Configurations and computations as data
Universal Turing machine

Configurations as data

Let us recall what are the **configurations** of different kinds of machines:

- **one-tape Turing machine**:
  — a state of the control unit, a content of the tape, a position of the head
- **multitape Turing machine**:
  — a state of the control unit, contents of all tapes, positions of all heads
- **Random Access Machine**:
  — instruction pointer, a content of working memory, contents of the input and output tapes
- **control-flow graph**:
  — a control state (a node in the control-flow graph), a content of memory (values of all variables)
- **Minsky machine**:
  — a state of the control unit, values of all counters

Configurations as data

In all these cases (and also for other models of computations), configurations of a given machine are **finite** object that can be manipulated as **data**:

- in high-level programming languages, they can be for example represented as a suitable **data structure**.
- they can be also represented as **words** over some alphabet — it is necessary to choose some reasonable format how configurations are written

These data objects (data structures, objects, etc.) representing configurations can be identified with these configurations:

- For example, when we say that something can be done with configuration α, we mean that in fact, we work with configuration α in a form of data.

Configurations as data

Consider some paricular machine $\mathcal{M}$:

- Let *Conf* denote the set all configurations of a given machine $\mathcal{M}$.
- Let us define relation

$$\longrightarrow \subseteq \mathit{Conf} \times \mathit{Conf}$$

as the of those pairs of configurations α and α', for which it holds that the machine $\mathcal{M}$ can go by **one step** from configuration α to configuration α', which will be written as

$$\alpha \longrightarrow \alpha'$$

- Some configurations from the set *Conf* are denoted as **final** — the computation (successfully) ends in them.

  It holds for each final configuration α that there is no configuration α' such that $\alpha \longrightarrow \alpha'$

Configurations as data

A **computation** of a given machine $\mathcal{M}$ for inpute $w \in In$ (where In is the set of possible inputs) is a finite or infinite sequence of configurations from the set $Conf$, i.e.,

- **finite computation**: $\alpha_0, \alpha_1, \alpha_2, \ldots, \alpha_t$
- **infinite computation**: $\alpha_0, \alpha_1, \alpha_2, \ldots$

where:

- α_0 is an **initial configuration** for input w
- For each $i \in \mathbb{N}$ (in the case of a finite computation, for each $i \in \mathbb{N}$ such that $i < t$) we have
$$\alpha_i \longrightarrow \alpha_{i+1}$$
- In the case of finite computation, α_t is a final configuration.

Remark: In the case of a **deterministic** machine $\mathcal{M}$, for each configuration α, there is at most one configuration α' such that $\alpha \longrightarrow \alpha'$.

Configurations as data

We can perform different operations on data objects representing configurations of machine $\mathcal{M}$:

- to construct an **initial configuration** α_0 for a given input w of machine $\mathcal{M}$
- to test whether α is a **final configuration** of machine $\mathcal{M}$ — if this is the case, then to determine the output of machine $\mathcal{M}$ that halted in this configuration α
- to compute, for a given configuration α that is not final, a configuration α' such that machine $\mathcal{M}$ can go from configuration α to configuration α' in one step
- to test for a given pair of configurations α and α' whether machine $\mathcal{M}$ can go in one step from configuration α to configuration α'

Such operations can be easily implemented as algorithms.

Configurations as data

Using these operations, for example a **simulation** of a computation of machine $\mathcal{M}$ over input w can be implemented:

Algorithm: Simulation of a computation of machine $\mathcal{M}$ over input w

```
RUN(w):
    α := INIT-CONF(w)
    while not IS-FINAL(α) do
        α := NEXT-CONF(α)
    return EXTRACT-OUTPUT(α)
```

where:

- INIT-CONF(w) — computes an initial configuration of machine $\mathcal{M}$ for input w
- IS-FINAL(α) — tests if α is a final configuration
- NEXT-CONF(α) — computes configuration α' such that $\alpha \longrightarrow \alpha'$
- EXTRACT-OUTPUT(α) — extracts an output of machine $\mathcal{M}$ from final configuration α

Computations as data

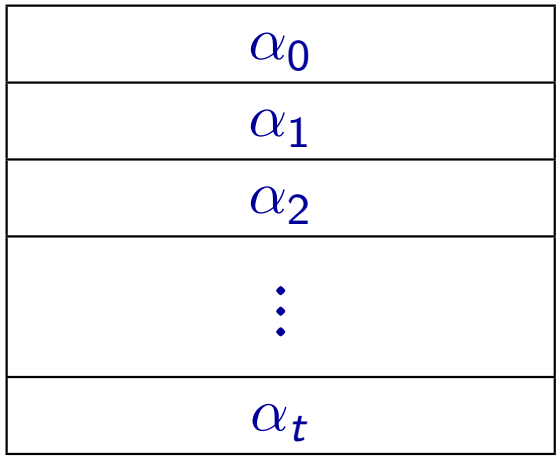
We can work with a **finite** computation

$$\alpha_0 \longrightarrow \alpha_1 \longrightarrow \quad \cdots \quad \longrightarrow \alpha_t$$

as with data.

It can be represented for example as:

- some appropriate **data structure** represening a sequence of configurations



Computations as data

We can work with a **finite** computation

$$\alpha_0 \longrightarrow \alpha_1 \longrightarrow \quad \cdots \quad \longrightarrow \alpha_t$$

as with data.

It can be represented for example as:

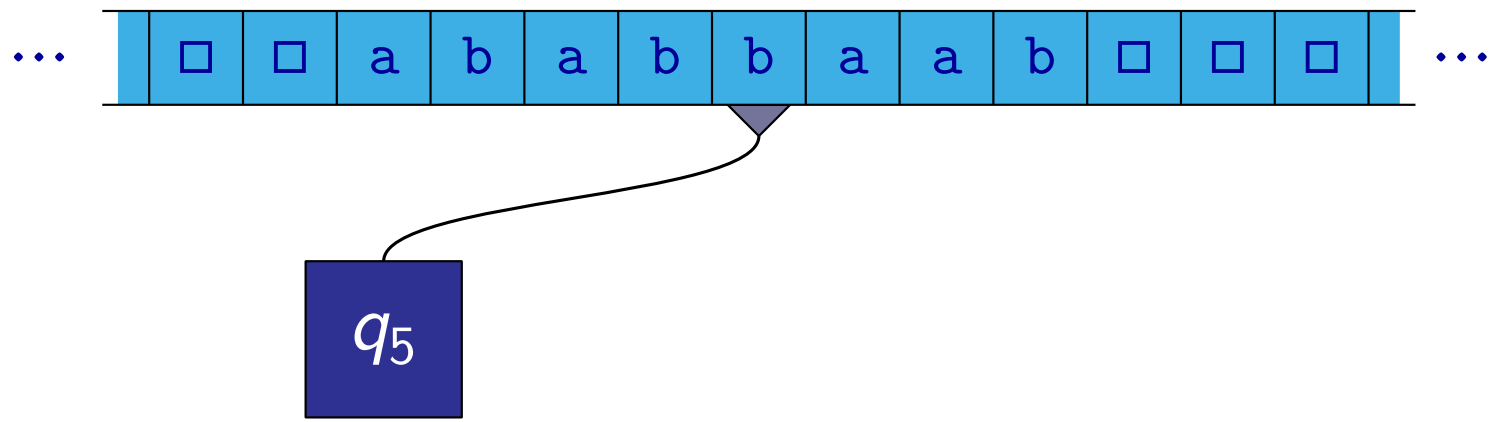
- a **word**

$$\#\alpha_0\#\alpha_1\#\alpha_2\# \quad \cdots \quad \#\alpha_t\#$$

  where:

  - $\alpha_0, \alpha_1, \ldots, \alpha_t$ — words representing configurations of the computation
  - $\#$ – a symbol chosen as a separator of configurations (it does not occur in the words representing configurations)

Configurations of a Turing machine as data

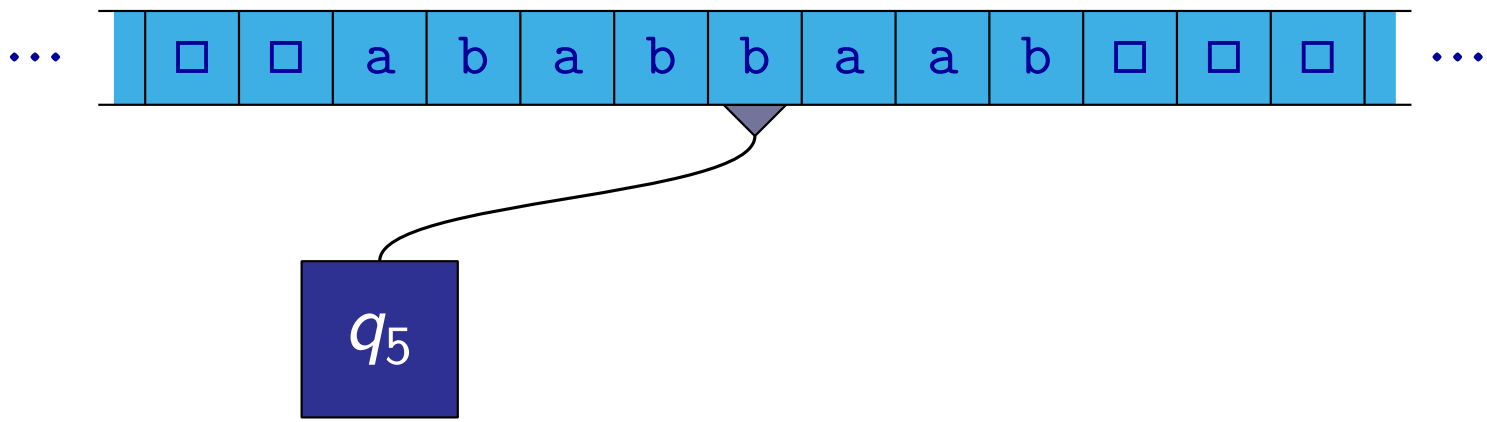


Consider for example a one-tape Turing machine $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, q_0, F)$.

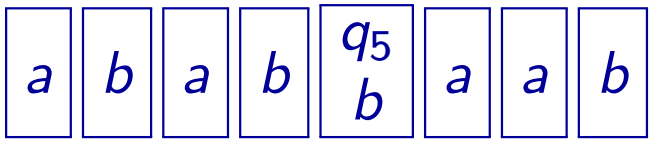
A configuration of such a machine must contain an information about:

- a state of the control unit
- a content of the tape
- a position of the head

Configurations of a Turing machine as data

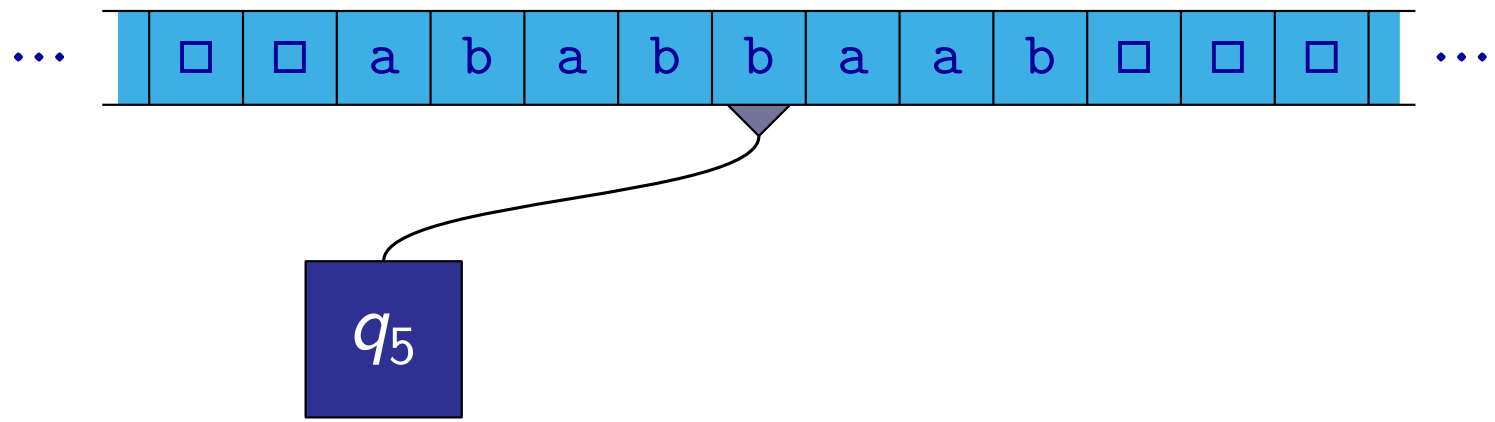


Configurations can be represented for example as words over alphabet $\Delta = \Gamma \cup (Q \times \Gamma)$:

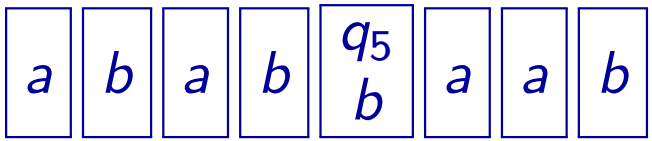


This word always contains exactly one symbol from $(Q \times \Gamma)$ that denotes a state of the control unit and a position of the head.

Configurations of a Turing machine as data

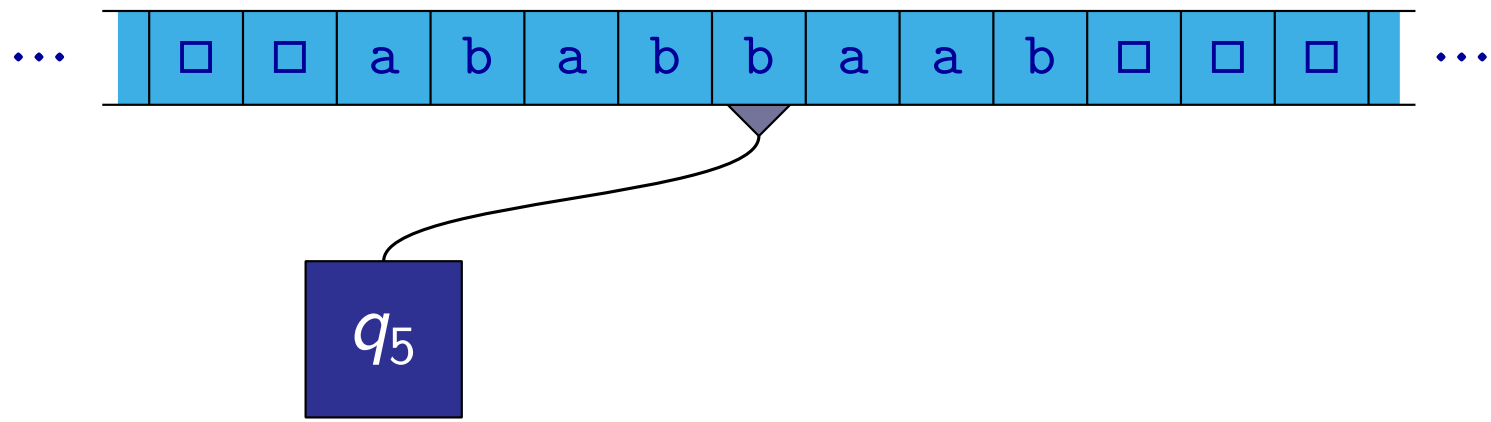


Configurations can be represented for example as words over alphabet $\Delta = \Gamma \cup (Q \times \Gamma)$:

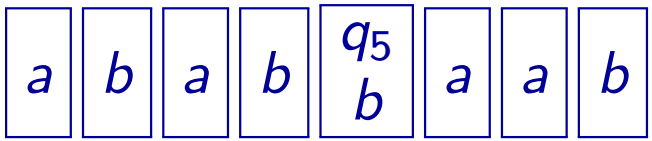


Remark: Symbols from $(Q \times \Gamma)$ can be also written as $\boxed{\begin{matrix} q \\ a \end{matrix}}$ instead of $\boxed{(q, a)}$.

Configurations of a Turing machine as data

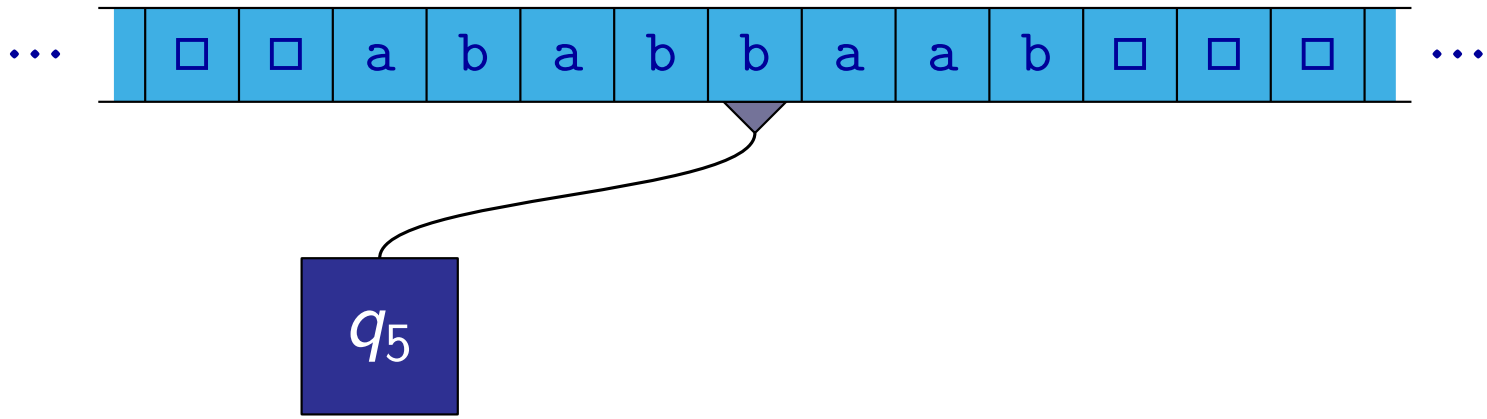


Configurations can be represented for example as words over alphabet $\Delta = \Gamma \cup (Q \times \Gamma)$:

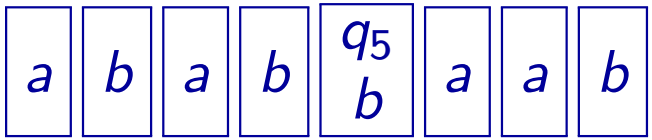


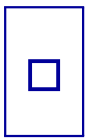
Other symbols (from Γ) represent a content of the tape.

Configurations of a Turing machine as data



Configurations can be represented for example as words over alphabet $\Delta = \Gamma \cup (Q \times \Gamma)$:



Those cells of the tape that are not included in the word contain the symbol  .

Configurations of a Turing machine as data

Let $Conf$ denote the set all configurations of a given machine $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, q_0, F)$.

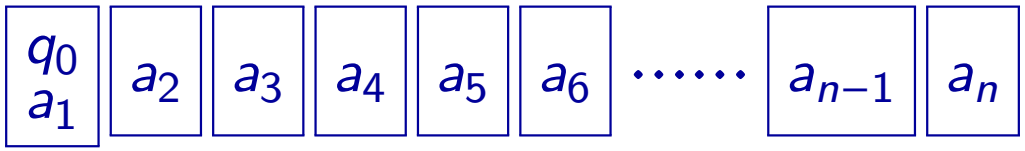
- If we represent configurations of the Turing machine $\mathcal{M}$ as words over the alphabet $\Delta = \Gamma \cup (Q \times \Gamma)$, the set $Conf$ can be identified with the set of those words over alphabet Δ that contain exactly one occurrence of a symbol from $(Q \times \Gamma)$, i.e., with the set of words of the form

$$u(q, a)v$$

where $q \in Q$, $a \in \Gamma$, $u, v \in \Gamma^*$.

Configurations of a Turing machine as data

The **initial configuration for an input** $w \in \Sigma^*$, where $w = a_1 a_2 \cdots a_n$, is then represented as the word



or (when we use a standard mathematical notation) as:

$$(q_0, a_1)\, a_2\, a_3 \cdots a_{n-1}\, a_n$$

Remark: If $w = \varepsilon$ then the initial configuration is represented as the word $(q_0, \square)$.

Configurations of a Turing machine as data

For each configuration α of the form

$$u(q,a)v$$

where $q \in (Q - F)$, $a \in \Gamma$, $u, v \in \Gamma^*$, there exists exactly one configuration α' such that $\alpha \longrightarrow \alpha'$.

This configuration α' is determined by the transition function δ.

Let us assume that $\delta(q,a) = (q', a', d)$:

- If $d = 0$ then $\alpha' = u(q', a')v$.
- If $d = -1$:
  - If $u = \varepsilon$ then $\alpha' = (q', \Box)a'v$.
  - If $u = u'b$ (where $u' \in \Gamma^*$ and $b \in \Gamma$) then $\alpha' = u'(q', b)a'v$.
- If $d = +1$:
  - If $v = \varepsilon$ then $\alpha' = ua'(q', \Box)$.
  - If $v = bv'$ (where $b \in \Gamma$ and $v' \in \Gamma^*$) then $\alpha' = ua'(q', b)v'$.

Configurations of a Turing machine as data

Final configurations are configurations of the form

$$u(q, a)v$$

where $q \in F$, $a \in \Gamma$, $u, v \in \Gamma^*$.

Computations of a Turing machine as data

A finite computation of a Turing machine

$$\alpha_0 \longrightarrow \alpha_1 \longrightarrow \alpha_2 \longrightarrow \alpha_3 \longrightarrow \quad \cdots \quad \longrightarrow \alpha_{t-1} \longrightarrow \alpha_t$$

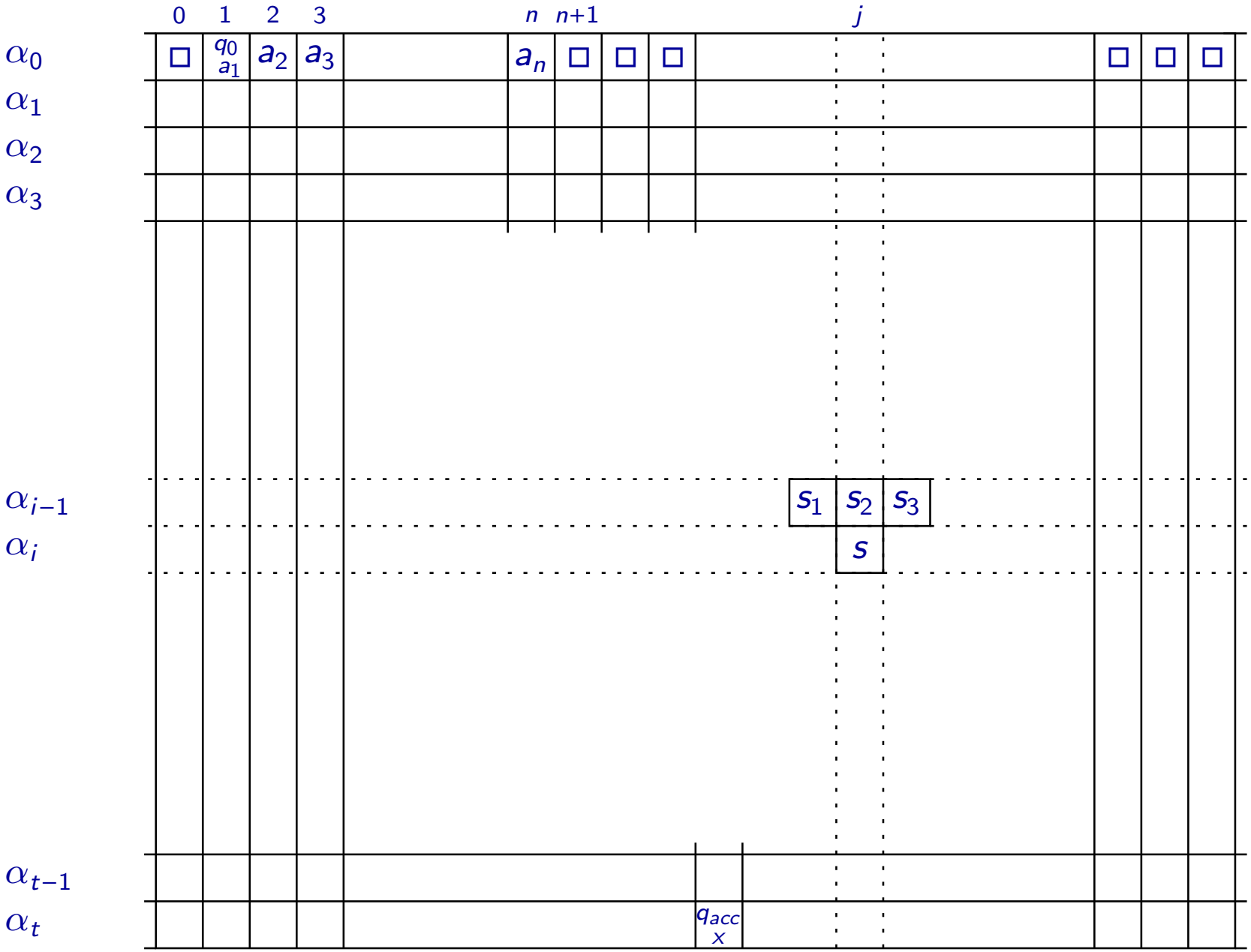
then can be represented as for example:

- a sequence of configurations separated by a special symbol $\# \notin \Delta$.

  — i.e., as one long **word** over alphabet $\Delta \cup \{\#\}$

- A **table** whose cells contain symbols from alphabet Δ, and where rows correspond to individual configurations, and columns to positions on the tape.

  Note that in this representation a content of cell i (where $i > 0$) and in column j is completely determined by a description of machine $\mathcal{M}$ and the content of cells in columns $j-1$, j, and $j+1$ in row $i-1$.


0
1
2
3
n
n+1
j
α_0
α_1
α_2
α_3
α_{i-1}
α_i
α_{t-1}
α_t
q_0
a_1
a_2
a_3
a_n
s_1
s_2
s_3
s
q_{acc}
x

Computations of a Turing machine as data

When we have such representation of a finite computation of a machine $\mathcal{M}$ on input w, for example, in the form of a word consisting of a sequence of configurations, in the of a table, etc., we can test for this representation for example if:

- α_0 is really the initial configuration of machine $\mathcal{M}$ for input w.
- for each $i < t$ it holds that $\alpha_i \longrightarrow \alpha_{i+1}$.
- α_t is the final configuration, and what is the corresponing output of machine $\mathcal{M}$ in this configuration.

For example, the machine $\mathcal{M}$ gives answer YES for a given input w iff there exists a representation of the computation of the machine $\mathcal{M}$ on word w satisfying the above conditions, and moreover that the machine $\mathcal{M}$ return YES in the final configuration α_t as an output.

Code of a Turing machine as Data

A description of arbitrary Turing machine $\mathcal{M}$ (or other computation model) can be represented in a form of data — e.g., a word over some alphabet.

Let $Code(\mathcal{M})$ denote such representation of machine $\mathcal{M}$ (in some particular format).

- A representation $Code(\mathcal{M})$ contains information about states of the control unit, a transition functions, etc.
- $Code(\mathcal{M})$ can be viewed as a code of a program.

Universal Turing Machine

A **universal Turing machine** $\mathcal{U}$ is a machine that when obtains a word $Code(\mathcal{M})$ and word $w \in \Sigma^*$ (where Σ is the input alphabet of the machine $\mathcal{M}$), starts to simulate a computation of the machine $\mathcal{M}$ over input w.

(The machine $\mathcal{U}$ can obtain the input $Code(\mathcal{M})$ and w for example as a word $Code(\mathcal{M})\#w$.)

So the universal Turing machine is able to perform a computation of any other Turing machine (whose description it obtains as a part of its input).

Remark: This corresponds to a situation where we have:

- hardware of a computer (machine $\mathcal{U}$), which is able to execute an arbitrary algorithm
- code of a program ($Code(\mathcal{M})$) running on this computer
- input data for this program (word w)

Universal Turing Machines

Remark: It is possible to construct suprisingly small universal Turing machines.

For example, there are known universal Turing machines simulating behaviour of a given one-tape Turing machine with:

- 3 states and 11 symbols
- 5 states and 7 symbols
- 6 states and 6 symbols
- 7 states and 5 symbols
- 8 states and 4 symbols

(Here, the number of states is the number of state of the control unit Q where finil states are not counted. Number of symbols is the size of the tape alphabet Γ of the given universal machine.)

Even smaller universal Turing machines can be constructed but they do not simulate Turing machines but some other (extremely restricted) Turing complete models.

Undecidable Problems

Algorithmically Solvable Problems

Let us assume we have a problem P.

If there is an algorithm solving the problem P then we say that the problem P is **algorithmically solvable**.

If P is a decision problem and there is an algorithm solving the problem P then we say that the problem P is **decidable (by an algorithm)**.

If we want to show that a problem P is algorithmically solvable, it is sufficient to show some algorithm solving it (and possibly show that the algorithm really solves the problem P).

Algorithmically Unsolvable Problems

A problem that is not algorithmically solvable is **algorithmically unsolvable**.

A decision problem that is not decidable is **undecidable**.

Surprisingly, there are many (exactly defined) problems, for which it was proved that they are not algorithmically solvable.

Halting Problem

Let us consider some general programming language $\mathcal{L}$.

Futhermore, let us assume that programs in language $\mathcal{L}$ run on some idealized machine where a (potentially) unbounded amount of memory is available — i.e., the allocation of memory never fails.

Example: The following problem called the **Halting problem** is undecidable:

Halting problem

Input: A source code of a $\mathcal{L}$ program P, input data x.

Question: Does the computation of P on the input x halt after some finite number of steps?

Halting Problem

Let us assume that there is a program that can decide the Halting problem.

So we could construct a subroutine H, declared as

Bool H(String code, String input)

where $H(P, x)$ returns:

- true if the program P halts on the input x,
- false if the program P does not halt on the input x.

Remark: Let us say that subroutine $H(P, x)$ returns false if P is not a syntactically correct program.

Halting Problem

Using the subroutine H we can construct a program D that performs the following steps:

- It reads its input into a variable x of type String.
- It calls the subroutine $H(x, x)$.
- If subroutine H returns true, program D jumps into an infinite loop

```
loop: goto loop
```

In case that H returns false, program D halts.

What does the program D do if it gets its own code as an input?

Halting Problem

If D gets its own code as an input, it either halts or not.

- If D halts then $H(D, D)$ returns true and D jumps into the infinite loop. A contradiction!
- If D does not halt then $H(D, D)$ returns false and D halts. A contradiction!

In both case we obtain a contradiction and there is no other possibility. So the assumption that H solves the Halting problem must be wrong.

Semidecidable Problems

A problem is **semidecidable** if there is an algorithm such that:

- If it obtains as an input an instance of the problem P, for which the correct answer is YES, it will halt after a finite number of steps on this input, and gives answer YES.
- If it obtains as an input an instance of the problem P, for which the correct answer is NO, it will halt on this input and gives answer NO, or it will not halt on this input and runs on it forever.

Semidecidable Problems

Remarks:

- So an algorithm $\mathcal{A}$ that semi-decides a problem P, need not halt for all inputs but only needs to halt for those inputs where answer is YES.
- The algorithm $\mathcal{A}$ of course should not give incorrect answers for those inputs where it halts.
- It is obvious that every decidable problem is also semidecidable:
  — An algorithm $\mathcal{A}$ that solves a given decision problem P also semidecides this problem.
- There exist semidecidable problems that are not decidable.

Semidecidable Problems

Example: **Halting problem (HP)** is a typical example of a problem that is semidecidable but not decidable.

Algorithm $\mathcal{A}$ that semidecides HP:

- To read program P and its input data x.
- To simulate behaviour of program P on input data x step by step.
- If this simulation ends (i.e., if the computation of program P on data x halts after some finite number of steps), then to write answer YES.
- If the computation of program P on input data x never halts, then also the simulation runs forever.

  — This is a correct behaviour since the correct answer in this case is NO, and so the algorithm $\mathcal{A}$ can run forever for the input consisting of the pair P and x.

Complement problems

The **complement** problem for a given decision problem P is a problem where inputs are the same as for the problem P and the question is the negation of the question from the problem P.

Examples:

- The complement problem of the Halting problem:

Input:	A source code of a $\mathcal{L}$ program P, input data x.
Question:	Does the computation of P on the input x not halt after some finite number of steps?

- The complement problem of the SAT problem:

Input:	Boolean formula φ.
Question:	Is formula φ unsatisfiable (i.e., is it a contradiction)?

Complement problems

- If P is a decision problem, its complement problem will be denoted $\overline{P}$.
- It is obvious that problem P is decidable iff its complement problem $\overline{P}$ is decidable:

  — Using algorithm $\mathcal{A}$ solving problem P, we can easily construct an algorithm $\mathcal{A}'$ solving problem $\overline{P}$ in such a way that $\mathcal{A}'$ calls $\mathcal{A}$ as a subroutine with the given input, negates the output produced by $\mathcal{A}$, and this negated answer gives as its output.

  — In the same way, an algorithm solving problem $\overline{P}$ can be transformed into an algorithm solving problem P.
- But if problem P is not decidable, it is not possible that both P and $\overline{P}$ would be semidecidable.

  This follows from the following theorem.

Post's Theorem

Post's Theorem

If a problem P and its complement problem $\overline{P}$ are semidecidable then the problem P is decidable.

Proof:

- Let us assume that problem P is semidecidable, and so that there is an algorithm $\mathcal{A}_1$ semideciding it.

  Moreover, we can assume that the algorithm $\mathcal{A}_1$ halts exactly for those instances of problem P where the answer is $\textsc{Yes}$.

- Let us assume also that problem $\overline{P}$ is semidecidable, and so that there is an algorithm $\mathcal{A}_2$ that semidecides it.

  Moreover, we can assume that the algorithm $\mathcal{A}_2$ halts exactly for those instances of problem $\overline{P}$ where the answer is $\textsc{Yes}$, which are exactly those instances where in problem P the answer is $\textsc{No}$.

Post's Theorem

- An algorithm solving problem P can not call algorithms $\mathcal{A}_1$ and $\mathcal{A}_2$ as subroutines because these algorithms can not halt for some inputs.
- But for each instance x of problem P, one of these algorithms always halts after some finite number of steps.
- So the algorithm $\mathcal{A}$ solving problem P can work by simulating computations of both algorithms $\mathcal{A}_1$ and $\mathcal{A}_2$ on input x **in parallel**:
  - It stores configurations of both algorithms running in parallel.
  - It alternates in performing steps of algorithms $\mathcal{A}_1$ and $\mathcal{A}_2$.
  - As soon as one of these algorithms reaches a final configurations, the algorithm $\mathcal{A}$ halts and returns the corresponding answer:
    YES — if $\mathcal{A}_1$ has halted, NO — if $\mathcal{A}_2$ has halted

Post's Theorem

The following follows from Post's theorem:

The complement problem of Halting problem is not semidecidable.

Proof:

- Halting problem is semidecidable.
- if its complement problem would be semidecidable, then Post's theorem would imply that the Halting problem is decidable.
- But this is a contradiction with the previously proved fact that Halting problem is undecidable. So the complement problem of Halting problem can not be semidecidable.

The same arguments can be used to show for any semidecidable problem P that is not decidable, that its complement problem $\overline{P}$ is not semidecidable.

Semidecidability — alternative characterizations

Semidecidability of problem P can be alternatively chacterized as follows:

- Let us assume that In se the set of inputs of problem P.
- Let us assume also that $\mathcal{W}$ is the set of **potential witnesses** of the fact that for a given instance $x \in In$, the correct answer is $\textsc{Yes}$:

A problem P is **semidecidable** iff there exists an algorithm $\mathcal{A}$ that for every pair (x, w), where $x \in In$ and $w \in \mathcal{W}$, determines after a finite number of steps, whether w is a **actual witness** confirming that the answer for x is really $\textsc{Yes}$, i.e.,

$$\text{the answer for } x \in In \text{ is } \textsc{Yes}$$
$$\Longleftrightarrow$$
$$\text{there exists an } \textbf{actual witness } w \in \mathcal{W}$$

Semidecidability — alternative characterizations

Let us assume for simplicity that In and $\mathcal{W}$ are sets of words over some alphabet Σ (e.g., $\{0, 1\}$).

- Let us assume that for problem P there exists an algorithm $\mathcal{A}$ that halts (and given the answer $\textsc{Yes}$) exactly for those inputs $x \in In$ where the answer is $\textsc{Yes}$:
  - As the set of potential witnesses $\mathcal{W}$ we can take the representations of computations of algorithm $\mathcal{A}$ over differents inputs (in the form of a sequence of configurations).
  - A sequence of configurations w will be an (actual) witness for x iff w is a correct representation of the algorithm $\mathcal{A}$ over input x that gives the answer $\textsc{Yes}$.

Semidecidability — alternative characterizations

- Let us assume that for problem P there exists a set of potential witnesses $\mathcal{W}$ and an algorithm $\mathcal{A}$ that for each pair (x, w), where $x \in In$ and $w \in \mathcal{W}$ decides whether w is an actual witness of the fact that the answer for x is YES.

  We can construct an algorithm $\mathcal{A}'$ that semidecides the problem P:

  - The algorithm reads an input $x \in In$.
  - The algorithm $\mathcal{A}'$ will systematically generate all potential witnesses, i.e., all elements of the set $\mathcal{W}$, as a sequence $w_0, w_1, w_2, \ldots$ (e.g., all words over a given alphabet in the order according to their lengths and sorted lexicographically for the same length)
  - For each generated potential witness w_i, it will call the algorithm $\mathcal{A}$ as a subroutine for the pair (x, w_i).
    If $\mathcal{A}$ returns the answer YES, the algorithm $\mathcal{A}'$ halts with the answer YES.
    Otherwise, it continues with generating the next potential witness.

Semidecidability — alternative characterizations

Other possible characterization of semidecidable problems looks as follows:

A problem P se **semidecidable** iff there exists an algorithm $\mathcal{A}$ that:

- It does not expect anything on the input.
- It runs forever.
- it successively produces as an output a sequence of instances of problem P:

  $$x_0, x_1, x_2, \ldots$$

  Individual instances are seperated by some reasonable way, e.g., by some special symbol, so it is easy to recognize when output of each individual instance was finished.
- This sequence consists of exactly those instances of problem P where the correct answer is YES, i.e., each such instance will appear in this sequence after some finite number of steps of the algorithm $\mathcal{A}$.

Semidecidability — alternative characterizations

If have an algorithm $\mathcal{A}$ generating a (possibly infinite) sequence of all instances of problem P, for which that answer is YES, we can use it to construct an algorithm $\mathcal{A}'$ that semidecides the problem P:

- The algorithm $\mathcal{A}'$ reads input x and stores it into memory.
- The algorithm $\mathcal{A}'$ starts to simulate individual steps of the algorithm $\mathcal{A}$.
- Everytime, when $\mathcal{A}$ generates a next instance x_i of problem P, the simulation is interrupted and $\mathcal{A}'$ checks if $x_i = x$.

  It is is true that $x_i = x$, the computation of $\mathcal{A}'$ ends and $\mathcal{A}'$ produces YES as an output.

  If $x_i \neq x$, then $\mathcal{A}'$ continues in the simulation of algorithm $\mathcal{A}$ as long as until the next instance, for which the answer is YES, is generated.

Semidecidability — alternative characterizations

On the other hand, if we have an algorithm $\mathcal{A}$ that semidecides problem P (i.e., it halts and returns the answer YES for exactly those inputs where the answer is YES), then it is possible to construct an algorithm $\mathcal{A}'$ that runs forever and successively generates a sequence of all instances of the problem P where the answer is YES:

- The algorithm $\mathcal{A}'$ will successively generate all instances of the problem P from the set In.
- The algorithm $\mathcal{A}'$ will simulate the behaviour of the algorithm $\mathcal{A}$ on these instances.
- But the algorithm $\mathcal{A}'$ can not do this in such a way that it would start to simulate the computation of the algorithm $\mathcal{A}$ on a given instance x_i and simulate it as long until this computation halts, since this computation can run forever, and $\mathcal{A}'$ would never start to simulate computations for remaining instances.

Semidecidability — alternative characterizations

- Instead, the algorithm $\mathcal{A}'$ will simulate the computations of the algorithm $\mathcal{A}$ on many inputs in parallel.

  It will forever repeat the cycle where in one interation of the cycle always performs the following operations:

  - It will simulate one step of a computation for each computation simulated so far.
  - The set of computations simulated so far will be extended with the next simulated computation of the algorithm $\mathcal{A}$ where its input will be the next instance from the set In.
  - As soon as one of the simulated computations halts with the answer YES, the algorithm $\mathcal{A}$ will write the input of this computation to the output.

    (To write this outpus, the algorithm $\mathcal{A}'$ must remember for each simulated computation what was the input for this computation.)

Semidecidability — alternative characterizations

The following fourth characterization of semidecidable problems is similar to the previous one:

Problem P is **semidecidable** iff there exists an algorithm $\mathcal{A}$ with the following properties:

- It expects as an input a natural number i (i.e., $i \in \mathbb{N}$).
- For each input $i \in \mathbb{N}$, the algorithm $\mathcal{A}$ halts after some finite number of steps and produces some instance of the problem P, for which the answer is $\textsc{Yes}$, as an output.
- If denote the output of the algorithm $\mathcal{A}$ for input i as $f(i)$, then the infinite sequence

$$f(0), f(1), f(2), \ldots$$

will contain all instances of the problem P where the answer is $\textsc{Yes}$.

Semidecidability — alternative characterizations

Remark:

- Above mentioned characterizations assumes that there is at least one instance of problem P, for which the answer is YES.
- It is obvious how using the algorithm $\mathcal{A}$ with the given properties to construct an algorithm that runs forever and successively generates all instances of problem P, for which the answer is YES.
- On the other hand, when we have an algorithm $\mathcal{A}'$ running forever and generating all instances, for which the answer is YES, it is easy to construct an algorithm $\mathcal{A}$ with the properties specified above:
  - It reads a number i and stores it in a counter k.
  - Then it will simulate algorithm $\mathcal{A}'$ step by step.
    Always, where $\mathcal{A}'$ generates a next instance of problem P, it decrements the counter k by 1.
  - When the counter k reaches zero, $\mathcal{A}$ will continue in the simulation of $\mathcal{A}'$ as long until the next instance is generated.
  - This instance is written to the output and the simulation ends.

Recursive and Recursively Enumerable Sets

In literature, you can find also the following terminology:
Let us say that A is the set of those instances of problem P where the answer is YES.

- The set A is called **recursive** if there exists an algorithm that for each instance x determines whether x belongs to A, i.e., whether the problem P is decidable.
- The set A is called **recursively enumerable** if there exists an algorithm that write seccessively all these instance, i.e., if it is semidecidable for each instance x whether x belongs to A.

Reduction between Problems

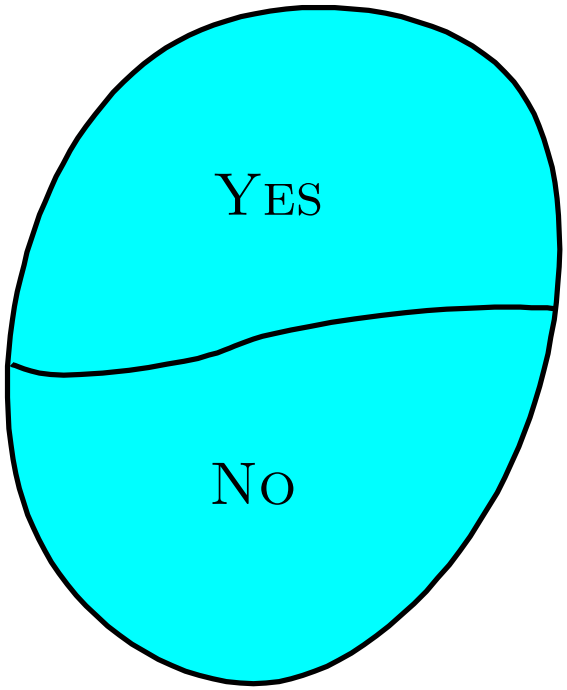
If we have already proved a (decision) problem to be undecidable, we can prove undecidability of other problems by reductions.

Problem P_1 can be **reduced** to problem P_2 if there is an algorithm Alg such that:

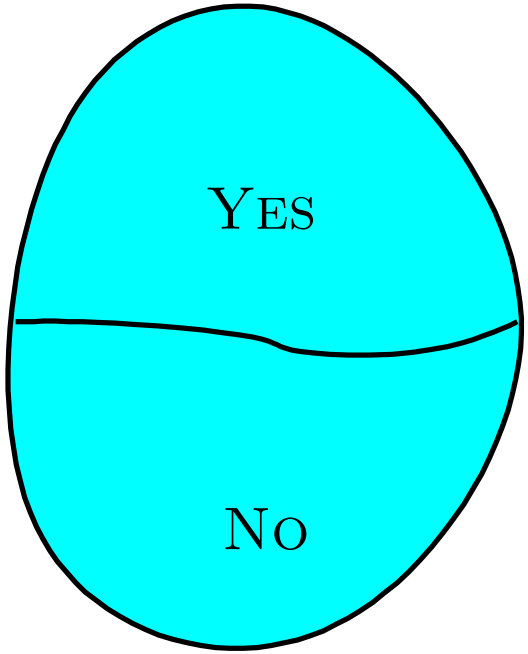
- It can get an arbitrary instance of problem P_1 as an input.
- For an instance of a problem P_1 obtained as an input (let us denote it as w) it produces an instance of a problem P_2 as an output.
- It holds i.e., the answer for the input w of problem P_1 is YES iff the answer for the input $Alg(w)$ of problem P_2 is YES.

Reductions between Problems

Inputs of problem P_1



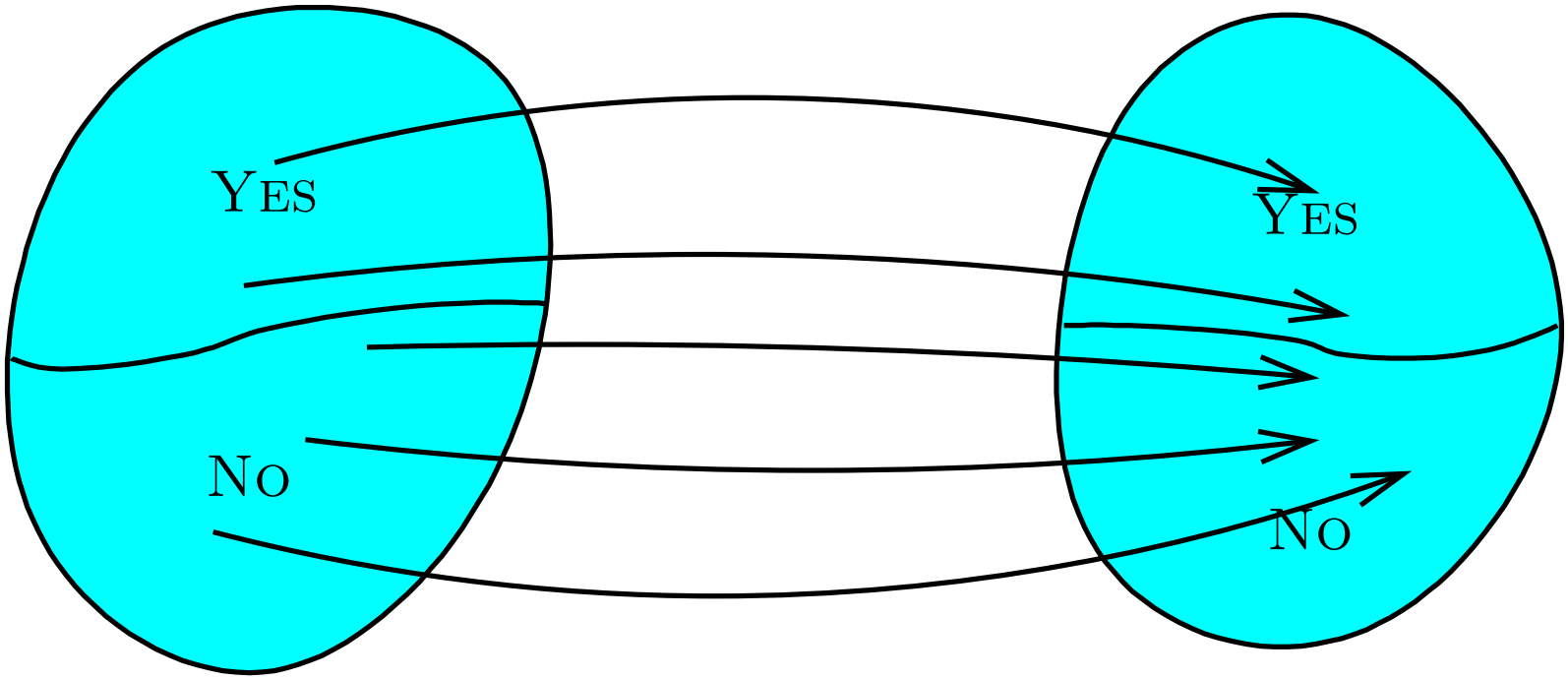
Inputs of problem P_2



Reductions between Problems

Inputs of problem P_1

Inputs of problem P_2



Alg

Reductions between Problems

Let us say there is some reduction Alg from problem P_1 to problem P_2.

If problem P_2 is decidable then problem P_1 is also decidable.

Solution of problem P_1 for an input x:

- Call Alg with x as an input, it returns a value $Alg(x)$.
- Call the algorithm solving problem P_2 with input $Alg(x)$.
- Write the returned value to the output as the result.

It is obvious that if P_1 is undecidable then P_2 cannot be decidable.

Halting Problem

For purposes of proofs, the following version of Halting problem is often used:

Halting problem

Input: A description of a Turing machine $\mathcal{M}$ and a word w.

Question: Does the computation of the machine $\mathcal{M}$ on the word w halt after some finite number of steps?

Halting Problem

This problem is undecidable even in the case, where we assume that the input for machine $\mathcal{M}$ is an empty word ε:

Halting problem (where the input is ε)

Input: A description of a Turing machine $\mathcal{M}$.

Question: Does the computation of the machine $\mathcal{M}$ on the word ε halt after some finite number of steps?

A reduction from the standard Halting problem to this variant is simple.

For each machine $\mathcal{M}$ with input w we construct a machine $\mathcal{M}'$ such that:

- It writes word w to the tape and moves its head to the beginning.
- It starts to behave as $\mathcal{M}$.

Halting Problem

When Halting problem is used in reductions to prove undecidability of some other problems, it can be sometimes useful to assume some other restrictions on this machines, e.g.:

- that it uses just one tape that is infinite only in one direction
- that it uses tape alphabet $\{0, 1\}$
- that after the end of computation, the head is on the same position where it was at the beginning
- that the tape is empty after the end of the computation
- ...

Halting Problem

Halting problem is also often used in reductions in the variant where a Minsky machine is used instead of a Turing machine:

Halting problem (for Minsky machine)

Input: Description of a Minsky machine $\mathcal{M}$.

Question: Does machine $\mathcal{M}$ halt after some number of steps when it starts in a configuration where every counter is set to 0 ?

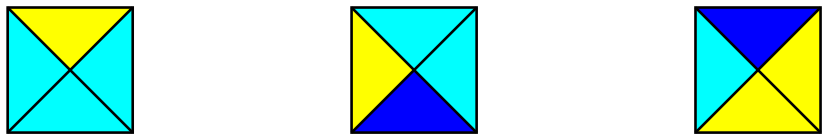
Remark: Also here, it can be useful to use some simplifying assumptions:

- that the machine never tries to decrement a counter whose value is 0
- that there are only two counters
- that all counters contain value 0 after a computation.

Tiling

We will see now another example of an **undecidable** problem.

An input is a set of types of tiles, such as:

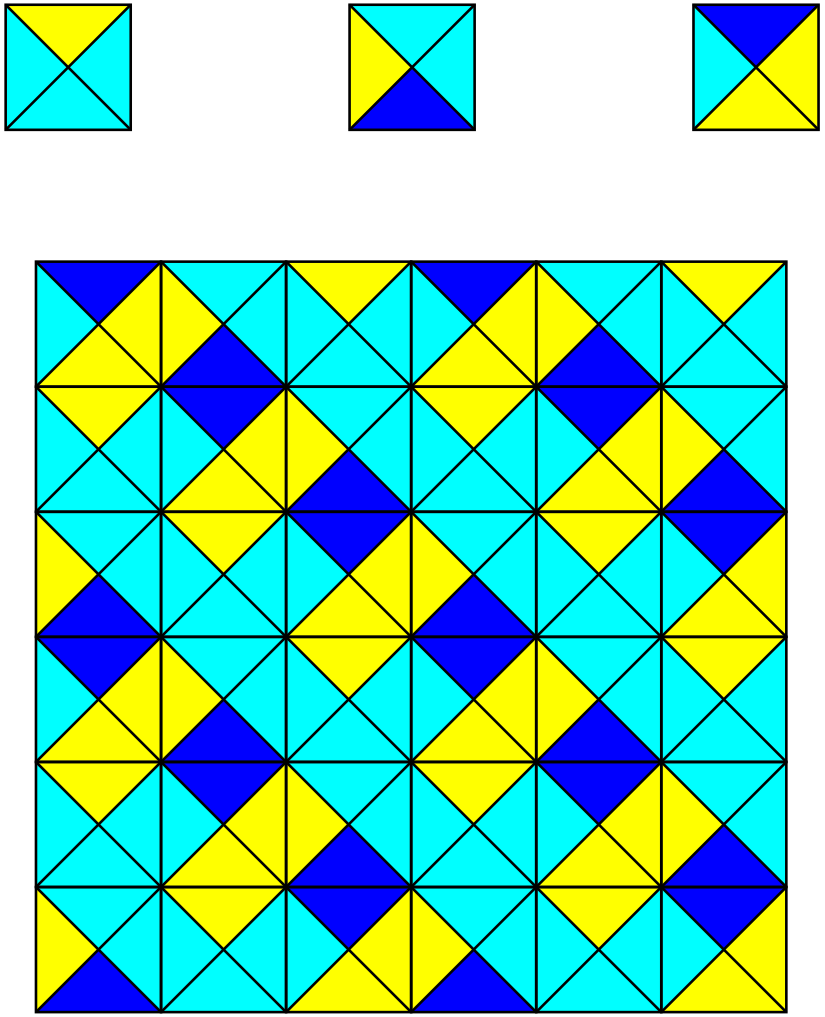


The question is whether it is possible to cover whole infinite plane using the given types of tiles in such a way that the colors of neighboring tiles agree.

Remark: We can assume that we have an infinite number of tiles of all types.

The tiles cannot be rotated.

Tiling



Tiling

This problem can be described more formally as follows:

- Let us assume that C is a finite set of **colors**.
- The set $\{\mathtt{N}, \mathtt{S}, \mathtt{E}, \mathtt{W}\}$ represents four **directions** — north, south, east, west.
- A **type of a tile** is given as an assignment of colors to directions, i.e., as a function $\tau : \{\mathtt{N}, \mathtt{S}, \mathtt{E}, \mathtt{W}\} \to C$.
- Let us assume that we have a set of types of tiles $\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_n\}$.
- **Covering of a plane** with tiles is a function $p : \mathbb{Z} \times \mathbb{Z} \to \mathcal{T}$ satisfying the following two conditions for each $i, j \in \mathbb{Z}$:
  - If $p(i,j) = \tau$ and $p(i+1,j) = \tau'$, then $\tau(\mathtt{E}) = \tau'(\mathtt{W})$.
  - If $p(i,j) = \tau$ and $p(i,j+1) = \tau'$, then $\tau(\mathtt{N}) = \tau'(\mathtt{S})$.

Tiling

Consider the following variant of the problem:

Input: A set of types of tiles $\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_n\}$ and an initial tile $\tau_0 \in \mathcal{T}$.

Question: Is there some covering of a plane p with tiles from the set $\mathcal{T}$ such that $p(0,0) = \tau_0$?

So in this variant, one of the types of tiles is distinguished as a special initial tile, and the question is whether it is possible to cover whole plane in a way where this special tile is used.
(The other types of tiles can, but need not be, used.)

Tiling

Undecidability of this problem (resp. of the complement problem of this problem) can be proved for example using a reduction from Halting problem in the following variant :

Input: Turing machine $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, q_0, \{q_f\})$.

Question: Does the Turing machine $\mathcal{M}$ halts after a finite number of steps if it obtains the empty word ε as an input?

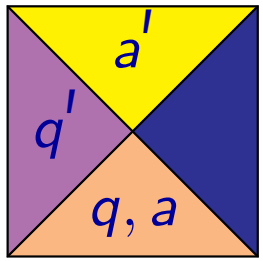
We will describe an algorithm that:

- It obtains a Turing machine $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, q_0, \{q_f\})$ as an input.
- It constructs (and outputs) a set of types of tiles $\mathcal{T}$ with a special distinguished tile $\tau_0 \in \mathcal{T}$.
- The following will hold: It will be possible to cover whole plane using tiles from $\mathcal{T}$ so that on position $(0, 0)$ is the tile τ_0 iff the machine $\mathcal{M}$ on input ε never halts (i.e., its computation is infinite).

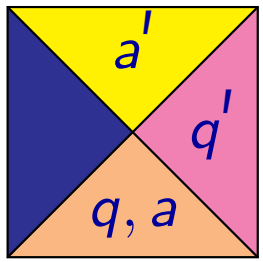
Tiling

The algorithm constructs the tiles for the given machine $\mathcal{M}$ as follows (in the following description of tiles, the names of elements from set Q, Γ a $(Q \times \Gamma)$ are used in addition to colors — replacing them with additional colors is straightforward):

- For each $q, q' \in Q - F$ a $a, a' \in \Gamma$, where $\delta(q, a) = (q', a', -1)$, we add the following type of a tile:

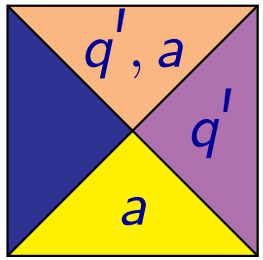


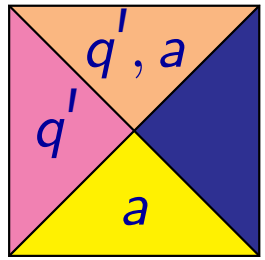
- For each $q, q' \in Q - F$ a $a, a' \in \Gamma$, where $\delta(q, a) = (q', a', +1)$, we add the following type of a tile:



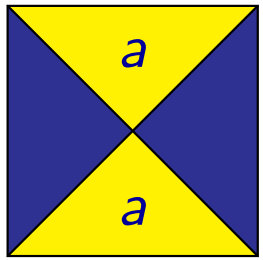
Tiling

- For each $q' \in Q$ a $a \in \Gamma$ we add the following two types of tiles:



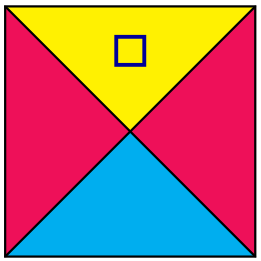


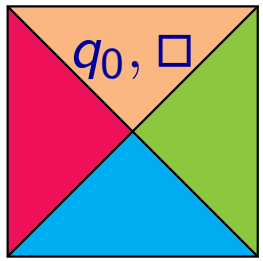
- For each $a \in \Gamma$ we add the following type of a tile:

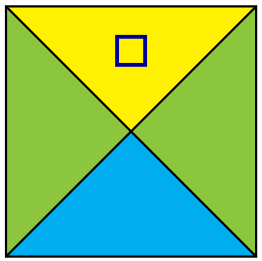


Tiling

- We add the following three types of tiles
  (the tile in the middle will be distinguished as the **initial** tile τ_0):

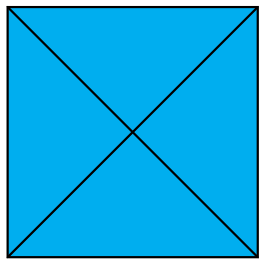






  (Here, the symbol $\square \in \Gamma$ represents the **blank** symbol of the Turing machine $\mathcal{M}$.)

- Finally, we add the following "empty" tile:



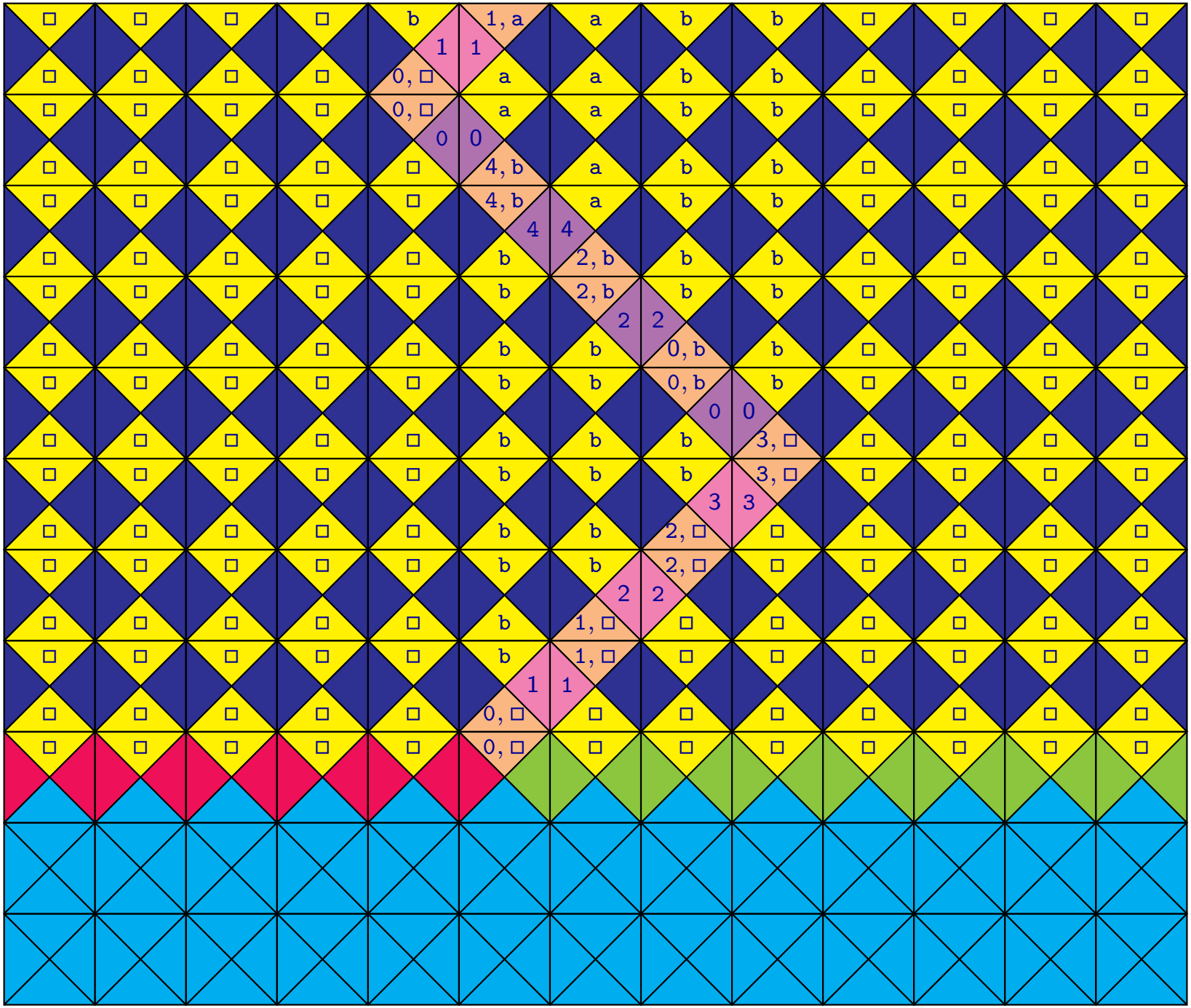
Tiling

Let us say that the computation of the Turing machine $\mathcal{M}$ over the empty word ε consists of the sequence of configurations

$$\alpha_0, \alpha_1, \alpha_2, \ldots$$

It is not hard to check that any possible covering of a plane using these tiles has the following properties (when the position $(0, 0)$ contains the tile τ_0):

- The colors on the upper sides of tiles in row 0 correspond to the initial configuration α_0.
- This enforces that the colors on the upper sides of tiles in row 1 must correspond to configuration α_1.
- This enforces that the colors on the upper sides of tiles in row 2 must correspond to configuration α_2.
- ...


□ □ □ □ b 1, a a b b □ □ □ □
1 1
□ □ □ □ 0, □ a a b b □ □ □ □
□ □ □ □ 0, □ a a b b □ □ □ □
0 0
□ □ □ □ □ 4, b a b b □ □ □ □
□ □ □ □ □ 4, b a b b □ □ □ □
4 4
□ □ □ □ □ b 2, b b b □ □ □ □
□ □ □ □ □ b 2, b b b □ □ □ □
2 2
□ □ □ □ □ b b 0, b b □ □ □ □
□ □ □ □ □ b b 0, b b □ □ □ □
0 0
□ □ □ □ □ b b b 3, □ □ □ □ □
□ □ □ □ □ b b b 3, □ □ □ □ □
3 3
□ □ □ □ □ b b 2, □ □ □ □ □ □
□ □ □ □ □ b b 2, □ □ □ □ □ □
2 2
□ □ □ □ □ b 1, □ □ □ □ □ □ □
□ □ □ □ □ b 1, □ □ □ □ □ □ □
1 1
□ □ □ □ □ 0, □ □ □ □ □ □ □ □
□ □ □ □ □ 0, □ □ □ □ □ □ □ □

Tiling

So in general, for each i, where $i \geq 0$, it must hold that:

- The colors on the upper sides of tile in row i correspond to configuration α_i.

But this is only possible if the computation of machine $\mathcal{M}$ over input ε is infinite.

In the case, when a final configuration (e.g., in row t) is reached, the next row $(t + 1)$ can not be added.

If the computation is infinite:

- it is possible to fill in all rows i, kde $i \geq 0$.

The rows -1, -2, -3, ... can be always filled with "empty" tiles.

Tiling

So we see that the following holds:

- If the computation of $\mathcal{M}$ over ε is infinite, then it is possible to cover whole plane with the corresponding set of tiles.
- If the computation of $\mathcal{M}$ over ε halts after a finite number of steps, the plane cannot be covered.

If there would exist an algorithm that would be able to determine for an arbitrary set of tiles (and a given initial tile) whether whole plane can be filled, them this algorithm could be used also for solving Halting problem .

But this is not possible (we already know that there is no algorithm that would solve Halting problem), and so there cannot exist such algorithm.

Tiling

Remark: It is possible to show that the problem of covering whole plane is undecidable even in the variant where no "initial" tile is specified:

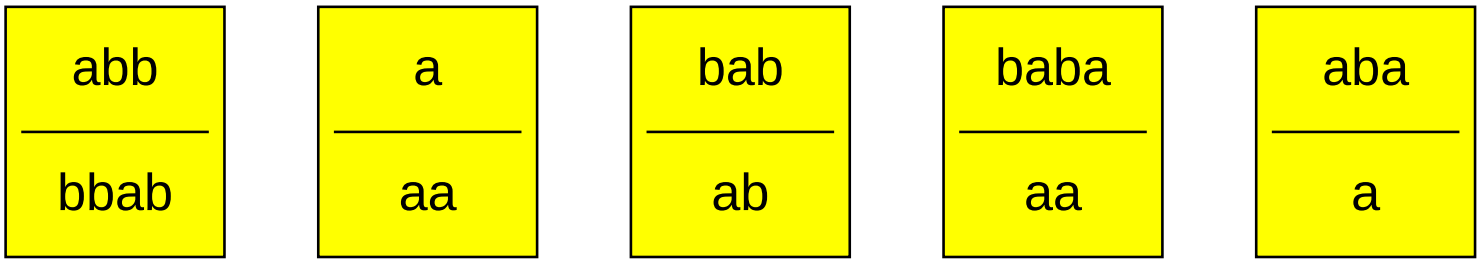
Input: A set of tupes of tiles $\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_n\}$.

Question: Is there some covering of whole plane with tiles from the set $\mathcal{T}$?

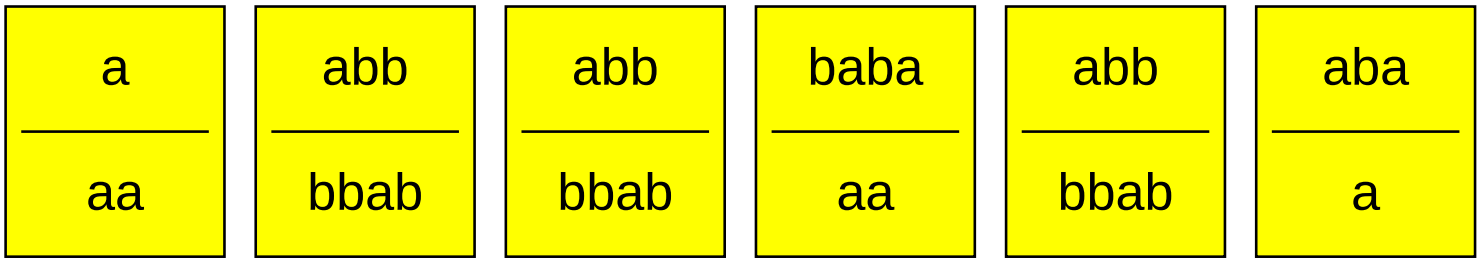
The proof is more technically complicated but is based on similar ideas as presented above — i.e., encoding of computations of the Turing machine in such a way that covering whole plane is possible exactly in those cases where the computation of the given machine are infinite.

Post correspondence problem

An input is a set of types of cards, such as:



The question is whether it is possible to construct from the given types of cards a non-empty finite sequence such that the concatenations of the words in the upper row and in the lower row are the same. Every type of a card can be used repeatedly.



In the upper and in the lower row we obtained the word aabbabbbabaabbaba.

Post correspondence problem

This problem is called **Post Correspondence Problem** (**PCP**):

Post correspondence problem (PCP)

Input: Sequences of words $u_1, u_2, \ldots, u_n$ and $v_1, v_2, \ldots, v_n$ over some alphabet Σ.

Question: Is there some sequence $i_1, i_2, \ldots, i_m$, where $m \geq 1$, where for each i_j holds $1 \leq i_j \leq n$, and where

$$u_{i_1} u_{i_2} \cdots u_{i_m} = v_{i_1} v_{i_2} \cdots v_{i_m} \, ?$$

Post correspondence problem

Undecidability of this problem can be shown by reduction from Halting problem (HP).

In the describtion of this reduction, it is useful to use as an intermediate step the following variant of Post correspondence problem where one of the cards is distiguished as initial:

Inicial Post correspondence problem (IPCP)

Input: Sequences of words $u_1, u_2, \ldots, u_n$ and $v_1, v_2, \ldots, v_n$ over some alphabet Σ.

Question: Is there a sequence $i_1, i_2, \ldots, i_m$, where $m \geq 1$, where for each i_j holds $1 \leq i_j \leq n$, and where

$$u_{i_1} u_{i_2} \cdots u_{i_m} = v_{i_1} v_{i_2} \cdots v_{i_m}$$

and where moreover is $i_1 = 1$?

Post correspondence problem

The reduction from HP to IPCP:

- The algorithm obtains a description of a Turing machine $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, q_0, F)$ and its input $w = a_1 a_2 \dots a_n$ as an input.
- The algorithm creates an instance of IPCP, i.e., a set of types of cards.
- The create instance of IPCP will have a solution iff machine $\mathcal{M}$ over input w halts.

  The solutions will be such that the common word created in upper and lower line will be basically a description of the computation of machine $\mathcal{M}$ over word w:

  - it will be a sequence of individual configurations written as words
  - these configurations will be separated by a special symbol #

Post correspondence problem

- The first card, distinguished as the initial card, is as follows:

$$\left[\frac{\#}{\#q_0 a_1 a_2 \cdots a_n \#}\right]$$

- For each $q, q' \in Q$ and $a, a' \in \Gamma$, where $\delta(q, a) = (q', a', +1)$, the following card is added:

$$\left[\frac{qa}{a'q'}\right]$$

- For each $q, q' \in Q$ and $a, a', b \in \Gamma$, where $\delta(q, a) = (q', a', -1)$, the following card is added:

$$\left[\frac{bqa}{q'ba'}\right]$$

Post correspondence problem

- For each $a \in \Gamma$, the following card is added:

$$\left[\begin{array}{c} a \\ \hline a \end{array}\right]$$

- The following cards are added:

$$\left[\begin{array}{c} \# \\ \hline \# \end{array}\right] \qquad \left[\begin{array}{c} \# \\ \hline \square\# \end{array}\right]$$

- For each $a \in \Gamma$ and $q_f \in F$, the following cards are added:

$$\left[\begin{array}{c} aq_f \\ \hline q_f \end{array}\right] \qquad \left[\begin{array}{c} q_f a \\ \hline q_f \end{array}\right] \qquad \left[\begin{array}{c} q_f \#\# \\ \hline \# \end{array}\right]$$

Post correspondence problem

The reduction from IPCP to PCP can be done as follows:

- Instead of each card of the form

$$\left[\frac{a_1 a_2 \cdots a_k}{b_1 b_2 \cdots b_\ell}\right]$$

  the card of the following form is added

$$\left[\frac{*a_1 * a_2 * \cdots * a_k}{b_1 * b_2 * \cdots * b_\ell *}\right]$$

- For the initial card, the card of the following form is also added

$$\left[\frac{*a_1 * a_2 * \cdots * a_k}{*b_1 * b_2 * \cdots * b_\ell *}\right]$$

Post correspondence problem

- The following card is added

$$\left[\frac{*\diamond}{\diamond}\right]$$

Remark: It is assumed that symbols $*$ and $\diamond$ are some new special symbols that do not occur in the original instance of IPCP.

Post correspondence problem

Undecidability of several other problems dealing with context-free grammars can be proved by reductions from the previous problem:

Problem

Input: Context-free grammars $\mathcal{G}_1$ and $\mathcal{G}_2$.

Question: Is $\mathcal{L}(\mathcal{G}_1) \cap \mathcal{L}(\mathcal{G}_2) = \varnothing$?

Problem

Input: A context-free grammar $\mathcal{G}$.

Question: Is $\mathcal{G}$ ambiguous?

Equivalence of context-free grammars

Also the following two problems concerning context-free grammars are undecidable:

Problem

Input: Context-free grammars $\mathcal{G}_1$ and $\mathcal{G}_2$.

Question: Is $\mathcal{L}(\mathcal{G}_1) = \mathcal{L}(\mathcal{G}_2)$?

Problem

Input: A context-free grammar generating a language over an alphabet Σ.

Question: Is $\mathcal{L}(\mathcal{G}) = \Sigma^*$?

Equivalence of context-free grammars

The proof of undecidability can be done again by a reduction from HP.

For a given Turing machine $\mathcal{M}$ and its input w, a context-free grammar $\mathcal{G}$ is constructed such that:

- if machines $\mathcal{M}$ halts on w, the language $\mathcal{L}(\mathcal{G})$ will contain all words, except a word that is a description of the computation of machine $\mathcal{M}$ over word w in the form of a sequence of configurations
- if machine $\mathcal{M}$ halts on w, the language $\mathcal{L}(\mathcal{G})$ will contain all words

Equivalence of context-free grammars

So the grammar $\mathcal{G}$ will generate exactly those words that **are not** descriptions of the computations of machine $\mathcal{M}$ over word w, i.e.:

- they do not have a form of a sequence of configurations, or
- they do not start with the initial configuration
- they do not end with a final configuration
- there exists a consucutive pair of configurations that does not correspond the step that would be performed by $\mathcal{M}$

To allow to describe the fourth of the above mentioned conditions using a context-free grammar, the following idea is used:

- configurations on even positions are written as usual from left to right
- configuration on odd positions are written in reverse, i.e., from right to left

Arithmetic on natural numbers

Problem

Input: A closed formula of the first order predicate logic where the only predicate symbol is $=$, the only function symbols are $+$ and $\cdot$, and the only constant symbols are 0 and 1.

Question: Is the given formula true in the domain of natural numbers (using the natural interpretation of all function and predicate symbols)?

An example of an input:

$$\forall x \exists y \forall z((x \cdot y = z) \wedge (y + 1 = x))$$

Remark: There is a close connection with Gödel's incompleteness theorem.

Arithmetic on natural numbers

We will show that this problem is **undecidable**.

For this proof, a reduction from the halting problem for Minsky machine will be used:

Input: Description of a Minsky machine $\mathcal{M}$.

Question: Does the machine $\mathcal{M}$ halts after some finite number of steps when it starts in a configuration where all counters have value 0 ?

Arithmetic on natural numbers

We will describe an algorithm that:

- It obtains a description of a Minsky machine $\mathcal{M}$ as an input.
- For this machine $\mathcal{M}$ constructs a formula φ, and produces this formula as an output.
- The following will hold for this formula:

  The formula φ will be true (in the standard interpretation on natural numbers) iff machine $\mathcal{M}$ halts after some finite number of steps.

Remark: Formula φ will be built in several steps.

It will be built from simpler formulas.

Arithmetic on natural numbers

Let Var be an countably infinite set of all **variables** that can occur in formulas — $Var = \{x, y, z, \ldots\}$

Terms are defined as follows:

- Every variable x from set Var is a well-formed term.
- Constants 0 and 1 are well-formed terms.
- If t_1 and t_2 are well-formed terms, then also i $t_1 + t_2$ and $t_1 \cdot t_2$ are well-formed terms.

Formulas are defined as follows:

- If t_1 and t_2 are well-formed terms, then $t_1 = t_2$ is a well-formed formula.
- If φ_1 and φ_2 are well-formed formulas, then also $\neg\varphi_1$, $\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$, $\varphi_1 \rightarrow \varphi_2$, and $\varphi_1 \leftrightarrow \varphi_2$ are well-formed formulas.
- If φ is a well-formed formula and x is a variable from the set Var, then also $\forall x.\varphi$ and $\exists x.\varphi$ are well-formed formulas.

Arithmetic on natural numbers

Remarks:

- Formulas will be interpreted over the set of **natural numbers** $\mathbb{N} = \{0, 1, 2, \ldots\}$.
- Constants 2, 3, 4, $\ldots$ can be viewed as shorthands for $1 + 1$, $1 + 1 + 1$, $1 + 1 + 1 + 1$, $\ldots$
- Notation $t_1 \leq t_2$ is a shorthand for

  $$\exists x.(t_1 + x = t_2)$$

  (It is assumed that x does not occur in t_1 nor in t_2.)
- Similarly, notation $t_1 < t_2$ is a shorthand for

  $$\exists x.(t_1 + x + 1 = t_2)$$

- In a similar way, we can define also $t_1 \geq t_2$ and $t_1 > t_2$.

Arithmetic on natural numbers

- $\textsc{Divides}(x, y)$ — x is a divisor of y:

$$\exists k.(x \cdot k = y)$$

- $\textsc{Prime}(p)$ — p is a prime:

$$p > 1 \ \wedge \ \forall x.(\textsc{Divides}(x, p) \ \rightarrow \ (x = 1) \vee (x = p))$$

- $\textsc{Prime-Power}(p, x)$ — x is a power of a prime p (i.e., there exists $i \in \mathbb{N}$ such that $x = p^i$):

$$\textsc{Prime}(p) \ \wedge \ (x \geq 1) \wedge \forall y.(\textsc{Divides}(y, x) \ \wedge \ \textsc{Prime}(y) \ \rightarrow \ (y = p))$$

Arithmetic on natural numbers

Let us say we have a Minsky machine $\mathcal{M}$:

- The set of **states** of the control unit of machine $\mathcal{M}$ is $S = \{0, 1, \ldots, s\}$.
- The initial state is 0
- The final state s.
- The machine has r counters, denoted x_1, x_2, $\ldots$, x_r.

A **configuration** of machine $\mathcal{M}$ can be described as an $(r+1)$-tuple of natural numbers

$$(q, v_1, \ldots, v_r)$$

where q represents the current state of the control unit, and $v_1, \ldots, v_r$ are values of the counters $x_1, \ldots, x_r$.

Arithmetic on natural numbers

Let us say for concreteness that machine $\mathcal{M}$ will use for example 3 counters, i.e., $r = 3$.

It is easy to construct formulas that characterize **initial** and **final** configurations:

- $\textsc{Initial-Conf}(q, v_1, v_2, v_3)$ — it is an initial configuration:

$$(q = 0) \wedge (v_1 = 0) \wedge (v_2 = 0) \wedge (v_3 = 0)$$

- $\textsc{Final-Conf}(q, v_1, v_2, v_3)$ — it is a final configuration:

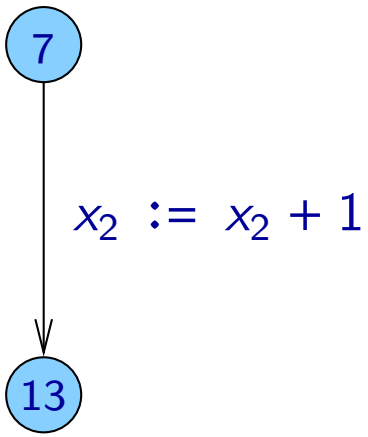
$$q = s$$

Arithmetic on natural numbers

Similarly, it is not too difficult to construct a formula for the given Minsky machine that characterizes when the machine can go from one configuration to another in one step:

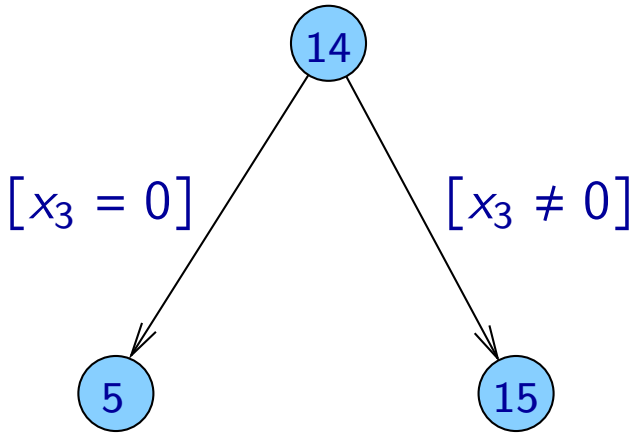
- $\textsc{Step}(q, v_1, v_2, v_3, q', v_1', v_2', v_3')$ — the machine $\mathcal{M}$ can go in one step from configuration (q, v_1, v_2, v_3) to configuration (q', v_1', v_2', v_3')

  This is a disjunction of many formulas where each of these formulas describes the behaviour of one instruction of the machine $\mathcal{M}$, e.g.,



$$(q = 7) \wedge (q' = 13) \wedge$$
$$(v_1' = v_1) \wedge (v_2' = v_2 + 1) \wedge (v_3' = v_3)$$

Arithmetic on natural numbers



$$
\begin{aligned}
&(q = 14) \wedge \\
&(((v_3 = 0) \wedge (q' = 5)) \vee \\
&\quad ((v_3 > 0) \wedge (q' = 15))) \wedge \\
&(v_1' = v_1) \wedge (v_2' = v_2) \wedge (v_3' = v_3)
\end{aligned}
$$

Arithmetic on natural numbers

A computation of the machine $\mathcal{M}$ can be described as a sequence of configurations

$$\alpha_0, \alpha_1, \alpha_2, \ldots$$

This sequence can be represented as several separate sequences:

- a sequence of states of the control unit
- a sequence of values of counter x_1
- a sequence of values of counter x_2

$$\vdots$$

- a sequence of values of counter x_r

In general, any finite sequence of natural numbers can be encoded as one natural number.

So if the machine $\mathcal{M}$ halts, each of sequences above can be represented as one natural number.

Arithmetic on natural numbers

If we have for example a sequence of natural numbers

$$a_0, a_1, \ldots, a_t$$

it can be encoded as number

$$a_t \cdot b^t + a_{t-1} \cdot b^{t-1} + \cdots + a_2 \cdot b^2 + a_1 \cdot b^1 + a_0 \cdot b^0$$

where b is a big enough natural number, i.e., a number where for all a_i (where $0 \leq i \leq t$) is

$$0 \leq a_i < b$$

So the sequence $a_0, a_1, \ldots, a_t$ can be represented by individual digits in the representation of a number written in base b.

Arithmetic on natural numbers

- If A is the number encoding a sequence $a_0, a_1, \ldots, a_t$ in the way described above, the value a_i can be expressed as follows:

$$\exists u.\exists v.((A = (u \cdot b + a_i) \cdot b^i + v) \ \wedge \ (v < b^i))$$

  But there is a problem how to express b^i.

- We can use some big enough prime p as the base b.

- In fact, we need not work directly with indexes i (where $0 \leq i \leq t$).

  Instead, we can work with powers of prime p, i.e., with values $p^0, p^1, p^2, p^3, \ldots$

  So, instead of i, we will use value p^i.

Arithmetic on natural numbers

So, let us assume that p is a prime, and that a sequence $a_0, a_1, \ldots, a_t$ is encoded as number

$$A = a_t \cdot p^t + a_{t-1} \cdot p^{t-1} + \cdots + a_2 \cdot p^2 + a_1 \cdot p^1 + a_0 \cdot p^0$$

(It is assumed that for each i we have $0 \leq a_i < p$.)

- $\textsc{Digit}(p, d, A, c)$ — for some $i \in \mathbb{N}$ it holds that $d = p^i$ and in the sequence $a_0, a_1, \ldots, a_t$, encoded as number A, is $a_i = c$:

$$\textsc{Prime-Power}(p, d) \ \wedge \ (c < p) \ \wedge \ \exists u.\exists v.((A = (u \cdot p + c) \cdot d + v) \ \wedge \ (v < d))$$

Arithmetic on natural numbers

This means that the computation of machine $\mathcal{M}$ can be fully described by the values of the following variables (for concreteness, we assume that there are three counters, i.e., $r = 3$):

- p — a big enough prime (bigger than the number of states s and bigger than the value of any counter during the computation)
- T — the value p^t where t is the total number of steps performed by the machine $\mathcal{M}$ during the computation
- Q — the number encoding the sequence of states of the control unit
- X_1 — the number encoding the sequence of values of counter x_1
- X_2 — the number encoding the sequence of values of counter x_2
- X_3 — the number encoding the sequence of values of counter x_3

Arithmetic on natural numbers

- $\textsc{Conf}(p, d, Q, X_1, X_2, X_3, q, v_1, v_2, v_3)$:
  — configuration α_i, where $d = p^i$, is equal to (q, v_1, v_2, v_3)

$$\textsc{Digit}(p, d, Q, q) \wedge \textsc{Digit}(p, d, X_1, v_1) \wedge \textsc{Digit}(p, d, X_2, v_2) \wedge \textsc{Digit}(p, d, X_3, v_3)$$

- $\textsc{Check-Initial}(p, Q, X_1, X_2, X_3)$:
  — checking that the computation starts with the initial configuration

$$\exists q.\exists v_1.\exists v_2.\exists v_3.(\textsc{Conf}(p, 1, Q, X_1, X_2, X_3, q, v_1, v_2, v_3) \wedge \textsc{Initial-Conf}(q, v_1, v_2, v_3))$$

- $\textsc{Check-Final}(p, T, Q, X_1, X_2, X_3)$:
  — checking that the computation ends with a final configuration

$$\exists q.\exists v_1.\exists v_2.\exists v_3.(\textsc{Conf}(p, T, Q, X_1, X_2, X_3, q, v_1, v_2, v_3) \wedge \textsc{Final-Conf}(q, v_1, v_2, v_3))$$

Arithmetic on natural numbers

- $\textsc{Check-One-Step}(p, d, Q, X_1, X_2, X_3)$:
  — checking that in the given computation, the machine correctly goes from the configuration α_i to the configuration α_{i+1}, where $d = p^i$

$$\begin{aligned}
&\exists q.\exists v_1.\exists v_2.\exists v_3.\exists q'.\exists v_1'.\exists v_2'.\exists v_3'.(\\
&\qquad \textsc{Conf}(p, d, Q, X_1, X_2, X_3, q, v_1, v_2, v_3) \wedge \\
&\qquad \textsc{Conf}(p, d \cdot p, Q, X_1, X_2, X_3, q', v_1', v_2', v_3') \wedge \\
&\qquad \textsc{Step}(q, v_1, v_2, v_3, q', v_1', v_2', v_3'))
\end{aligned}$$

- $\textsc{Check-All-Steps}(p, T, Q, X_1, X_2, X_3)$:
  — checking that all steps are correct

$$\begin{aligned}
&\forall d.((d < T) \wedge \textsc{Prime-Power}(p, d) \rightarrow \\
&\qquad\qquad \textsc{Check-One-Step}(p, d, Q, X_1, X_2, X_3))
\end{aligned}$$

Arithmetic on natural numbers

- MACHINE-HALTS:
  — checking that there exists a finite computation of the given machine

$$\begin{aligned}
&\exists p.\exists T.\exists Q.\exists X_1.\exists X_2.\exists X_3.(\\
&\qquad \textsc{Prime}(p) \wedge \\
&\qquad \textsc{Prime-Power}(p, T) \wedge \\
&\qquad \textsc{Check-Initial}(p, Q, X_1, X_2, X_3) \wedge \\
&\qquad \textsc{Check-All-Steps}(p, T, Q, X_1, X_2, X_3) \wedge \\
&\qquad \textsc{Check-Final}(p, T, Q, X_1, X_2, X_3))
\end{aligned}$$

It is not hard to check that this formula is true iff the computation of machine $\mathcal{M}$ halts after some finite number of steps.

Arithmetic on natural numbers

So if there would exist an algorithm that could find out for each such formula whether it is true, we would have an algorithm solving Halting problem. But this is not possible.

Remarks:

- It is interesting that an analogous problem, where real numbers are considered instead of natural numbers, is decidable (but the algorithm for it and the proof of its correctness are quite nontrivial).

- Also when we consider natural numbers or integers and the same formulas as in the previous case but with the restriction that it is not allowed to use the multiplication function symbol $\cdot$, the problem is algorithmically decidable.

Arithmetic on natural numbers

If the function symbol $\cdot$ can be used then even the very restricted case is undecidable:

Hilbert's tenth problem

Input: A polynomial $f(x_1, x_2, \ldots, x_n)$ constructed from variables $x_1, x_2, \ldots, x_n$ and integer constants.

Question: Are there some natural numbers $x_1, x_2, \ldots, x_n$ such that $f(x_1, x_2, \ldots, x_n) = 0$?

An example of an input: $5x^2y - 8yz + 3z^2 - 15$

I.e., the question is whether

$$\exists x \exists y \exists z (5 \cdot x \cdot x \cdot y + (-8) \cdot y \cdot z + 3 \cdot z \cdot z + (-15) = 0)$$

holds in the domain of natural numbers.

Other Undecidable Problems

Also the following problem is algorithmically undecidable:

Problem

Input: A closed formula φ of the first-order predicate logic.

Question: Is $\models \varphi$?

Remark: Notation $\models \varphi$ denotes that formula φ is logically valid, i.e., it is true in all interpretations.

Other Undecidable Problems

By reductions from the Halting problem we can show undecidability of many other problems dealing with a behaviour of programs:

- Does a given program produce the output YES for some input?
- Does a given program halt for an arbitrary input?
- Do two given programs produce the same outputs for the same inputs?
- ...

Rice's Theorem

Let P be an arbitrary prorperty of Turing machines.

The property P is:

- **nontrivial** — if there exists at least one machine that has the property P, and at least one machine that does not have the property P
- **input-output** — if in every pair of machines that halt on the same inputs and that give the same outputs for the same inputs, either both machines have the property P or both do not have it

Theorem

Every problem of the form

Input: A Turing machine $\mathcal{M}$.

Question: Does machine $\mathcal{M}$ have property P?

where P is a nontrivial input-output property, is undecidable.

Rice's Theorem

Proof:

- The proof is done by a reduction from Halting problem.
- This is not a single reduction but a general **schema** describing how to construct the corresponding reduction from Halting problem to one of two following problems for **each** particular nontrivial input-output property P:
  - The question whether the given Turing machine has the given property P.
  - The question whether the given Turing machine does not have the given property P.

Rice's Theorem

The algorithm performing this reduction:

- It obtains an instance $(\mathcal{M}, w)$ of Halting problem as an input (kde $\mathcal{M}$ je Turingův stroj a w jeho vstup).
- It constructs a Turing machine $\mathcal{M}$' for the given pair.

One of two following possibilities will be always true for the reduction:

- Machine $\mathcal{M}'$ will have the property P iff machine $\mathcal{M}$ halts over input w.
- Machine $\mathcal{M}'$ will have the property P iff machine $\mathcal{M}$ does not halt over input w.

Which of these possiblities will hold depends on the property P.

Rice's Theorem

Let $\mathcal{M}_0$ be the Turing machine that never halts for any input instance and never produces any output, i.e., for each input, it always runs forever.

There are two possibilities:

- The machine $\mathcal{M}_0$ has the property P.
- The machine $\mathcal{M}_0$ does not have property P.

We will concentrate on the second possibility, i.e., when $\mathcal{M}_0$ does not have the property P.
(The proof for the first possibility, i.e., when $\mathcal{M}_0$ has the property P, is similar.)

Because the property P is nontrivial, there must exist at least one machine $\mathcal{M}_1$ that has the property P.

Rice's Theorem

The algorithm performing the reduction for the given instance $(\mathcal{M}, w)$ of Halting problem, which it obtains as an input, constructs a Turing machine $\mathcal{M}'$ that will behave as follows:

- It will leave its input w' on the tape intact and will use the remaining "empty" part of the tape for a simulation of the computation of machine $\mathcal{M}$ on input w.
- If this simulation of the computation of machine $\mathcal{M}$ on input w halts, then:
  - it clears all cells on the tape used in this simulation (i.e., it rewrites them with blank symbols),
  - it goes with its head to the beginning of word w',
  - it will start to bahave as machine $\mathcal{M}_1$.

Rice's Theorem

It is obvious that:

- If the computation of the machine $\mathcal{M}$ on the input w halts then:

  The machine $\mathcal{M}'$ behaves from the point of view of input and output exactly as the machine $\mathcal{M}_1$.

  So the machine $\mathcal{M}'$ has the property P
  (since it is an input-output property and the machine $\mathcal{M}_1$ has this property).

- If the computation of the machine $\mathcal{M}$ on the input w does not halt, then:

  The simulation of the behaviour of the machine $\mathcal{M}$ on the input w performed by the machine $\mathcal{M}'$ never halts.

  So the machine $\mathcal{M}_0$ behaves exactly as the machine $\mathcal{M}_0$ from the point of view of input and output.

  So the machine $\mathcal{M}'$ does not have the property P in this case
  (since it is an input-output property and the machine $\mathcal{M}_0$ does not have this property).

Rice's Theorem

The case when the machine $\mathcal{M}_0$ has the property P is similar:

- As $\mathcal{M}_1$, we can choose some machine that does not have the property P.
- If $\mathcal{M}$ halts on w, then $\mathcal{M}'$ behaves as $\mathcal{M}_1$ and so it does not have the property P.
- If $\mathcal{M}$ does not halt on w, then $\mathcal{M}'$ behaves as $\mathcal{M}_0$ and so it has the property P.