 1$ `and goto line` n

The rough idea. Recall that only the odd positions of the word are scanned. First we decrement b, in other words, we halve the number of symbol $\mathtt{b}_\ell$ by choosing $\delta(\mathtt{b}_\ell) = \dot{\mathtt{b}}_\ell$. If $b = 0$ this will make the length of the string odd, otherwise the length is even. While decrementing, we also make sure that every symbol is repeated twice, i.e., for all $\mathtt{x}_\ell \neq \mathtt{b}_\ell$ we choose $\delta(\mathtt{x}_\ell) = \dot{\mathtt{x}}_\ell\dot{\mathtt{x}}_\ell$. Then we go through the whole string and replace the symbols $\dot{\mathtt{x}}_\ell$ by $\mathtt{x}_n\mathtt{x}_m$. This strings has variables labeled by n in its odd positions, and variables labeled by m in its even positions. Now, the main idea: if the string has odd length, the last replacement of a symbol $\dot{\mathtt{x}}_\ell$ will delete the first symbol $\mathtt{x}_n$. The positions that were previously even, became odd, and they are all labeled by m. If $b > 0$, then this would not happen, and the odd positions still contain symbols labeled by n. Moreover, the string is such that the simulation of the next computation step of the register machine can start.

Now the detailed construction. To simulate a line ℓ that modifies a variable v, we divide the word in 4 parts which are schematically represented as

$$\mathtt{x}_\ell? \ldots \mathtt{x}_\ell?\mathtt{v}_\ell? \ldots \mathtt{v}_\ell?\mathtt{V}_\ell?\mathtt{y}_\ell? \ldots \mathtt{y}_\ell?.$$

The leftmost part is the part that contains everything at the left of the symbols $\mathtt{v}_\ell$, the second part is the part that contains the repetitions of $\mathtt{v}_\ell?$, the third part, is the pair of symbols $\mathtt{V}_\ell?$, and the forth part is everything else.

In its first run through the string, we use copies of the symbols $\mathtt{z}$ given by $\dot{\mathtt{z}}$, $\mathtt{z}^>$ and $\mathtt{z}^=$ and the rules

$$\begin{aligned} \delta(\mathtt{x}_\ell) &= \dot{\mathtt{x}}_\ell* & \qquad \delta(\mathtt{V}_\ell) &= \mathtt{V}^=_\ell\mathtt{V}^>_\ell \\ \delta(\mathtt{v}_\ell) &= \dot{\mathtt{v}}_\ell & \qquad \delta(\mathtt{y}_\ell) &= \mathtt{y}^=_\ell\mathtt{y}^>_\ell. \end{aligned}$$

After scanning all odd positions we obtain the word

$$\dot{\mathtt{x}}_\ell* \ldots \dot{\mathtt{x}}_\ell*\dot{\mathtt{v}}_\ell \ldots \dot{\mathtt{v}}_\ell\mathtt{V}^>_\ell\mathtt{V}^=_\ell\mathtt{y}^>_\ell\mathtt{y}^=_\ell \ldots \mathtt{y}^>_\ell\mathtt{y}^=_\ell.$$

Thus this halves the number of $\mathtt{v}$'s and makes the symbols in odd and even positions different in the two rightmost parts.

Let us now consider the case where $v = 0$. We obtain a string with a single $\dot{\mathtt{v}}_\ell$ symbol. In other words, the state above is

$$\dot{\mathtt{x}}_\ell* \ldots \dot{\mathtt{x}}_\ell*\dot{\mathtt{v}}_\ell\mathtt{V}^>_\ell\mathtt{V}^=_\ell\mathtt{y}^>_\ell\mathtt{y}^=_\ell \ldots \mathtt{y}^>_\ell\mathtt{y}^=_\ell$$

This string has odd length. The second time the tag machine goes through the string, it uses

$$\begin{aligned} \delta(\dot{\mathtt{x}}_\ell) &= \mathtt{x}_n\mathtt{x}_m & \qquad \delta(\mathtt{V}^=_\ell) &= *\mathtt{V}_m* \\ \delta(\dot{\mathtt{v}}_\ell) &= \mathtt{v}_n\mathtt{v}_m & \qquad \delta(\mathtt{y}^=_\ell) &= \mathtt{y}_m*. \end{aligned}$$

The replacement of the last symbol $\mathtt{y}^=_\ell$ deletes the first symbol $\mathtt{x}_n$. We obtain the string

$$\cancel{\mathtt{x}_n}\mathtt{x}_m \ldots \mathtt{x}_n\mathtt{x}_m\mathtt{v}_n\mathtt{v}_m*\mathtt{V}_m*\mathtt{y}_m* \ldots \mathtt{y}_m*$$

The string is now in the form

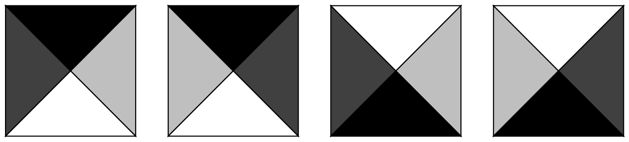
$$\mathtt{x}_m? \dots \mathtt{x}_m?\mathtt{v}_m?\mathtt{V}_m?\mathtt{y}_m? \dots \mathtt{y}_m?$$

and the next step of the simulation can be started. Note that this string represents a register machine where $v = 0$.

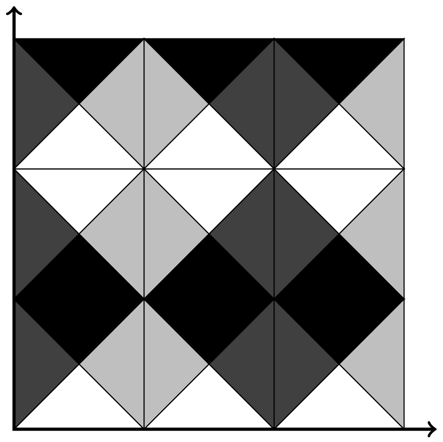
The case $v > 0$ goes in a similar way, and it is even simpler, because the first symbol $\mathtt{x}_n$ is not removed. We choose $\delta(\mathtt{V}^{>}_{\ell}) = \mathtt{V}_n*$ and $\delta(\mathtt{y}^{>}_{\ell}) = \mathtt{y}_n*$. Note that the represented value of v has decreased by one. □

4.4 Tilings

A Wang tile is a square characterized by 4 colors on each side. Here are some examples:



Tiles can not be rotated, only translated. With these tiles, we can fill a quarter of a plane such that touching edges have the same color:



Definition 17. *A* tile set *T is a finite subset of C^4 for some set C. Let* $(\mathrm{N}, \mathrm{E}, \mathrm{S}, \mathrm{W}) = (1, 2, 3, 4)$*. Thus $t \in T$ equals $(t_\mathrm{N}, t_\mathrm{E}, t_\mathrm{S}, t_\mathrm{W})$. A* (T, t)-tiling *is a mapping $f : \mathbb{N} \times \mathbb{N} \to T$ with $f(1, 1) = t$ such that the colours of neighbouring tiles match, i.e., for all $(i, j) \in \mathbb{N} \times \mathbb{N}$, $f(i, j)_\mathrm{E} = f(i + 1, j)_\mathrm{W}$ and $f(i, j)_\mathrm{N} = f(i, j + 1)_\mathrm{S}$.*

Proposition 18. *The set of pairs (T, t) for which no (T, t)-tiling exists is recognizable.*

We say that f is a *(T, t)-tiling for R* for some $R \subseteq \mathbb{N} \times \mathbb{N}$ if neighbouring colours of tiles in R match. The proof of the proposition uses the following lemma.

Lemma 19. *If there exists a (T, t)-tiling for every finite set $R \subseteq \mathbb{N} \times \mathbb{N}$, there exists a (T, t)-tiling of $\mathbb{N} \times \mathbb{N}$.*

Exercise 15. We have defined a tile set T as a finite set. Show that if we consider infinite tile sets there exists a counter example for the above lemma.

Proof of Proposition 18. The recognizer searches for a finite $R \subseteq \mathbb{N} \times \mathbb{N}$ such that there is no (T, t)-tiling for R. If such an R is found, it accepts, otherwise it searchers forever.

Obviously, if there exists a (T, t)-tiling of $\mathbb{N} \times \mathbb{N}$, the algorithm never accepts. On the other hand, if every finite R has a tiling, then Lemma 19 implies that there exists a tiling of the plane. □

It remains to prove Lemma 19. A *tree* is a graph in which any two vertices are connected by precisely one path. We say that a tree is *finitely branching* if every vertex in the tree has finitely many children.

Lemma 20 (König's lemma)**.** *If a finitely branching tree has infinitely many vertices, then it has an infinite path.*

You might find this lemma obvious and skip its proof. However, whether even this lemma is true is a very complicated question. We explain a bit more for the interested reader. The statement of the lemma above is equivalent to the axiom of choice. This axiom is useful to simplify proofs, and is used by many mathematicians without hesitation. It has been shown that the axiom of choice can not be proved or disproved using standard mathematical reasoning.[5] On the other hand, this axiom contradicts the axiom of determinacy, which is another axiom that is useful to simplify proofs and is also used without hesitation.

Fortunately, we only need the lemma for trees with countably many vertices. In this case we can avoid the axiom of choice: we can use a list of vertices (from the proof that there are countably many) to "choose" specific vertices when needed. We now present the details of this proof.

Proof for a rooted tree with countably many vertices. Fix a list L of all vertices. For each subset S of vertices of the tree, let v_S be the element of S that appears first in L.

We construct the path. The first element is the root. By assumption, the root has infinitely many descendants. We use an inductive procedure to extend the path with one element. At each point we assume that the current endpoint e has infinitely many descendants. Because e has finitely many children, at least one of the children has infinitely many descendants. Let S be the set of children with infinitely many descendants. Extended the path with the element v_S (as defined above). The induction step can now be repeated. □

Proof of Lemma 19. Let $R_n = \{1, 2, \ldots, n\} \times \{1, 2, \ldots, n\}$. The set V_n is given by all functions $f : \mathbb{N} \times \mathbb{N} \to T$ that are a (t, T)-tiling for R_n and are undefined outside R_n. Note that each V_n has finitely many elements. Consider the graph on the vertices $\bigcup_{n \in \mathbb{N}} V_n$. There is an edge between $f \in V_n$ and $g \in V_{n+1}$ if g is an extension of f, i.e., if f and g are equal in R_n. This graph is a tree where the root is the single function in V_0 (which equals the function with empty domain).

If there is a tiling for every $R \subseteq \mathbb{N} \times \mathbb{N}$, then this tree has infinitely many vertices. By König's lemma it must have an infinite path. Hence, there are functions $f_1, f_2, \ldots$ that are extensions of each other and are tilings in R_n. Their limit is a Tiling of $\mathbb{N} \times \mathbb{N}$. □

[5] With this we mean Zermelo-Fraenkel set theory. We assume that this set theory is consistent.

Exercise 16. Let Σ be a finite set and let $S \subseteq \Sigma^*$ be a set that contains a string of every length. Show that there exists a sequence $\omega \in \Sigma^\infty$ such that every prefix $\omega_1 \dots \omega_n$ of ω is an element of S. Can you also prove this directly without using König's lemma?

Theorem 21. *The set of pairs (T, t) for which no (T, t)-tiling exists is undecidable.*

We prove this by reducing H_ε to a set of pairs (T, t) for which no tiling exists. We construct the tiles such that a (T, t)-tiling encodes a computation history of a Turing machine. If the computation does not halt on empty input, the tiling fills the whole plane. Otherwise, we let a tile appear that can not be matched by any other tile.

The configurations of the Turing machine at subsequent computation steps are represented by increasing rows in the tiling. Every tile of a row encodes precisely one tape cell. In every row there are two tiles that represents the computation head and a cell. The first represents the head and its cell at the beginning of the computation step. The second tile corresponds to the end of the computation step. Obviously, these two tiles are next to each other in a tiling, see figure 3.

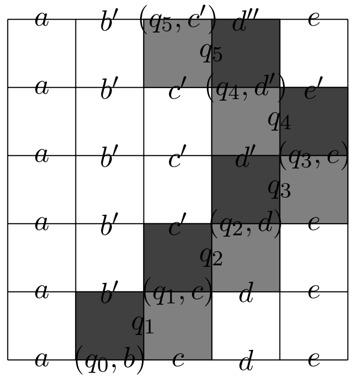


Figure 3: A tiling that simulates a Turing machine going through states $q_0, \dots, q_5$. The machine moves three times right and one times left. Dark gray and light gray tiles represent the computation head at the beginning and end of the computation step.

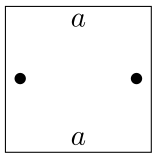
To initialize the computation correctly, we use some additional tiles. These tiles only appear in the first row of a tiling.

Proof of Theorem 21. It is easy to see whether a Turing machine halts immediately on empty input. Such Turing machines are mapped to a fixed tile set that can not fill the plane. Consider a Turing machine $(Q, \{0, 1\}, A, \delta, q_{\text{start}})$ that makes at least one move. The colours of the tile set are given by

$$C = (Q \times \{L, R\}) \cup (Q \times A) \cup A \cup \{\bullet, \square, \diamond, \not{\iota}\}.$$

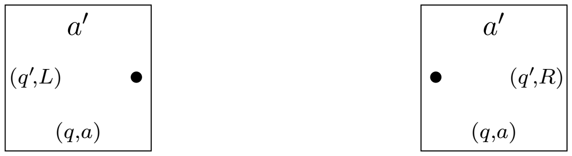
The set T consists of 5 types of tiles.

1. For each tape symbol, we have a tile that represents a cell of the tape on which the computation head is not located:

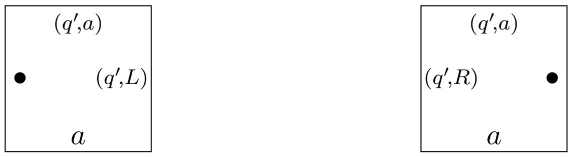


Above and below them should be tiles that encode the same symbol; therefore the colours on the North and the South are chosen to be the same symbol of A.

2. For each $(q, a) \in \operatorname{Dom} \delta$, there is a tile that represents the computation head at the beginning of a computation step with control state q that reads a. Let $(a', m, q') = \delta(q, a)$. This tile also encodes the writing of a', a move in the direction $m \in \{L, R\}$ and a changes to the control state q'. Depending on the value of m, we either add the left $(m = L)$ or the right $(m = R)$ tile to the tile set T:

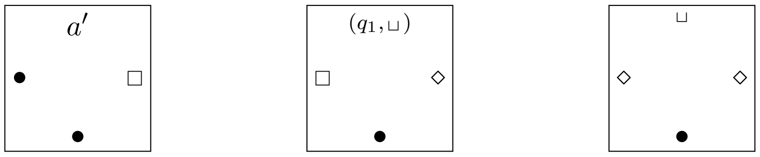


3. For each $(q, a) \in Q \times A$, there is a tile that represents the head at the end of the computation step. For left and right movements they are:



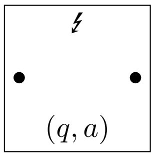
Note that only tiles of this and the above type have colours $Q \times \{L, R\} \subseteq C$, on the left or the right side. This implies that a tile of this type always has exactly one neighbour which is a tile of the previous type and vice versa. Also note that the colours $Q \times A \subseteq C$ only appear in tiles of this and the previous type. (There is only one exception, corresponding to a single tile that can only appear on the first row.) This implies that in each subsequent row, the position of the computation head will be in the desired place. It is not possible that a second computation head is created, because tiles of the previous type can only appear above tiles of this type.

4. The initial configuration is given by the state $q_{\text{start}} \in Q$ and a tape containing only blank symbols $\sqcup \in A$. By assumption, the machine does not halt immediately, and the head must move right. Let a' be the symbol that is written in this step and q_1 be the new state. The force a enforce a correct first row of a tiling, the tile set contains



We choose t to be the leftmost tile.

5. For each $(q, a) \notin \operatorname{Dom}\delta$, (which can only appear in a terminating configuration), we add the tile



The colour ↯ does not appear in the South of any tile.

In a tiling of the plane each row has exactly one tile of the second and third type. Therefore, the control states encoded in these cells correspond to the to the control states of a valid computation history. Hence, if the computation never terminates, this tile set can fill the plane. Otherwise, the tile with the colour ↯ appears. No other tile fits above this tile, hence, in this case there is no (T, t)-tiling. $\square$

There exists a tile set T and a $t \in T$ for which a tiling exists, but for which every such tiling f is a non-computable function. We will prove this result in the two subsequent exercises.

Exercise 17. Use exercise 4 to show that there exists an oracle machine U that when started on empty input halts if and only if the oracle contains a computable set.

Using tilings, we can only simulate machines with a single tape. So how can we get rid of the work tape and obtain a single tape Turing machine satisfying the conditions of the exercise? We use a single tape both to carry the information of the oracle and to perform some computation. For this we expanding the tape alphabet Γ to $\{0, 1\} \times \Gamma$. We make sure that the new machine never changes the first part of an element in the cell.

Exercise 18. Show that there exists a tile set T and a $t \in T$ for which a tiling exists, but for which every such tiling f is a non-computable function.

5 Gödel's incompleteness theorem

In this section we argue that for every "proof language" there is a true arithmetical formula that has no proof.

Arithmetical formulas are formulas that contain variables, have equality relations in them, contain the constants 0 and 1, contain operations for addition + and multiplication (written by concatenation), contain the logical connectives $\wedge$ "and", $\vee$ "or", $\neg$ "not", $\Rightarrow$, and the quantifiers $\forall$ "for all" and $\exists$ "there exists".[6] We say that a formula is *true* if it is true when variables take values over $\mathbb{Z}_{\geq 0} = \{0, 1, 2, \dots\}$, and the operations and logical connectives have their standard meaning. Otherwise, the formula is *false*. For example, $\forall c(F(c))$ is true if the expressions $F(0), F(1), \dots$ are all true.

Some examples of arithmetical formula: $0 = 1$ and $\exists a(aa = 2)$. Both formulas are false. The formula $\forall a \forall b\, (aa - bb = (a + b)(a - b))$ is true. In the construction of bigger formula we

[6] Formally, arithmetical formulas are defined as the first order language that contains one binary predicate symbol (equality), two constants (0 and 1), and two binary function symbols (addition and multiplication).

also consider expressions with free variables: the formulas $a = b \Rightarrow aa = bb$, $\exists c(a + c = b)$ and $\forall c(\neg(a = b + 1 + c))$ have two such variables a and b. The first formula is true for all values of a and b. The following formula has no free variables and states that every even natural number that exceeds 2 is the sum of two primes:

$$\forall k \exists p \exists q \Big(2(k+2) = p + q \wedge \forall r \forall s \big((rs = p \vee rs = q) \Rightarrow (r = 1 \vee s = 1) \big) \Big)$$

This is Goldbach's conjecture. Whether this formula is true is probably the most famous open question in mathematics.

Proposition 22. *The set of true arithmetical formulas is undecidable.*

Definition 23. *A relation $R \subseteq \mathbb{Z}_{\geq 0} \times \mathbb{Z}_{\geq 0}$ is* arithmetical *if there exists an arithmetical formula $F(a,b)$ with 2 free variables a and b such that $F(a,b)$ is true if and only if $(a,b) \in R$. In a similar way, relations in $R \subseteq \mathbb{Z}^k_{\geq 0} \times \mathbb{Z}^k_{\geq 0}$ can be defined using formulas with $2k$ free variables.*

Definition 24. *The* transitive closure *of a relation $E \subseteq V \times V$ is given by the relation*

$$\{(v,w) : \text{ there exists a path from } v \text{ to } w \text{ in the graph } (V,E)\}.$$

Lemma 25. *If a relation is arithmetical then its transitive closure is arithmetical.*

Proof. We first show the lemma for a relation on $\mathbb{Z}_{\geq 0}$. Let F be an arithmetical formula with two free variables. For each $a, c \in \mathbb{Z}_{\geq 0}$ we need to find a formula with the meaning:

"There exists a list $\ell_1, \ldots, \ell_n$ with $\ell_1 = a$ and $\ell_n = b$
such that $F(\ell_i, \ell_{i+1})$ holds for all $i = 1, \ldots, n-1$."

Every arithmetical formula has a fixed number of quantifiers, while the list $\ell_1, \ldots, \ell_n$ can have any length. Hence, it is not possible to translate this statement directly. Our approach is inspired by how we write numbers in some base b. Given a large number L written in base b, the ith digit from the right equals $\lfloor L/b^{i-1} \rfloor \bmod b$, where $\lfloor r \rfloor$ is the largest integer that does not exceed $r \in \mathbb{R}$. From now on we only use integer devision, and omit $\lfloor \cdot \rfloor$. We show that the statement above is equivalent to

$$\begin{aligned} \exists L, b, \overline{B} \Big[& \overline{B} \neq 0 \wedge b \neq 0 \wedge \text{"all dividers of } \overline{B} \text{ are powers of } b\text{"} \\ & \wedge L \bmod b = a \\ & \wedge L/\overline{B} = c \\ & \wedge \forall B \ \Big(Bb | \overline{B} \Rightarrow F\Big((L/B) \bmod b, (L/bB) \bmod b \Big) \Big) \Big]. \end{aligned}$$

Afterwards, we write this as an arithmetical formula.

Assume the above "formula" is true, then the list $\ell_1 = L \bmod b$, $\ell_2 = (L/b) \bmod b$, $\ldots$, $\ell_n = (L/\overline{B}) \bmod b$ satisfies all properties of the statement. We show that the statement implies the "formula". Let $\ell_1, \ldots, \ell_n$ be the values that make the statement true. We choose b to be a prime number that exceeds all these values. Let $\overline{B} = b^{n-1}$ and let $L = \sum_1^n \ell_i b^{i-1}$. The divisors of $\overline{B}$ are exactly b^i for $i = 0, \ldots, n-1$, thus the values for $((L/B) \bmod b, (L/bB) \bmod b)$ for the values B such that $Bb|\overline{B}$ are precisely (ℓ_i, ℓ_{i+1}) for $i = 1, \ldots, n-1$. By assumption F is true for all these pairs.

The "formula" can be rewritten as an arithmetical formula by using the equivalences

$$\begin{aligned} a \neq b &\Leftrightarrow \neg(a = b) \\ a < b &\Leftrightarrow \forall k[a \neq b + k], \end{aligned}$$

and for any expression E we have

$$\begin{array}{ll} E(a \bmod b) & \Leftrightarrow \forall u \forall r((a = ub + r \wedge r < b) \Rightarrow E(r)) \\ E(a/b) & \Leftrightarrow \forall u \forall r((a = ub + r \wedge r < b) \Rightarrow E(u)) \\ \forall D(D|\overline{D} \Rightarrow E(D)) & \Leftrightarrow \forall D \forall q(qD = \overline{D} \Rightarrow E(D)). \end{array}$$

Finally, the statement "the only dividers of M are powers of b" is equivalent to:

$$M \neq 0 \ \wedge \ b \neq 0 \ \wedge \ \forall u, v\Big((M = uv \wedge v \neq 1) \Rightarrow M = ub(v/b)\Big).$$

For relations over $(\mathbb{Z}_{\geq 0})^k$ the formula is similar: it starts with existential quantifiers over $\overline{B}$, b, and over k variables $L_1, \ldots, L_k$. Every divisor B of $\overline{B}/b$ defines a k-tuple $((L_1/B) \bmod b, \ldots, (L_k/B) \bmod b)$. It is easy to adapt the formula above. □

Proof of Proposition 22. We reduce H_ε to the set of true arithmetical formula. We describe the state a Turing machine by 3 natural numbers, and show that there is an arithmetical relation to describe subsequent states in a computation. Then we use Lemma 25 to obtain an expression that is equivalent to whether the machine halts on empty input.

Let (Q, Γ, δ, s) be a deterministic Turing machine. We code a state (i, q, T) of the machine with three numbers.

- A number that codes q and the symbol T_i of the cell of the head.
- A number that when written in base b corresponds to the part of the tape at the left of the computation head.
- A number that when written in base b in reversed form corresponds to the right part of the tape.

Associate the elements of Γ with $0, 1, \ldots, b-1$, where b is the number of elements in Γ. For $a \in \Gamma$, we write this encodig as $\langle a \rangle$. We choose $\langle \sqcup \rangle = 0$. Using this mapping, we can interprete $x = x_1 \ldots x_n \in \Gamma^*$ as a number in base b. We also associate each $q \in Q$ with a number $\langle q \rangle$, and we associate a pair (q, a) with $\langle a \rangle + b\langle q \rangle$. A state (i, q, T) is described by

$$h = \langle T_i \rangle + b\langle q \rangle, \qquad \ell = \sum_{j=0}^{n-1} \langle T_{i-1-j} \rangle b^j, \qquad r = \sum_{j=0}^{\infty} \langle T_{i+1+j} \rangle b^j.$$

The coding of the tapes is illustrated in figure 4 for a machine with tape alphabet $\Gamma = \{\sqcup, \mathtt{1}\}$. The encoding is $\langle \sqcup \rangle = 0$ and $\langle \mathtt{1} \rangle = 1$.

Let us see how these number change when a machine performs a computation step with a right move. Assume that the current cell contains a, the machine writes a' and goes to a state q':

$$\begin{array}{ccl} r & \to & r/b, \\ \ell & \to & b\ell + a', \\ h' & \to & (r \bmod b) + b\langle q' \rangle. \end{array}$$

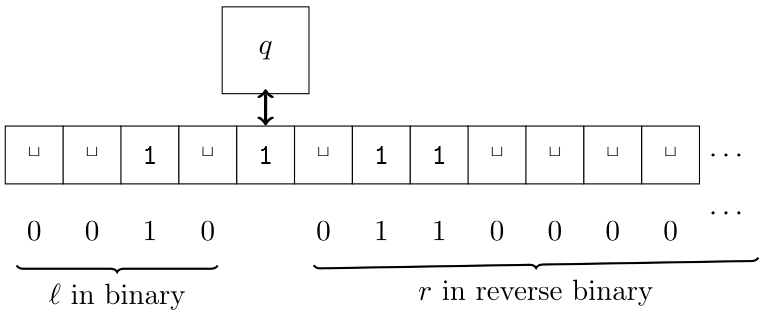


Figure 4: Example for a machine with $\Gamma = \{\sqcup, 1\}$. We have $\ell = 2, r = 6$.

Similar for a left move. Now it is easy to see how we can write the relation that checks whether two triples (h, ℓ, r) and (h', ℓ', r') code subsequent states of a computation step:

$$S(h, \ell, r; h', \ell', r') = \bigvee_{\substack{(q,a)\in \operatorname{Dom}\delta \\ \delta(q,a)=(a',R,q')}} (r' = r/2) \wedge (\ell' = b\ell + \langle a' \rangle) \wedge (h' = b\langle q' \rangle + r \bmod b)$$

$$\bigvee_{\substack{(q,a)\in \operatorname{Dom}\delta \\ \delta(q,a)=(a',L,q')}} (r' = br + \langle a' \rangle) \wedge (\ell' = \ell/b) \wedge (h' = b\langle q' \rangle + r \bmod b).$$

Now Lemma 25 gives us a relation T that checks whether any finite number of computation steps can transform a state described by (h, ℓ, r) to a state (h', ℓ', r'). Let $h_1, \ldots, h_k$ be the set of values that encode pairs (q, a) for which $\delta(q, a)$ is undefined. The condition that a Turing machine halts, is equivalent to

$$\exists h \exists \ell \exists r \big[T(0, 0, 0; h, \ell, r) \wedge (h = h_1 \vee \cdots \vee h = h_k) \big]$$

This formula can be computed given a description of the Turing machine. Hence, H_ε reduces to the set of true arithmetical formula. □

Gödel's theorem states that for any correct proof system there is a statement which can not be proven. For this we must explain what we mean by a proof system. A proof system is a computable language over an alphabet, whose elements we call *proofs*, together with a computable binary function $f(x, y)$, which expresses whether a proof x *proves a formula* y. Note that the set of formulas that have a proof, is recognizable.

Corollary 26. *Every proof system either proves a false formula or has a true formula without proof.*

Proof. Suppose there is a correct proof system that can prove all true statements. For each formula w, either w or its negation $\mathrm{NOT}(w)$ is true (and not both). Hence, either w or $\mathrm{NOT}(w)$ is provable. This implies that both the sets of true formula and false formula are recognizable. The third item of exercise 9 implies that the set of true formulas is decidable, and this contradicts Proposition 22. Hence, the assumption must be false: a correct proof system can not prove all true formulas. □

We can prove the corollary in a more direct way. Consider the following statement:

$$\boxed{\text{The statement in the box has no proof.}}$$

Suppose the statement has a proof. Then the proof system proves a false theorem. Otherwise, it is true, and hence unprovable by assumption. In both cases it satisfies the conditions of Gödel's theorem.

How can we express the statement above using an arithmetical formula? Consider a Turing machine U that takes a binary encoding $\langle\alpha\rangle$ of an arithmetical formula α as input and searches for a proof. Upon finding such a proof it halts, otherwise it runs forever. From the proof of Proposition 22 we obtain an arithmetical statement that is equivalent to "U halts on input $\langle\alpha\rangle$". This statement is true if and only if "α has a proof".

Now, we must find a way to let a formula refer to itself. We can compute a list of all arithmetical formulas that contain one free variable. Let V be a Turing machine that on input n computes the nth such formula, then replaces the variable with the value n, and finally it simulates U with this formula as input. $V(n)$ halts if and only if

"After substituting the value n for the free variable in the nth formula,
we obtain a provable formula."

With the construction above we obtain an equivalent arithmetical formula $\alpha(n)$. Consider its negation:

$$\text{NOT } \alpha(n).$$

Clearly, this formula has one parameter, and it appears in the list. Let N be its index. The formula

$$\text{NOT } \alpha(N)$$

has no parameters. It means "After substituting the value N for the free variable in the Nth formula, we obtain an unprovable formula." This Nth formula after substitution is NOT $\alpha(N)$, thus the formula refers to itself. Hence, it is either unprovable or false. This concludes the (rather informal) proof.