Turing complete problems

In this section we argue that there exist problems for which no algorithm exists that correctly solves all instances. The goal of this section is to train to recognize such problems. If one needs to solve an intrinsically hard problem, one needs to reflect about the desired requirements. Maybe, by imposing some additional assumptions on the data, one can obtain a problem for which both an algorithm provably works and might also be useful in practice...

Recall that we are still studying computability theory, which means that we do not worry about time and space efficiency of our programs. We just wonder whether there exists an algorithm that solves the problem and terminates. At the same time, several techniques such as diagonalization and the use of Turing reductions which we learn here, are also important in computational complexity theory.

1 Diagonalization and the Halting problem

Previously, we argued that many functions are Turing computable. We now argue that there exist functions that are not computable. We use the technique of diagonalization to show that some object does not belong to some set. The main idea is illustrated by a funny story for children (also used in in bars late in the night). Imagine that there is a village. In the village works a barber for which the following rule holds: *The barber razes everyone that lives in the village and that does not raze himself.* Does the barber live in the village? To answer this question, assume that the barber also lives in the village. Does he raze himself? If he razes himself, he should not raise himself, and if he doesn't raise himself, he should raise himself. We get a contradiction, and our assumption must be wrong: the barber can not live in the village.

Diagonalization can be used to show that the real numbers are not "countable". Let us first clarify 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 sets (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 $1, 2, 3, 4, \ldots$ Many other sets are countable:

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

Figure 1: Countable sets

Verify that every string you can think of, will appear in the list above. Is it possible to make such a list for the real numbers?

Exercise 1. If A and B are countable, give lists that contain all elements of $A \cup B$, $A \times B$, A^k for all k and A^*. These sets are all countable.

Subsets of countable sets are also countable. All sets whose elements can be encoded as bitstrings, are countable because the set of bitstrings are countable. For example, there are countably many finite state machines and Turing machines. 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.*

1	**1**0111...
2	0**1**001...
3	01**1**10...
4	000**0**1...
5	0110**0**...
inverse diagonal	**00011**...

Figure 2: The method of diagonalization

Proof. Suppose there exists a list that contains all binary sequences. Consider the sequence containing the inverse of the 1st bit of the 1st sequence, the inverse of the 2nd bit of the 2nd real, etc., see figure 2. Because the table contains all sequences, the constructed sequence must appear in it, say in the ith row. By construction, the ith bit must equal its inverse, which is impossible. □

With a similar argument we can show that also the set of real numbers in the interval $[0, 1]$ is not countable: Suppose a list of all real numbers exists. We convert this list into a list of all binary representations of real numbers. Note that some real numbers have two binary representations, for example: $0.0111\cdots = 0.1000\ldots$, so we insert both representations in the list. In the same way as in the proof above we construct a sequence that does not appear in the list to obtain a contradiction.

Corollary 2. *There exists a binary function that is not computable.*

Proof. The set of Turing machines is countable, because Turing machines can be encoded as strings and the set of strings is countable. Hence, also the set of computable binary sequences is countable. Therefore, they must be a strict subset of the binary sequences. □

In fact, this argument shows that most binary functions are not computable: if we define a binary function f by choosing the bits $f(1), f(2), f(3), \ldots$ from independent fair coin flips, the function f is not computable with probability one.

Theorem 3. *If for a binary function f, the bits $f(1)$, $f(2)$, ... are chosen independently and uniformly randomly, then f is not computable with probability one.*

Proof. Because there are only countably many Turing machines, also the list of computable functions is countable. Let $f_1, f_2, \ldots$ be a list of computable binary functions and choose any $\ell \in \mathbb{N}$. We show that f is not computable with probability $2^{-\ell}$. 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 $\ell+2, \ldots, 2\ell+3$. With probability at most $2^{-\ell-3}$, f equals f_3 on the next segment of length $\ell+3$, and so on. By the union bound, f equals one of the functions $f_1, f_2, \ldots$ with probability $2^{-\ell-1} + 2^{-\ell-2} + \cdots = 2^{-\ell}$. This holds for any ℓ, hence, with probability one, f is not computable. □

Exercise 2. Recall that computable functions are total. Let $\varphi : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ be a function such that for every computable function $f : \mathbb{N} \to \mathbb{N}$ there exists an m such that $f = \varphi(m, \cdot)$. Show that φ is not computable.

Imagine there exists a programming language and a code checker. This checker inspects some code and verifies whether it implements a unary function in this language and that this program returns a value for each argument. Imagine this verifier is computable and correct for all possible codes in this programming language. Argue that there exists a computable function that can not be implemented in this language (but evaluating this function might go unrealistically slow). (Hint: apply the checker to all possible codes to generate a list containing all unary functions it can produce. By assumption all these functions are total.)

We now show that there exist a more natural set that is not decidable. Using a similar proof technique as in Theorem 1, we prove that the halting set is not decidable. Let φ be a universal function (See Theorem 9 of the previous notes). Recall that a universal function defines a list of all computable one argument functions φ_n. The *halting set* is the set of all pairs $(n, m) \in \mathbb{N} \times \mathbb{N}$ for which $\varphi_n(m)$ is defined.

Theorem 4. *Let φ be a universal numbering. The Halting problem on φ is not decidable, i.e., the function*

$$h(n,m) = \begin{cases} 1 & \text{if } \varphi_n(m) \text{ halts} \\ 0 & \text{otherwise} \end{cases}$$

is not computable.

Proof. [1] For every computable function f, there exists a row in the table of h that represents the domain of f. The inverse diagonal of the table represents the domain of the function

$$g(n) = \begin{cases} 1 & \text{if } \varphi_n(n) \text{ is undefined, i.e., if } h(n,n) = 0 \\ \text{undefined} & \text{otherwise, i.e., if } h(n,n) = 1. \end{cases}$$

If h were computable, then g would be partial computable and there exists a row in the table that corresponds to the domain of g. Let this row be the mth row. By construction $g(m)$ *is defined if and only if* $h(m,m) = 0$. But $h(m, \cdot)$ represents g's domain: $g(m)$ *is defined if and only if* $h(m,m) = 1$. Both statements contradict each other, hence, h can not be computable. □

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

In fact, in the proof above, we obtain a contradiction if we assume that the set $\{n : \phi_n(n)$ halts$\}$ is decidable. Hence, already this set is undecidable.

Exercise 3. An extension of a partial function g is a partial function which equals g on all arguments for which g is defined. Modify the proof of Theorem 4 to show that there exists a partial computable binary function that has no computable extension. (Hint: choose $g(n) = 1 - \varphi_n(n)$ if $\varphi_n(n) \in \{0, 1\}$ and undefined otherwise.)

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 does not belong to the set. Such sets are called enumerable.

Definition 5. *A set is* enumerable *if there exists a partial computable function whose domain equals the set.*

Obviously, the halting set is enumerable. If a set of strings is enumerable, there exists a machine that never terminates and outputs on a separate output tape a sequence of strings, separated by blanks, containing all elements of the set. Such machines are called *enumerators*. Vice versa, if a set has such an enumerator, then it is enumerable. In the following exercise some more equivalences are given.

Exercise 4. show that the following are equivalent for a set $A \subseteq \{0, 1\}^*$:

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

Also computable sets can be characterized by images of computable functions.

Exercise 5. Show that a set of natural numbers is decidable if and only if it is the image of an increasing function.

Exercise 6. Show that:

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

2 Turing reduction

In the previous paragraph we used diagonalization to prove that the halting set is not decidable. We now prove the same for many other sets. It is rather time consuming to repeat a similar diagonalization argument for each such set. A faster method is to show that these problems are "at least as hard" than the halting problem. By this, we mean that if an algorithm for the problem exists, this algorithm could be transformed to an algorithm that decides the halting problem. Hence, if the original problem where decidable, then the halting problem would be decidable too, which is impossible.

2.1 Informal examples

Let us illustrate the relation "at least as hard as" informally. Because Turing machines are rather hard to code, we consider real code in any popular programming language. The set *Halt* contains all pairs (p, x) for which p represents some code for a one argument function such that the code terminates for the argument x. Consider now the set *HaltSomewhere* of programs p for which there exists some x such that p halts on input x. We claim that *HaltSomewhere is at least as hard to decide as Halt*, in other words, *HaltSomewhere* ***computes*** *Halt*.

Given (p, x) and the set *HaltSomehwere*, how can we know whether $(p, x) \in$ *Halt*? We might ask whether $p \in$ *HaltSomewhere*, if not, we know that p does not halt on input x. Otherwise, we can not conclude anything. A better approach is to transform the code p and hard code the argument x in it. Suppose p represents some Python code that implements a one argument function:

```
def foo(arg):
    'some code'
```

This code is transformed to

```
def foo(arg):
    arg = x
    'some code'
```

Let us denote this code as $q_{p,x}$. If the original code does not terminate on input x, then the transformed code will not terminate on any input. If the original code does terminate on input x, then the transformed code will terminate on any input.[2] Hence, $(p, x) \in$ *Halt* if and only if $q_{p,x} \in$ *HaltSomewhere*. So *HaltSomewhere* computes *Halt*. This implies that also *HaltSomewhere* is not decidable.

The reverse also holds: *Halt computes HaltSomewhere*. We want to know whether some program p halts on some input and we can use the set *Halt*. Let us assume the program p takes natural numbers as input. We consider the program that ignores its argument and tries p on all possible inputs in parallel and terminates as soon as it finds one input on which p terminates. This can be done in the following way:

[2] In Python the above construction will not work in some rather esoteric cases. For example if one accesses line numbers of the source code, or input variables that are stored through the function frame object. In these cases, the transformation is a bit more complicated but still possible.

evaluate p on input	1	for	1	computation step(s)
	1		2	
	2		2	
	1		3	
	2		3	
	3		3	
	1		4	

...

Terminate if one of the evaluation processes terminates.

Let q_p be this transformed program. $p \in \mathit{HaltSomewhere}$ if and only if $(q_p, 1) \in \mathit{Halt}$. Hence, *Halt* computes *HaltSomewhere*.

Exercise 7. Let *HaltOnOne* be the set of programs that halt on input 1. Argue that the set *HaltOnOne* computes *Halt* and vice versa.

2.2 Definition

To define Turing reducibility, we need to represent a possibly infinite set $A \subseteq \mathbb{N}$ on a tape of a Turing machine. This is straightforward: on the nth cell we place a 1 if $n \in A$ and zero otherwise. For any other type of elements in a countable set, e.g. strings or pairs of natural numbers we represent set membership in the same way using some natural order of elements (for example the order from table 1).

A Turing machine with an *oracle* for a set A is a multitape Turing machine for which at the beginning of the computation, one of the tapes represents A.

Definition 6. *We say that a set A* (Turing) computes *a set B, (or B Turing reduces to A) notation $B \leq_T A$, if there exists an oracle Turing machine U with states* `accept` *and* `reject` *such that $U^A(x)$ terminates in the state* `accept` *if $x \in B$ and terminates in the state* `reject` *if $x \notin B$.*

Exercise 8. Let $A, B, C \subset \mathbb{N}$. Prove that:

- If $A \leq_T B$, then $A \leq_T \mathbb{N} \setminus B$, $\mathbb{N} \setminus A \leq_T B$ and $\mathbb{N} \setminus A \leq_T \mathbb{N} \setminus B$.
- If $A \leq_T B$ and $B \leq_T C$, then $A \leq_T C$.
- If B is decidable and $A \leq_T B$, then A is decidable.
- Let f be a computable function such that $x \in A$ if and only if $f(x) \in B$. Show that $A \leq_T B$.
- Question: If B is enumerable and $A \leq_T B$, can we conclude that A is enumerable? Prove the statement or give a counter example.

2.3 Examples of reductions and Rice's theorem

We illustrate the definition of a Turing reduction. We start by repeating the first example from subsection 2.1. φ is a *Gödel numbering* if for every partial computable two argument function ψ, there exists a computable function w such that $\varphi_n = \psi_{w(n)}$ for all n. An optimal Gödel numbering is a Gödel numbering, and such numberings exist by Theorem 10 of the previous chapter. Let φ be a Gödel numbering.

$$\begin{aligned} \mathit{Halt} &= \{(n,m) : \varphi_n(m) \text{ is defined}\} \\ \mathit{HaltSomewhere} &= \{n : \exists m\, [\varphi_n(m) \text{ is defined}]\}\,. \end{aligned}$$

We show that *HaltSomewhere* computes *Halt*. (Hence, *HaltSomewhere* is undecidable.) Let $\langle\cdot,\cdot\rangle$ denote the function that maps pairs of natural numbers to natural numbers defined in table 1. Clearly, this function is computable. To decide whether $(n,m) \in \mathit{Halt}$, we consider a numbering ψ defining a series of constant functions:

$$\psi_{\langle n,m\rangle}(k) = \begin{cases} 1 & \text{if } \varphi_n(m) \text{ is defined} \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Note that ψ is a partial computable 2-argument function. Because φ is a Gödel numbering, there exists a computable function w such that

$$\varphi_{w(\langle n,m\rangle)} = \psi_{\langle n,m\rangle}.$$

(In the informal discussion $w(\langle n,m\rangle)$ corresponds to $q_{n,m}$, i.e., the program n with hard coded argument m.) Hence, $(n,m) \in \mathit{Halt}$ if and only if $w(\langle n,m\rangle) \in \mathit{HaltSomewhere}$. Hence, *HaltSomewhere* computes *Halt*. Indeed, in Definition 6 one can choose $U^A(\ell)$ to be a machine that first computes $w(\ell)$, then accepts if $w(\ell) \in A$ or rejects otherwise.

Consider the following problem: given a Turing machine and a deterministic finite automaton, decide whether the Turing machine halts precisely on the strings that are accepted by the automaton. To formalize this question consider a list of all automata and let R_1, R_2, R_3 the corresponding sets of accepted strings. The list should be such that the set $\{(i,x) : x \in R_i\}$ is decidable. If φ is the Gödel function constructed in the proof of the existence of optimal Gödel functions, (Theorem 10 of the previous chapter), then $\varphi_1, \varphi_2, \varphi_3, \ldots$ are the functions computed by a list of all Turing machines. The deciding whether the domain of some Turing machine equals the language accepted by a finite state machine is equivalent to deciding

$$\mathit{HaltOnReg} = \{(n,r) : \operatorname{Dom}\varphi_n = R_r\}\,.$$

Does *HaltOnReg* compute *Halt*? The empty set is a regular set. Hence, *HaltSomewhere* is a subproblem of *HaltOnReg*. More formally, if r_0 is an index of the empty set in the list of regular sets, then $n \in \mathit{HaltSomewhere}$ if and only if $(n,r_0) \notin \mathit{HaltOnReg}$. Hence, *HaltOnReg* computes *HaltSomewhere*. Therefore *HaltOnReg* also computes *Halt* by the second property of exercise 8.

Let us now consider a closely related problem. Is the set of strings on which a Turing machine halts, a regular set? Let

$$\mathit{DomIsReg} = \{n : \operatorname{Dom}\varphi_n \text{ is regular}\}\,.$$

We show that *DomIsReg* computes $HaltOnOne = \{n : \varphi_n(1) \text{ halts}\}$, hence, by exercise 7, the set is undecidable. Obviously, the empty set is decidable. Let NR be a computable non-regular set (for example the set of all strings 0^i1^i). Let ψ be a partial computable function such that

$$\operatorname{Dom}\psi_n = \begin{cases} NR & \text{if } \varphi_n(1) \text{ is defined} \\ \emptyset & \text{otherwise.} \end{cases}$$

For example, $\psi_n(x)$ equals 1 if $\varphi_n(1)$ halts and $x \in NR$, and is undefined otherwise. Let w be such that $\varphi_{w(n)} = \psi_n$ for all n, then

$$w(n) \in DomIsReg \Leftrightarrow n \in HaltOnOne.$$

Hence, *HaltOnOne* computes *DomIsReg*.

All previous examples somehow state that given a program, nothing "can be said" about the output of the program. For any nontrivial property of a one argument partial function, one can not decide whether a program computes a function with this property. For some programs, it is easy to see whether their output satisfies the property, but their always exist programs for which it is very hard to understand whether the property holds. With a *trivial* property, we mean that it either holds for all partial computable functions or for none.

Theorem 7. *If φ is a Gödel numbering and F is a set of functions, then*

$$I = \{n : \varphi_n \in F\}$$

is either empty, equals $\mathbb{N}$, or is undecidable.

Proof. First assume that the everywhere undefined function does not belong to F. If I is empty nothing needs to be proven. Assume I is non-empty and let k be such that $\varphi_k \in F$. Let

$$\psi_n = \begin{cases} \varphi_k & \text{if } \varphi_n(1) \text{ is defined} \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Note that ψ is a partial computable function. Let w be the function such that $\varphi_{w(n)} = \psi_n$ for all n. Hence,

$$w(n) \in I \Leftrightarrow n \in HaltOnOne$$

Hence, I computes *HaltOnOne*, and this implies that I is undecidable.

The case where the everywhere undefined function does belong to F is proven in the same way by choosing k such that $\varphi_k \notin F$. This is possible if $I \neq \mathbb{N}$. □

In fact, this proof implies a stronger result: the set I is either empty, equals $\mathbb{N}$ or decides *Halt*. Observe that the statement of the theorem is symmetric after replacing F by its complement, i.e., by letting I be the set of n such that $\phi_n \notin F$. One can also give a proof where this symmetry is better visible.

Second proof. In exercise 3 it is shown that there exists a partial computable binary function g without computable extension. We show that if I was nontrivial and decidable, then we can

construct a computable extension $\tilde{g}$. Let φ_{in} and φ_{out} be two partial computable functions in and out F. Unless $I = \emptyset$ or $I = \mathbb{N}$, these functions exist. Let

$$\psi_n = \begin{cases} \varphi_{\text{in}} & \text{if } g(n) = 1 \\ \varphi_{\text{out}} & \text{if } g(n) = 0 \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Consider the following extension of g:

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

If I was decidable, then $\tilde{g}$ is a computable extension of g, which contradicts the choice of g. The theorem is proven. □

Rice's theorem implies that in every universal programming language, there are infinitely many programs that compute the same function. Indeed, every finite set is decidable, and for some program n we can apply Rice's theorem with $F = \{\varphi_n\}$.

2.4 Turing complete sets

Informally, the Turing complete sets are the "most difficult" enumerable sets to decide.

Definition 8. *A set is* Turing complete *if it is enumerable and it computes any other enumerable set.*

The Halting set *Halt* is Turing complete. Indeed, if S is enumerable, it is the domain of a partial computable function. Therefore, it equals $\operatorname{Dom} \varphi_n$ for some n, and deciding this set is a subproblem of deciding *Halt*.

Exercise 9. Prove the statement or provide a counter example.

- If A is Turing complete and $A \leq_T B$ for some enumerable B, is B Turing complete?
- If A and B are Turing complete, is $A \cup B$ Turing complete?

Exercise 10. Let φ be a Gödel numbering. Show that the sets $\{n : \varphi_n(1) \text{ halts}\}$ and $\{n : \varphi_n(n) \text{ halts}\}$ are Turing complete.

All the enumerable sets we have seen so far are either computable or Turing 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. Remarkably, it has been shown that there exist enumerable sets that are neither decidable nor Turing complete (the Friedberg-Muchnik theorem). In fact, there is an infinite chain of enumerable sets such that $A_1 <_T A_2 <_T \dots$ where $A <_T B$ means that $A \leq_T B$ and $B \not\leq_T A$.

3 Beyond Turing completeness

The goal of this section is to show that there exist natural problems that are even harder to solve than the Halting problem.

First we start with some philosophical note. Imagine at some point in the future, humans construct a new device that is able to decide a set which is undecidable using Turing machines. From this point on, computers are equipped with such a device. The definition of a Turing machine becomes obsolete and is replaced by a more powerful concept. What should be changed in our theory of computation? Is Rice's theorem still valid for this new type of machine? What about other theorems or properties we've met in the exercises? It turns out that basically nothing will change. Why? We can model the communication with the device by a binary function and represent this as an oracle on a Turing machine. All our proofs that use ordinary Turing machines, also work for machines that have an oracle.

3.1 The jump operator

With the same arguments as in the proof of Theorem 10 of the previous chapter, it can be shown that there exist oracle Turing machines that define optimal Gödel numberings relative to any oracle, i.e., we can construct an oracle Turing machine that simulates any other oracle machine by prefixing it with some fixed instructions. We fix such a machine and denote it as φ^A when used with an oracle A. The halting problem on such a machine is denoted as A'.

$$A' = \{(n, m) : \varphi_n^A(m) \text{ is defined}\}.$$

With the same arguments as above, we conclude that A' is not decidable on a machine with oracle A, i.e., $A' \not\leq_T A$. However, from A' we can recover A.

Lemma 9. $A \leq_T A'$

Proof. Let

$$\psi_n^A = \begin{cases} 1 & \text{if } n \in A \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Clearly ψ^A is partial computable with oracle A. Let w be the function for which $\varphi_{w(n)}^A = \psi_n^A$ for all n. Then,

$$(w(n), 1) \in A' \Leftrightarrow n \in A$$ □

Exercise 11. Show that if B is enumerable on a machine with oracle A, then $B \leq_T A'$.

The mapping A to A' is a mapping from sets to sets, and this can be iterated: A'' is the halting problem on a machine with oracle A'. We denote the empty set as $\mathbf{0}$. The halting problem is then written as $\mathbf{0}'$. The halting problem relative to the halting problem is $\mathbf{0}''$. In the subsequent section we will show that deciding whether a Turing machine halts on every input, is Turing equivalent to $\mathbf{0}''$.

Exercise 12. Show that

$$\{n : \varphi_n \text{ is total}\} \leq_T \mathbf{0}''$$
$$\{n : \varphi_n \text{ has finite domain}\} \leq_T \mathbf{0}''.$$

For both sets, also the $\geq_T$ inequality holds. This will be proven in the next subsection.

3.2 The arithmetical hierarchy

In computability theory, the arithmetical hierarchy is a very convenient technical tool to prove that some problem is even harder than the halting problem. There exists a polynomial variant of the arithmetical hierarchy and we will study this too, in the next part about computational complexity.

A *predicate* is a function with the logical values `True` or `False`. A Σ_1-formula is a formula of the type

$$\exists s \in \mathbb{N} : R(s, n).$$

for some computable predicate R. With this formula we associate the set of all n for which the expression is true. The class Σ_1 is the class of sets associated to Σ_1-formulas. The set defined by $\exists s \exists t : R(s, t, n)$ is equivalent to a Σ_1-formula: using a computable pairing function one can replace the variables s and t by a single variable. In exercise 4 it is shown that Σ_1 coincides with the class of enumerable sets. (Checking whether a Turing machine U on input n halts within t steps can be done by a computable function.)

The class Π_1 contains the sets characterized by Π_1-formulas, i.e., formulas of the type

$$\forall s \in \mathbb{N} : R(s, n)$$

A set is Π_1 if and only if its complement is Σ_1.

Definition 10. *The classes Σ_0 and Π_0 equal the decidable sets. For $k = 0, 1, 2, \ldots$, we say that:*

- *a set A is Σ_{k+1} if and only if there exists a Π_k-set B such that A contains the elements n for which $\exists t \in \mathbb{N} : B(t, n)$.*
- *a set A is Π_{k+1} if and only if there exists a Σ_k-set B such that A contains the elements n for which $\forall t \in \mathbb{N} : B(t, n)$.*

One can verify by induction that a set is Σ_k if its complement is Π_k. A Σ_k-*formula* is a formula that consists of k alternations of quantifiers $\exists$ and $\forall$ followed by a computable relation R:

$$\overbrace{\exists s_1 \forall s_2 \exists s_3 \ldots \overset{\forall s_k}{\exists s_k}}^{k \text{ alternations}} : R(n, s_1, s_2, s_3, \ldots, s_k).$$

Similar for Π_k-*formulas* where one starts with a $\forall$ quantifier. The sets in Σ_k and Π_k are precisely the sets that have Σ_k and Π_k-formulas. Recall that subsequent quantifiers $\ldots \exists u \exists v \exists w \ldots$ can always be contracted to a single $\ldots \exists z \ldots$ variable and similar for $\forall$; thus we could equivalently define Σ_k and Π_k-formulas by formulas that start with any number of quantifiers but that alternate at most k times between $\exists$ and $\forall$. Note that $\Sigma_k \subset \Sigma_\ell$ and $\Sigma_k \subset \Pi_\ell$ set for all $\ell > k$, by adding dummy variables to R.

Exercise 13. Σ_k and Π_k are closed under intersection and union, i.e., $A, B \in \Sigma_k$, then $A \cup B, A \cap B \in \Sigma_k$ and similar for Π_k.

Exercise 14. Verify that

1. $\{n : \operatorname{Dom} \varphi_n \neq \emptyset\}$ is Σ_1

2. $\{n : |\mathrm{Dom}\,\varphi_n| \leq 1\}$ is Π_1
3. $\{n : \mathrm{Dom}\,\varphi_n \text{ is finite}\}$ is Σ_2
4. $\{n : \varphi_n \text{ is total}\}$ is Π_2
5. $\{n : \mathrm{Dom}\,\varphi_n \text{ has finite complement}\}$ is Σ_3
6. $\{n : \mathrm{Dom}\,\varphi_n \text{ has infinite complement}\}$ is Π_3.

It is not hard to prove that if $A \in \Sigma_1$ or $A \in \Pi_1$ implies $A \leq_T \mathbf{0}'$.

Exercise 15. Use induction to show that $A \leq_T \mathbf{0^{(k)}}$ for all A in Σ_k and Π_k.

The next theorem shows that $\mathbf{0^{(k)}} \in \Sigma_k$ and hence, $\mathbf{0^{(k)}}$ is "Turing complete" in Σ_k.

Theorem 11. $\mathbf{0^{(k)}} \in \Sigma_k$ *for all* k.

To better understand the idea, we give a direct proof of the theorem for a special case. For $k = 0$ and $k = 1$ the arguments are straightforward, so let us choose the case $k = 2$. Afterwards we will repeat the proof with more detailed explanations and also prove the general case.

Direct proof of Theorem 11 for $k = 2$. Recall that any set $A \subset \mathbb{N}$ can be represented as an infinite sequence. We write $x \sqsubset A$ if x is an initial segment of this sequence. We need to show that $\mathbf{0''} \in \Sigma_2$. The statement: "$x \sqsubset \mathbf{0'}$" is equivalent to

$$(\exists t \forall i \leq |x|\,[x_i = 1 \Rightarrow U(i) \text{ halts in less than } t \text{ steps}]) \text{ AND}$$
$$(\forall s \forall i \leq |x|\,[x_i = 0 \Rightarrow U(i) \text{ does not halt in } s \text{ steps}])\,.$$

For some computable predicates R and S, this is of the form $\exists t[R(x,t)]$ AND $\forall s[S(x,s)]$, because the quantifier $\forall i \leq |x|$ can be included in the programs for R and S. Let $U^x(n)$ be the result of the computation of an oracle machine on input n if the string x is written on the beginning of the oracle tape and if during the computation only bits of x are accessed. If this computation does not halt, or other bits of the oracle are accessed, then $U^x(n)$ is undefined.

$n \in \mathbf{0''}$ if and only if $U^{\mathbf{0'}}(n)$ halts, i.e., if

$$\exists x \exists t[U^x(n) \text{ halts in less than } t \text{ steps}] \text{ AND } x \sqsubset \mathbf{0'}.$$

After substituting the above expression for "$x \sqsubset \mathbf{0'}$" and rearranging the quantifiers, we obtain a Σ_2-formula for $\mathbf{0'}$. □

We now repeat this argument more carefully. The argument is broken up in several steps in the lemmas below. For any set A, the class Σ_1^A is given by all sets $\{\exists t : R(n,t)\}$ where R is a predicate that is computable with oracle A. One easily verifies that $A' \in \Sigma_1^A$.

Lemma 12. *If* $A \in \Sigma_1^B$ *then* $\{x : x \sqsubset A\} = B \setminus C$ *for some* $B, C \in \Sigma_1^B$.

Proof. Let S be a predicate that is computable relative to B such that

$$n \in A \quad \Leftrightarrow \quad \exists t : S(t, n)$$

Let $\tilde{S}$ be the monotone variant of S, i.e., $\tilde{S}(s, n) \Leftrightarrow \exists t \leq s : S(t, n)$. Clearly, $\tilde{S}(t, n)$ implies $\tilde{S}(t + d, n)$ for all $d \geq 0$, and $\tilde{S}$ is computable relative to B. Let

$$\begin{aligned} x \sqsubset A \qquad \Leftrightarrow \qquad & \exists t \forall i \leq |x| \left[x_i = 1 \Rightarrow \tilde{S}(t, i) \right] \\ & \text{AND NOT } \exists t \exists i \leq |x| \left[x_i = 0 \Rightarrow \tilde{S}(t, i) \right]. \end{aligned}$$

The expression above corresponds to the difference of two Σ_1^B sets, because the $\forall i \leq |x|$ and $\exists i \leq |x|$ quantifiers can be included in the B-computable predicates (that both have arguments x and t). □

Definition 13. *Let $\mathcal{C}$ be a class of sets. The class of* $\mathcal{C}$-difference sets *is the class of all sets $A \setminus B$ for A and B in $\mathcal{C}$.*

Recall that $A' \in \Sigma_1^A$. Lemma 12 implies:

Corollary 14. *$\{x : x \sqsubset A'\}$ is a Σ_1^A-difference set.*

Lemma 15. *If A is a Σ_k-difference set, then $\{n : \exists t[A(n, t)]\}$ is a Σ_{k+1} set.*

Proof. Because A is a Σ_k-difference set, we have that $A = A_1 \cap A_2$ for some $A_1 \in \Sigma_k$ and $A_2 \in \Pi_k$. The set

$$\exists t : A_1(n, t)$$

is in Σ_k, and therefore also in Σ_{k+1}. The set

$$\exists t : A_2(n, t)$$

is Σ_{k+1} by definition of Σ_{k+1}. Hence, the intersection of both sets defined above is Σ_{k+1} too, and this intersection is precisely $\{n : \exists t[A(n, t)]\}$. □

Lemma 16. *If $A \in \Sigma_1^B$ and the set $\{b : b \sqsubset B\}$ is a Σ_k-difference set, then $A \in \Sigma_{k+1}$.*

Proof. Assume that $A \in \Sigma_1^B$ and let S be the predicate that is computable with oracle B for which $n \in A$ is equivalent to $\exists t : A(n, t)$. Let $A(n, t; b)$ be the predicate that is true if the computation of the predicate A is started on input (n, t) with the string b written on the oracle returns true; assume that if in this computation the oracle is accessed at a position at the right of b, then $A(n, t; b)$ is false. The statement $n \in A$ is equivalent to

$$\exists t \exists b \left[S(x, t; b) \wedge b \sqsubset B \right].$$

Because the set of prefixes of B is a Σ_k difference set, Lemma 15 implies that $A \in \Sigma_{k+1}$. □

Corollary 17. *If A is a Σ_1^B-difference set, and the set $\{b : b \sqsubset B\}$ is a Σ_k-difference set, then A is a Σ_{k+1}-difference set.*

Proof of Theorem 11. Clearly, the empty set $\mathbf{0}$ is computable and hence a Σ_0-difference set. Suppose that $\{x : x \sqsubset \mathbf{0^{(k)}}\}$ is a Σ_k-difference set for some $k \geq 0$. We show that $\{x : x \sqsubset \mathbf{0^{(k+1)}}\}$ is a Σ_{k+1}-difference set. By Corollary 14 this set is a $\Sigma_1^{\mathbf{0^{(k)}}}$-difference set and by Corollary 17 it is Σ_{k+1}. Recall that $A' \in \Sigma_1^A$ for all sets A. The theorem follows by one additional application of Lemma 16. □

Exercise 16. Show that for every set $A \in \Sigma_2$ and every n we can define a function f_n such that $n \notin A$ if and only if f_n is total. Conclude that $\mathbf{0''} \leq_T \{n : \varphi_n \text{ is total}\}$ (and by exercise 15 $\mathbf{0''} =_T \{n : \varphi_n \text{ is total}\}$). Using a very similar argument, show that also $\mathbf{0''} =_T \{n : \varphi_n \text{ has finite domain}\}$.

4 Turing complete problems

In conversational speech, one sometimes says that a computer language or any mechanism for computation is "Turing complete" if it can simulate a Turing machine. It is often cumbersome to define "simulate" mathematically. However, if a system can simulate a Turing machine, then deciding whether the system halts when initiated in some initial configuration is hard. Hence, we are interested in how difficult it is to decide the halting problem on several systems that can compute.

4.1 Mathematical problems

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 fractions $f_1, f_2, \ldots, f_e$. 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, then the computation halts. For example, the program given by $n = 5$ and the fractions $[2/3, 6/5, 1/2]$ halts and the values for n are: 5, 6, 4, 2, 1.

Exercise 17. Show that the set of halting FRACTRAN program is Turing complete. Tip: use the goto programming language defined before. Recall that the copy command is redundand. Encode finitely many variables $a, b, c, \ldots, e$ 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 $q_\ell 2^a 3^b 5^c \ldots p^g$.

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, deciding whether latex programs terminate is Turing complete.

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 $n \in S$ if there exist $x_1, \ldots, x_e \in \mathbb{N}$ such that $P(n, x_1, \ldots, x_e) = 0$. Clearly, a Diophantine set is enumerable. In 1970, Y. Matiyasevich showed that the reverse is also true: every enumerable set is Diophantine.

This settled Hilbert's 10th problem, which asked 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 an enumerable 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 Tag systems

Another simple mechanisms capable of computing is a k-tag systems. 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.

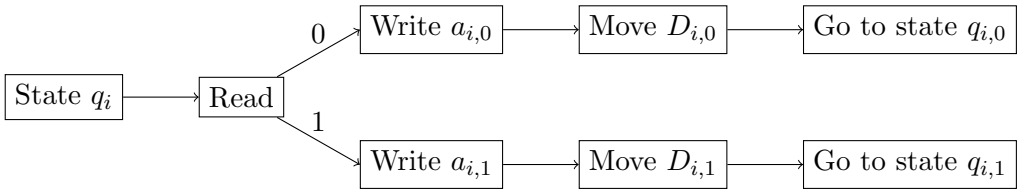
For example, consider a 2-tag system over $\{0, 1\}$ with $\delta(0) = 011$ and $\delta(1) = 1$. On input 01111 we obtain:

$$\to \sout{01}111\mathbf{011} \to \sout{11}10111\mathbf{1} \to \sout{10}1111\mathbf{1} \to \sout{11}111\mathbf{1} \to \sout{11}\mathbf{1} \to \sout{11}\mathbf{1}.$$

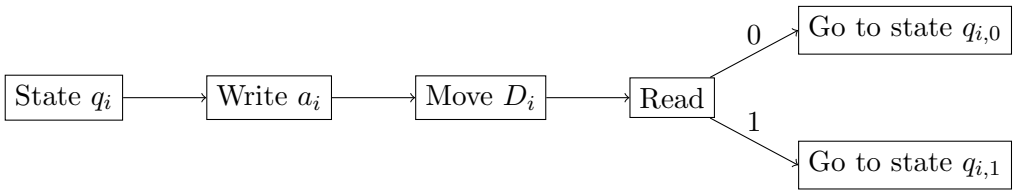
Exercise 18. Show that for every 1-tag system over A, the set of strings in A^* for which the computation halts is decidable.

On the other hand, 2-tag systems can simulate Turing machines. The remainder of this subsection gives a proof of this claim. This proof is given only as an illustration of the claim for the interested reader.

Before, we considered Turing machines that operate as follows:



Now, consider a variant of Turing machines that operate as follows:



Exercise 19. Show that by doubling the states, these machines can simulate ordinary machines. At the formal level, we run into a small problem: In the first computation step, something is written, but this means that the initial contents of the first cell can not be red. So we assume that on this type of machine, the input is preceded by an arbitrary symbol. Now the claim can be proved by induction.

We make some more assumptions on the machine:

- The head can only move left or right (and can not remain in the same position).

- The machine has a two symbol tape. As noticed below the proof of Theorem **??**, such machines can simulate any Turing machine by encoding the input using two cells for each bit. For the blank symbol we choose 00.

The idea is to represent a configuration of a Turing machine by three variables $\ell, r \in \mathbb{N}$ and $q \in Q$, as illustrated in figure 3.

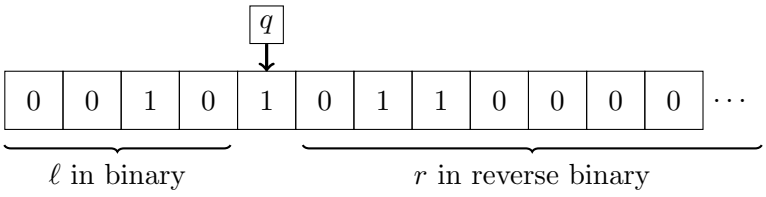


Figure 3: Example of configuration with $\ell = 2, r = 6$.

Let $\lfloor r \rfloor$ be the largest integer not above r. Let q be a state that writes b. Let q_0 and q_1 be the states that follow q after reading 0 and 1. Let (ℓ, r, q) be the state of the Turing machine just before writing. At the next computation step, also just before writing, the state is

$$\begin{aligned} \text{left:} \quad & (\lfloor \ell/2 \rfloor, 2r + b, q_{\ell \bmod 2}) \\ \text{right:} \quad & (2\ell + b, \lfloor r/2 \rfloor, q_{r \bmod 2}). \end{aligned}$$

We now describe the 2-tag system that simulates a Turing machine described above. The alphabet contains several copies of Q, more precisely:

$$Q \times \{{}_{\ominus}, {}_{\text{-}}, {}_{\oplus}, {}_{+}\} \times \{{}_{\text{-}}, {}^{\text{-}}\} \times \{{}_{\text{-}}, {}^{\cdot}_{\text{-}}, {}^{\cdot\cdot}_{\text{-}}, {}^{\smile}_{\text{-}}\}.$$

(we use sub and superscript, the notation will soon be clear). A state (ℓ, r, q), just before a bit b is written, is represented by

$$q_{\ominus}\bar{q}_{\ominus} \overbrace{q_{\text{-}}\bar{q}_{\text{-}} \cdots q_{\text{-}}\bar{q}_{\text{-}}}^{\ell \text{ times}} q_{\oplus}\bar{q}_{\oplus} \overbrace{q_{+}\bar{q}_{+} \cdots q_{+}\bar{q}_{+}}^{r \text{ times}}.$$

We explain how this string is transformed when the Turing machine moves *left* and writes b. The first transformation reflects the change of ℓ and r:

$$\begin{aligned} q_{\text{-}} &\to \dot{q}_{\text{-}} \\ q_{\ominus} &\to \dot{q}_{\ominus} \\ q_{+} &\to \dot{q}_{+}\dot{\bar{q}}_{+}\dot{q}_{+}\dot{\bar{q}}_{+} \\ q_{\oplus} &\to \begin{cases} \dot{q}_{\oplus}\dot{\bar{q}}_{\oplus} & \text{if } b = 0 \\ \dot{q}_{\oplus}\dot{\bar{q}}_{\oplus}\dot{q}_{+}\dot{\bar{q}}_{+} & \text{if } b = 1. \end{cases} \end{aligned}$$

After applying these rules, we obtain the word

$$\dot{q}_{\ominus} \overbrace{\dot{q}_{\text{-}} \cdots \dot{q}_{\text{-}}}^{\ell \text{ times}} \dot{q}_{\oplus}\dot{\bar{q}}_{\oplus} \overbrace{\dot{q}_{+}\dot{\bar{q}}_{+} \cdots \dot{q}_{+}\dot{\bar{q}}_{+}}^{2r+b \text{ times}}.$$

Now the machine reads a bit. In the left column we conseder the case where the machine reads a zero (i.e., ℓ is odd) and goes to state u. In the right column we consider the machine reads a one, (i.e., ℓ is even) and the machine goes to a state v. We apply the transformation:

$$\dot{q}_- \to \ddot{q}_-\ddot{\bar{q}}_-$$
$$\dot{q}_\ominus \to \ddot{q}_\ominus\ddot{q}_\ominus$$

$$\dot{q}_+ \to \breve{u}_+\breve{\bar{u}}_+ \qquad \dot{\bar{q}}_\oplus \to x\,\breve{v}_\oplus\breve{\bar{v}}_\oplus$$
$$\dot{q}_\oplus \to \breve{u}_\oplus\breve{\bar{u}}_\oplus \qquad \dot{\bar{q}}_+ \to \breve{v}_+\breve{\bar{v}}_+$$

After applying these rules, we obtain

$$\ddot{q}_\ominus\ddot{\bar{q}}_\ominus \overbrace{\ddot{q}_-\ddot{\bar{q}}_- \ldots \ddot{q}_-\ddot{\bar{q}}_-}^{\lfloor \ell/2 \rfloor \text{ times}} \breve{u}_\oplus\breve{\bar{u}}_\oplus \overbrace{\breve{u}_+\breve{\bar{u}}_+ \ldots \breve{u}_+\breve{\bar{u}}_+}^{2r+b \text{ times}}. \qquad \cancel{\ddot{q}}_\ominus\ddot{\bar{q}}_\ominus \overbrace{\ddot{q}_-\ddot{\bar{q}}_- \ldots \ddot{q}_-\ddot{\bar{q}}_-}^{\lfloor \ell/2 \rfloor \text{ times}} x\,\breve{v}_\oplus\breve{\bar{v}}_\oplus \overbrace{\breve{v}_+\breve{\bar{v}}_+ \ldots \breve{v}_+\breve{\bar{v}}_+}^{2r+b \text{ times}}.$$

We go a third time through the sequence, transforming the initial part of the state. For the right side, note that after the replacement of the last $\dot{\bar{q}}_+$, we have deleted the first $\ddot{q}_\ominus$.

$$\ddot{q}_\ominus \to u_\ominus\bar{u}_\ominus \qquad \ddot{\bar{q}}_\ominus \to v_\ominus\bar{v}_\ominus$$
$$\ddot{q}_- \to u_-\bar{u}_- \qquad \ddot{\bar{q}}_- \to v_-\bar{v}_-$$
$$\breve{u}_\oplus \to u_\oplus\bar{u}_\oplus \qquad \breve{v}_\oplus \to v_\oplus\bar{v}_\oplus$$
$$\breve{u}_+ \to u_+\bar{u}_+. \qquad \breve{v}_+ \to v_+\bar{v}_+$$

We obtain the state

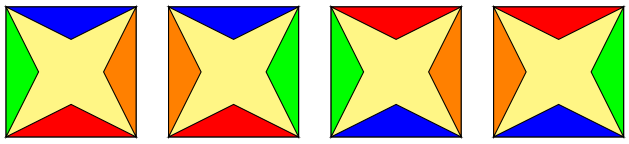
$$u_\ominus\bar{u}_\ominus \overbrace{u_-\bar{u}_- \ldots u_-\bar{u}_-}^{\lfloor \ell/2 \rfloor \text{ times}} u_\oplus\bar{u}_\oplus \overbrace{u_+\bar{u}_+ \ldots u_+\bar{u}_+}^{2r+b \text{ times}} \qquad v_\ominus\bar{v}_\ominus \overbrace{v_-\bar{v}_- \ldots v_-\bar{v}_-}^{\ell/2 \text{ times}} v_\oplus\bar{v}_\oplus \overbrace{v_+\bar{v}_+ \ldots v_+\bar{v}_+}^{2r+b \text{ times}}.$$

We are now ready to start the next computation step.

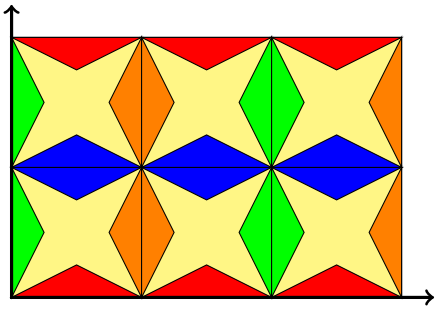
The simulation of a left movement is analogues. We conclude that we can use tag systems to simulate Turing machines.

4.3 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. Using the tiles above, we can fill a quarter of a plane such that touching edges have the same color:



We will show that the set of tile sets that can fill a quarter of a plane, is Turing complete. This will be helpful for the last section where we study the Gödel incompleteness theorem.

More formally, we consider a finite set C of *colours*. A *tile set* T is a subset of C^4. For convenience, we write the components of t as (t_N, t_E, t_S, t_W), which reflect the colours of the North, East, South and West sides of the tile. A (T,t)-*tiling* is a mapping $f : \mathbb{N}\times\mathbb{N} \to T$ such that $f(1,1) = t$ and such that the colours of neighbouring tiles match, i.e., for all $(i,j) \in \mathbb{N}\times\mathbb{N}$: $f(i,j)_E = f(i+1,j)_W$ and $f(i,j)_N = f(i,j+1)_S$.

Theorem 18. *The set of pairs (T,t) for which no (T,t)-tiling exists is Turing complete.*

Proof. We first show that the set of (T,t) for which no tiling exists is enumerable, i.e., we can "recognize" these pairs (T,t). For any $R \subset \mathbb{N}\times\mathbb{N}$, we say that f is a (T,t)-tiling for R if neighbouring colours of tiles in R match, i.e., for all $(i,j) \in R$ we have that $f(i,j)_N = f(i,j+1)_S$ and $f(i,j)_E = f(i+1,j)_W$.

Lemma 19. *There exists a (T,t)-tiling if and only if a (T,t)-tiling exists for all finite sets $R \subset \mathbb{N}\times\mathbb{N}$.*

This lemma implies that tiling problems without a solution are enumerable: one can search through all finite sets R, and check whether all possible configurations of tiles in R violate one of the conditions of a solution.

Exercise 20. We have defined the sets of colors C and tiles T to be finite sets. Show that if we also consider infinite sets of colors C and T, there exists a counter example for the above lemma. (Hint: The rough idea might be as follows. Let $C = \mathbb{N}$ and construct tiles such that in every tiling, the colours decrease when one needs to go to the right. Then create a problem when one becomes below some treshhold.)

Proof of Lemma 19. Obviously if a (T,t)-tiling exists, then there exists a tiling of any finite part of $\mathbb{N}\times\mathbb{N}$. We need to show that if there exists a tiling of any finite part, then one can construct a tiling for the whole plane.

Let $[1,n] = 1,2,\ldots,n$, and for any function f on $\mathbb{N}\times\mathbb{N}$, let $f^{(n)}$ be the restriction of this function on $[1,n]\times[1,n]$. Note that there exists at most finitely many functions from $[1,n]\times[1,n]$ to T. Let

$$f_1, f_2, f_3, \ldots$$

be a sequence such that f_n is a (T,t)-tiling of $[1,n]\times[1,n]$. We construct a solution f, by induction on n. Suppose we have defined $f(i,j)$ for all $i,j \le n$ such that $f^{(n)}$ equals $f_i^{(n)}$ for infinitely many i. Of those functions $f_i^{(n)}$ with $i \ge n+1$, there are infinitely many f_i that have

the same restriction $f_i^{(n+1)}$ (because there are finitely many functions from $[1, n+1] \times [1, n+1]$ to T). We fix f in $[1, n+1] \times [1, n+1]$ to be equal to this function.

The construction defines a function f on $\mathbb{N} \times \mathbb{N}$ and f is a (T, t)-tiling. □

We now show that for every Turing machine we can define a tiling set such that a tiling simulates represent a computation transcript of the Turing machine. The tile set T and the initial element t guarantee that there is a tiling of the plane if the machine never halts. Otherwise, if the machine halts, we make sure that a tiling becomes impossible: we let a colour appear that does not appear on any other tile. In each computation step, the configuration of a Turing machine is represented by a row of the tiling. The cells of the tape are represented by subsequent tiles that each encode the symbol of one tape cell. In every row (that corresponds to a valid computation step), there is a special tile located at the position of the computation head. This tile encodes both the state of the machine and the contents of the scanned cell.

The only technical issue appears in the movements of the computation head. We assume that in each computation step the machine always moves one cell to the right or the left, i.e., it can not remain in the same position. In every row, there is also a second special tile, located next to the position of the computation head. This tile encodes the next state of the machine and the contents of the current cell, see figure 4. To initialize the computation correctly, we will need to define some additional tiles, which can only appear on the first row of a tiling.

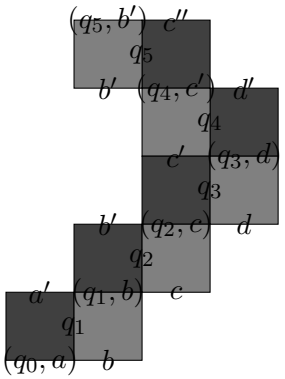


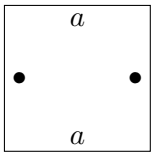
Figure 4: A tiling that simulates a Turing machine going through states $q_0, \ldots, q_5$. The machine moves three times right and one times left. Subsequent configureations of the machines are represented by rows from bottom to top. Letters in the beginning of the alphabet represent tape symbols. Darkgray tiles correspond to cells above which the computation head is located. Light gray tiles are located at the next position of the computation head.

We present now the details of the construction. They are tedious but straightforward. Consider a Turing machine $(Q, \{0, 1\}, A, \delta, q_s)$. As marked before, we can easily replace a Turing machine that does not move in some state q, by a machine that always moves if it does not halt (see exercise ...). The set of colours is given by

$$C = (Q \times \{L, R\}) \cup (Q \times A) \cup A \cup \{\square, \diamond, \bullet, \lightning\}$$

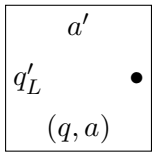
We now construct the tile set.

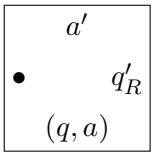
- For each tape symbol, we have a cell of the following type.



They encode tape cells on which the computation head is not located, and hence, above and below them should be tiles that encode the same symbol.

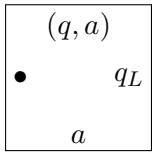
- For each $(q, a) \in \operatorname{Dom} \delta$, there is a tile that represents the reading, writing and change of state of the Turing machine. Let $\delta(q, a) = (q', a', D)$. D represents the direction in which the computation head moves. Corresponding to this direction we either add the left or the right tile to the tile set T:

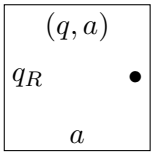




Here, q_D is shorthand for $(q, D) \in Q \times \{L, R\}$.

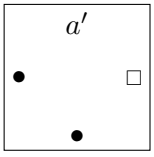
- For each $(q, a) \in Q \times A$, there is a tile that is needed to implement the movement of the head. For left and right movements we have:

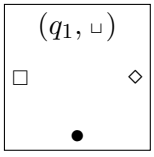


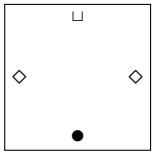


Note that only tiles of this and the above type have colours $Q \times \{L, R\} \subset C$, on the left or the right side. This implies that a tile of this type always has exactly one neighbour which is a tile of the previous type and vice versa. Also note that the colours $Q \times A \subset C$ only 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.

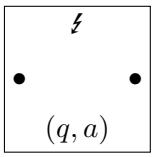
- The initial configuration is given by the state $q_s \in Q$ and a tape containing only blank symbols $\sqcup \in A$. Note that in the first computation step, the head always moves right. Let a' be the symbol that is written in this step and q_1 be the new state. (The case where the machine halts immediately is described in the next item.) To start the computation correctly, the first row of the plane should be filled with the following tiles:







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



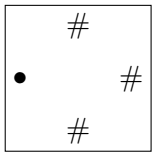
  The colour ↯ does not appear elsewhere in the tile set T, hence, such a tile can not appear in a (T, t)-tiling.

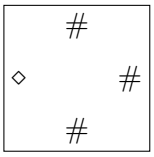
It is not hard to show by induction that in a tiling of the first n rows corresponding to n valid computation steps, there is exactly one cell of the second and third type. Therefore, the state encoded in these cells must correspond to the states of the Turing machine in this computation. Hence, if the computation never terminates, this tile set can fill the plane. Otherwise, no tiling exists. □

In section 5 we will use a closely related result. A *rectangular* (T, t)-tiling is a tiling of $[1, n] \times [1, m]$ for some n and m. For some colour $c \in C$, we say that a tiling is c-containing if it contains a tile that has a colour c.

Corollary 20. *The set of triples (c, c', T, t) such that a rectangular c and c'-containing (T, t)-tiling exists, is Turing complete.*

Proof. To the tile set constructed in the proof of Theorem 18 we add a new colour # and the tiles





These tiles can only appear on the right border of a tiling. The idea is that if such a border is present in a rectangle tiling, the computation head can not leave the rectangle during the simulated computation steps.

We show that if a Turing machine halts, then there exists an $[1, n] \times [1, m]$-tiling that contains tiles with the colours ↯ and #. Indeed, if the machine halts in the nth step, then there is a column m where the computation head never comes. In this position we have a column of tiles that have only the colour • on there west side. Hence, at this position we can place a column of #-containing tiles.

Now suppose that a rectangular colouring exists that contains the ↯ and # colours. Note that an #-containing tile can only have #-containing tiles below and above it, and it can have no other tile at the right of it. Hence, any rectangular tiling that contains an #-containing tile must have a full column of such tiles at the rightmost column. A tile that encodes a right moving computation head can not be a (left) neighbour of such a tile. Hence, we can obtain a computation transcript by assuming that all cells form the column of the #-containing cell on are blank. □

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 21. Show that there exists an oracle machine U that always halts when started on empty input with an oracle that contains a computable set, i.e., if A is computable, then $U^A(\varepsilon)$ halts. Tip: use exercise 3.

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 as a work tape by expanding the tape alphabet A_T to $\{0,1\} \times A_T$. (We don't loose any information if a cell $(b,a) \in \{0,1\} \times A_T$ is only overwritten by symbols (b,a').) In this way we can still read and write anything on the tape without loosing the information of the oracle. Now, you can understand why the required tile set exists.

Exercise 22. 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 there exist no proof systems that are both correct and can prove all true arithmetical formulas. We assume the reader is familiar with basic logical notation.

Arithmetical formulas are formulas that contain variables that have values over the non-negative integer numbers, have equality relations in them, contain the constants 0 and 1, contain operations for addition $+$ and multiplication $*$, contain the logical connectives $\wedge$ "and", $\vee$ "or", $\neg$ "not", $\Rightarrow$, and the quantifiers $\forall$ "for all" and $\exists$ "there exists". 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). We say that a formula is true if it is true in the standard interpretation over the set $\{0\} \cup \mathbb{N} = 0, 1, 2, \ldots$. For example, $\forall c(F(c))$ is true if the expression $F(c)$ is true for all non-negative integers c.

Some examples of arithmetical formula: $0 = 1$ and $\exists a(a * a = 2)$. Both formulas are false. The formula $\forall a \forall b \left(a^2 - b^2 = (a+b)(a-b)\right)$ is true. In the construction of bigger formula we also consider expressions with free variables: the formulas $a = b \Rightarrow a^2 = b^2$, $\exists c(a + c = b)$ 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 two others formulas are equivalent to $a \leq b$.

The following proposition shows that the set of arithmetical formula is Turing complete. Recall the definition of a rectangular tiling given below the proof of Theorem 18.

Proposition 21. *There exists a computable function that maps every quadruple (c, c', T, t) to an arithemtical formula that is true if and only if there exists a c and c'-containing rectangular (T,t)-tiling.*

Let us explain the main technical difficulty using some one dimensional variant of the problem. Suppose we are given a number N and a relation $F \subset \mathbb{N} \times \mathbb{N}$ which is expressible as an arithmetical formula. Is there a formula that is equivalent to:

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

Arithmetical formula have no quantifiers over lists of natural numbers. So we use an indirect approach, inspired by how we write numbers in some base b. Given a large number L, we get the ith number in base b by $\lfloor L/b^i \rfloor \bmod b$, where $\lfloor x \rfloor$ denotes the smallest integer that does not exceed x. From now on we only use integer devision, hence, we can omit $\lfloor \cdot \rfloor$. We show that the statement above is equivalent to

$$\begin{aligned}\exists L, b, \overline{B}\Big[&\text{“all dividers of } \overline{B} \text{ are powers of } b\text{”} \\ &\wedge L \bmod b = 1 \\ &\wedge \exists B\Big(B|\overline{B} \wedge (L/B) \bmod b = N\Big) \\ &\wedge \forall B\left(Bb|\overline{B} \Rightarrow F\Big((L/B) \bmod b, (L/bB) \bmod b\Big)\right)\Big].\end{aligned}$$

We first show that the claim implies the formula. If the list contains only one element, then $N = 1$. The formula is true for $L = \overline{B} = 1$ and $b = 2$. Suppose that for some $n \geq 2$, a list of numbers $\ell_1, \ldots, \ell_n$ exists that satisfies the statement. Let b be a prime number that exceeds all these numbers. Let $\overline{B} = b^{n-2}$ and let $L = \sum_1^n \ell_i b^{i-1}$. The divisors of $\overline{B}$ are exactly b^i for $i = 0, \ldots, n-2$, and hence, the possible values for $((L/B) \bmod b, (L/bB) \bmod b)$ in the formula above are precisely (ℓ_i, ℓ_{i+1}) for $i = 1, \ldots, n-1$.

We now show that the formula implies the claim. First note that the statement “all dividers of $\overline{B}$ are powers of b” implies that $\overline{B} \neq 0$, because all natural numbers divide zero. It also implies that $b \neq 0$ because 1 always divides $\overline{B}$ and is not a power of zero. The list $\ell_1, \ldots, \ell_n$ is given by $L \bmod b$, $(L/b) \bmod b$, $L/b^2 \bmod b$, up to the maximal b^i such that $b^i|\overline{B}$. This list satisfies all properties.

The statement above can be rewritten as an arithmetical formula by using the following equivalences: $a \neq b \Leftrightarrow \neg(a = b)$ and for any expression E we have

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

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

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

Indeed, the $\Downarrow$-direction is direct. For the $\Uparrow$ direction, assume that M has a divisor d that is not a power of b. Let $d = b^i v$ for some v such that $b \nmid v$. Note that $v \neq 1$ and $v|M$. Hence, $uv = M$ with $u \neq 0$. implies that $M > ub(v/b)$, i.e., the formula is false.

Proof of Proposition 21. We need to encode a 2d-array of tiles as a single number. We use the technique above twice. First we encode every row as a single number, and then we encode all these numbers as a single number. Let A be the number that encodes the array and let

$$[A]_{B,D}^{b,d} = \Big((A/D \bmod d)/B\Big) \bmod b$$

The tile at position $(i+1, j+1)$ is represented as $[A]_{b^i,b^j}^{b,c}$ for $i, j \in [0, n-1]$.

Let $t_1, t_2, \ldots, t_{|T|}$ be a list of all the tiles in T. Let H and V be the sets of tiles (i, j) with matching horizontal and vertical colors, i.e., $H = \{(i, j) : t_{i,E} = t_{j,W}\}$ and $V = \{(i, j) :$

$t_{i,N} = t_{j,S}\}$. For $c \in C$, let IT_c be the sets of indices of tiles that contain the colour c. The statement "There exists a rectangular c and c'-containing (T, t_1)-tiling" is equivalent to

$$\begin{aligned}\exists A, b, \overline{B}, d, \overline{D}\Big[&\text{“all dividers of } \overline{B} \text{ are powers of } b\text{”}\\ &\wedge \text{“all dividers of } \overline{D} \text{ are powers of } d\text{”}\\ &\wedge [A]^{b,d}_{1,1} = 1\\ &\wedge \exists B, D\Big(B|\overline{B} \wedge D|\overline{D} \wedge [A]^{b,d}_{B,D} \in IT_c\Big)\\ &\wedge \exists B, D\Big(B|\overline{B} \wedge D|\overline{D} \wedge [A]^{b,d}_{B,D} \in IT_{c'}\Big)\\ &\wedge \forall B\left((B|(\overline{B}/b) \wedge D|\overline{D}) \Rightarrow H\Big([A]^{b,d}_{B,D}, [A]^{b,d}_{bB,D}\Big)\right)\\ &\wedge \forall B\left((B|\overline{B} \wedge D|(\overline{D}/d)) \Rightarrow V\Big([A]^{b,d}_{B,D}, [A]^{b,d}_{B,dD}\Big)\right)\Big]\end{aligned}$$

With the same observations as above, this formula can be transformed to an arithmetical formula; it remains to note that testing whether $x \in S$ for some finite set $S = \{s_1, \ldots, s_e\}$, is the same as $x = s_1 \wedge \ldots \wedge x = s_e$.

Using the observations above the proof, we can show that this statement is equivalent with the existence of a rectangular c and c'-containing (T, t_1)-tiling. Moreover, the corresponding formula is computable from (c, c', T, t_1). □

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 enumerable.

Corollary 22. *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 NOT(w) is true (and not both). Hence, either w or NOT(w) is provable. The third item of exercise 6 implies that the set of true formulas is decidable. On the other hand, the Halting problem is an enumerable set, and by Proposition 21 it is expressible as an arithmetical set. Hence, we can solve the Halting problem by deciding whether some sequences are true; and this is impossible. Therefore, a correct proof system can not prove all true formulas. □

We can prove the corollary in a more direct (and closely related way). Consider the following:

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 supports Gödel's claim.

How can we express the statement above using an arithmetical formula? Consider a Turing machine U that takes a binary encoding x of an arithmetical formula α as input and

searches for a proof. Upon finding such a proof it outputs 1, otherwise it runs forever. By Proposition 21 there exists an arithmetical statement that is equivalent to $U(x) = 1$. 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. Note that we can make a list of arithmetical formulas containing all formulas with one free variable, i.e., that have one variable in them that is not in the scope of a quantifier ($\exists$ or $\forall$). We can make a Turing machine V that on input n computes the nth such formula and replaces the variable with the value n. Then V runs the machine above to check whether this formula has a proof. The statement $V(n) = 1$ means: "After substituting the value n for the free variable in the nth formula, we obtain a provable formula." We can now apply Proposition 21 to 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." The Nth formula in the list is NOT $\alpha(n)$, thus after replacing n with the value N, we obtain the formula itself. So it is either unprovable or false.