An application of Snakes

First application of Snakes: a two-dimensional cellular automaton **Snake XOR** that is injective on periodic configurations but is not injective on all configurations.

The **Snake XOR** CA confirms that in 2D

$$G \text{ injective } \not\Longleftarrow G_P \text{ injective.}$$

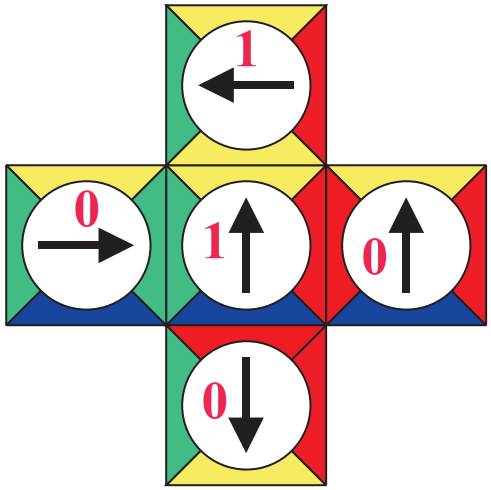
The state set of the CA is

$$S = \textsc{Snakes} \times \{\mathbf{0}, \mathbf{1}\}.$$

(Each snake tile is attached a red bit.)

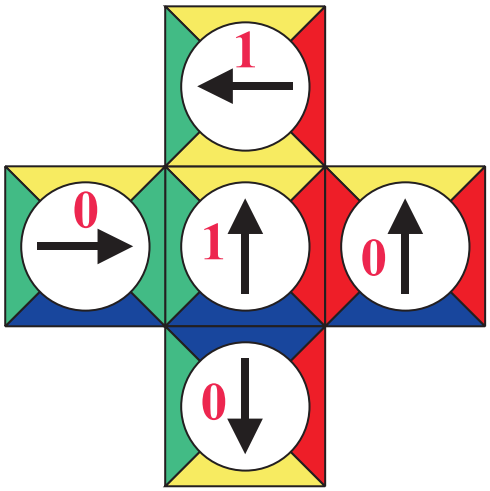
The local rule checks whether the tiling is valid at the cell:

- If there is a tiling error, no change in the state.



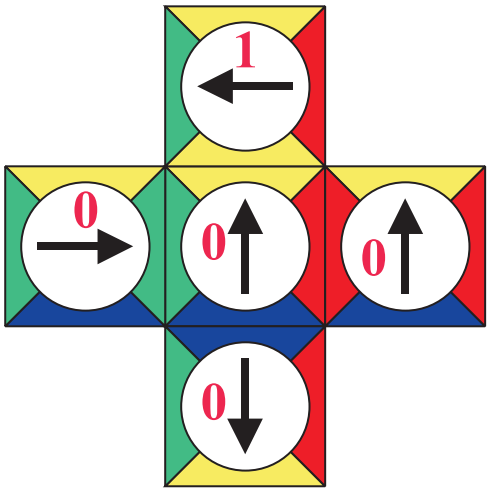
The local rule checks whether the tiling is valid at the cell:

- If there is a tiling error, no change in the state.
- If the tiling is valid, the cell is **active**: the bit of the neighbor next on the path is XOR'ed to the bit of the cell.



The local rule checks whether the tiling is valid at the cell:

- If there is a tiling error, no change in the state.
- If the tiling is valid, the cell is **active**: the bit of the neighbor next on the path is XOR'ed to the bit of the cell.



Snake XOR is not injective:

The following two configurations c_0 and c_1 have the same successor: The SNAKES layers in c_0 and c_1 form the same valid tiling of the plane. In c_0 all bits are set to 0, and in c_1 all bits are 1.

All cells are active because the tilings are correct. This means that all bits in both configurations become 0. So the two configurations become identical. The CA is not injective.

Snake XOR is injective on periodic configurations:

Suppose there are different periodic configurations c and d with the same successor. Since only bits may change, c and d must have identical SNAKES tiles everywhere. So they must have different bits 0 and 1 in some position $\vec{p}_1 \in \mathbb{Z}^2$.

Snake XOR is injective on periodic configurations:

Suppose there are different periodic configurations c and d with the same successor. Since only bits may change, c and d must have identical SNAKES tiles everywhere. So they must have different bits 0 and 1 in some position $\vec{p}_1 \in \mathbb{Z}^2$.

Because c and d have identical successors:

- The cell in position $\vec{p}_1$ must be active, that is, the SNAKES tiling is valid in position $\vec{p}_1$.
- The bits stored in the next position $\vec{p}_2$ (indicated by the direction) are different in c and d.

Snake XOR is injective on periodic configurations:

Suppose there are different periodic configurations c and d with the same successor. Since only bits may change, c and d must have identical SNAKES tiles everywhere. So they must have different bits 0 and 1 in some position $\vec{p}_1 \in \mathbb{Z}^2$.

Because c and d have identical successors:

- The cell in position $\vec{p}_1$ must be active, that is, the SNAKES tiling is valid in position $\vec{p}_1$.
- The bits stored in the next position $\vec{p}_2$ (indicated by the direction) are different in c and d.

Hence we can repeat the reasoning in position $\vec{p}_2$.

The same reasoning can be repeated over and over again. The positions $\vec{p}_1, \vec{p}_2, \vec{p}_3, \ldots$ form a path that follows the arrows on the tiles. There is no tiling error at any tile on this path.

But this contradicts the fact that the plane filling property of SNAKES guarantees that on periodic configuration every path encounters a tiling error.

Algorithmic aspects

Two types of aspects:

(1) Algorithmic questions concerning cellular automata (is a given CA injective? surjective? ...?). Many questions turn out to be **undecidable**: no algorithm exists.

(2) Computation by cellular automata: CA themselves are computation devices that can be used to solve algorithmic questions. Some cellular automata are **computationally universal**.

Algorithms, semi-algorithms and undecidability

Question: How can one determine if a given cellular automaton is reversible?

This is an example of an **algorithmic** question (looking for an algorithm to solve the problem).

More specifically, it is a **decision problem**: the desired algorithm

- takes an input, called an **instance** of the problem. (. . . **given** cellular automaton. . .)
- should return the correct **"yes"** or **"no"** answer. (So "yes" if and only if the cellular automaton is injective.)

Algorithms, semi-algorithms and undecidability

Question: How can one determine if a given cellular automaton is reversible?

This is an example of an **algorithmic** question (looking for an algorithm to solve the problem).

More specifically, it is a **decision problem**: the desired algorithm

- takes an input, called an **instance** of the problem. (. . . **given** cellular automaton. . .)
- should return the correct **"yes"** or **"no"** answer. (So "yes" if and only if the cellular automaton is injective.)

It turns out that such an algorithm **does not exist**: any candidate algorithm either gives a wrong answer for some cellular automaton, or gives no answer at all for some instances. We say the decision problem is **undecidable**.

If the instances are limited to **one-dimensional** CA, then an algorithm exists, and the problem is **decidable**.

An **algorithm** can be formally defined in various, equivalent ways.

For our purposes it is sufficient to understand a (decision) algorithm to be a **computer program** that takes some input and returns a "yes" or a "no" answer on each input. We say that the algorithm **solves** a decision problem if the algorithm returns the correct yes/no -answer on every instance of the problem.

If such an algorithm exists then the decision problem is called **decidable** and if no such algorithm exists then the problem is **undecidable**.

Example. The following two decision problems are decidable:

1D reversibility

Instance: One-dimensional cellular automaton A

Positive instance: A is reversible (=injective)

An algorithm constructs the pair graph and checks whether the graph contains a cycle through some vertex that is outside the diagonal Δ.

1D surjectivity

Instance: One-dimensional cellular automaton A

Positive instance: A is surjective (=pre-injective)

An algorithm constructs the pair graph and checks whether the graph contains a cycle that contains a vertex from Δ and from outside of Δ.

A **semi-algorithm** is a weaker concept than an algorithm: it is a computer program that must halt and return "yes" if the input is a positive instance of the problem, but it is allowed to run forever, without ever halting, on negative input instances. (It may never return a wrong answer.)

If a decision problem has a semi-algorithm then it is called **semi-decidable**. Clearly every decidable problem is also semi-decidable as an algorithm is also a semi-algorithm.

Example. The following decision problem is semi-decidable.

2D reversibility

Instance: Two-dimensional cellular automaton A

Positive instance: A is reversible (=injective)

Example. The following decision problem is semi-decidable.

2D reversibility

Instance: Two-dimensional cellular automaton A

Positive instance: A is reversible (=injective)

A **semi-algorithm** enumerates one-by-one all two-dimensional CA X that have the same state set as A. For each X it constructs the composed CA $X \circ A$ and checks whether $X \circ A$ computes the identity function. If it does, return "yes".

Note that we can **effectively** enumerate X, form the composition of A and X and verify if the result gives the identity.

If A is not injective the process will never halt, so this is not an algorithm but only a semi-algorithm.

The **complement** problem of a decision problem is the problem where the "yes" and "no" instances have been swapped.

Note that the complement of a decidable problem is also decidable: the same algorithm works, just swap the answer

$$\text{"yes"} \longleftrightarrow \text{"no"}$$

If the complement of problem P is semi-decidable we also may say that the negative instances of P are semi-decidable. Semi-decidability of P means that the positive instances are semi-decidable.

Example. The **complement** of the following decision problem is semi-decidable.

2D surjectivity

Instance: Two-dimensional cellular automaton A

Positive instance: A is surjective (=pre-injective)

Example. The **complement** of the following decision problem is semi-decidable.

2D surjectivity

Instance: Two-dimensional cellular automaton A

Positive instance: A is surjective (=pre-injective)

A semi-algorithm for the negative instances (non-surjective CA) enumerates one-by-one all finite patterns $p \in S^D$ and checks whether any of the (finitely many) patterns with domain $N(D)$ maps to p. If none of them gives p then p is an orphan, so the semi-algorithm halts and returns "no".

If A is surjective then the process will never halt.

Proposition. If problem P and the complement of P are both semi-decidable, then P is decidable.

Proof. Run the semi-algorithms for P and its complement in parallel. Eventually one of them gives an answer.

The input to a computer program is actually an **encoding** of an instance as a word over some alphabet Σ.

Strictly speaking then, decision problems come with functions that encode instances as words in Σ^*. The set of all encodings of positive instances is the **language** L corresponding to the decision problem (and the encoding function). Solving the decision problem means deciding if a given word is in the language L.

The decision problem whether a given word belongs to a fixed language $L \subseteq \Sigma^*$ is the **membership problem** of language L. Language L is called

- **recursive** if its membership problem is decidable,
- **recursively enumerable** if its membership problem is semi-decidable.

So a decision problem is decidable (or semi-decidable) if and only if the membership problem for the corresponding language is recursive (or recursively enumerable, respectively).

Remark: We assume effective encodings in the sense that the language of all encodings of instances (both positive and negative) is recursive. So we are able to determine if a given word is a valid encoding of an instance.

The number of different algorithms and semi-algorithms is **countable**: Each algorithm can be represented as a finite string (e.g. its source code in programming language C).

So there are only countably many recursive and recursively enumerable languages over any alphabet Σ.

The number of different algorithms and semi-algorithms is **countable**: Each algorithm can be represented as a finite string (e.g. its source code in programming language C).

So there are only countably many recursive and recursively enumerable languages over any alphabet Σ.

But there are **uncountably** many different languages $L \subseteq \Sigma^*$, which implies that many languages are not recursive (or even recursively enumerable). Their membership problem is not decidable (or semi-decidable), so we see that there are **many undecidable decision problems**.

This is not surprising, but what is more interesting is that we can prove some individual problems undecidable, and that many of these undecidable problems are quite natural.

To obtain a **first undecidable decision problem** we turn to (semi-)algorithms whose input is a semi-algorithm A.

Let $\Sigma = \{0, 1\}$ denote the binary alphabet. We can encode instances as words over Σ. So in the following we consider semi-algorithms whose input is a word over the alphabet Σ.

Any semi-algorithm A itself is written as a word over the alphabet Σ. This means that A is encoded as, say, a binary string representing the source code for A in language C.

To clarify the distinction we'll denote by $\langle A \rangle$ the **encoding** of A over the binary alphabet.

Semi-algorithm halting

Instance: Two binary words: a semi-algorithm $\langle A \rangle$ and an input word w

Positive instance: A halts on input w

Proposition. The decision problem Semi-algorithm halting is undecidable.

Proof. Diagonal argument by A. Turing.

Remark: The proof is a variant of Cantor's diagonal argument.

Imagine an infinite binary matrix M whose rows and columns are indexed by all semi-algorithms, and where

$$\mathbf{M[A,B]} = \begin{cases} 0, & \text{if } A \text{ does not halt on input } \langle B \rangle, \\ 1, & \text{if } A \text{ halts on input } \langle B \rangle. \end{cases}$$

Consider the diagonal elements $M[A, A]$ of the matrix. Reading the diagonal and swapping each value $0 \leftrightarrow 1$ creates a sequence that is different from every row of the matrix.

This means that there is no semi-algorithm enlisted that would halt on exactly those inputs $\langle A \rangle$ for which A does not halt on input $\langle A \rangle$. But such a semi-algorithm should exist if Semi-algorithm halting were decidable.

Turing machines

Turing machines are toy computers that are extremely simple but nevertheless powerful enough to simulate arbitrary algorithms.

We consider Turing machines because

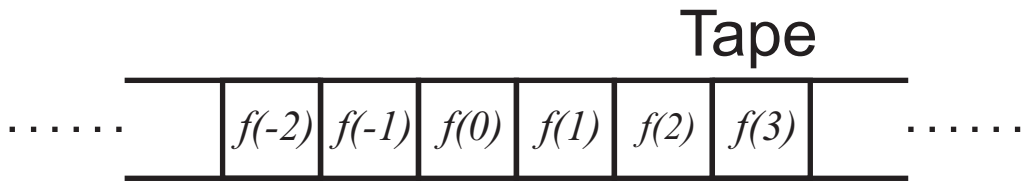
(i) We use **reductions** to establish new undecidability results. Direct reductions from C programs into questions concerning tilings and cellular automata would involve simulations of C programs by tilings or CA. Reductions from Turing machines are easier.

(ii) We want to show that cellular automata can perform **arbitrary computations**. This requires simulating arbitrary (semi-)algorithms on cellular automata. As in (i), it is much easier to show how to simulate a simple device such as a Turing machine rather than an algorithm given as a C program.

A Turing machine is an object with:

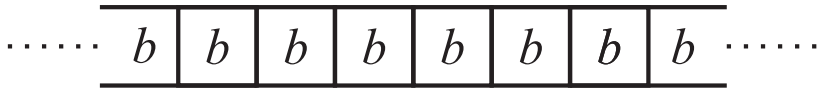
- a bi-infinite **tape** of cells indexed by $\mathbb{Z}$,
- a finite set Γ, the **tape alphabet**, of symbols that are written in the tape cells.

So the **content** of the tape is then a function $f : \mathbb{Z} \longrightarrow \Gamma$ where $f(i)$ is the symbol at location i. The set of possible tape contents is $\Gamma^{\mathbb{Z}}$.

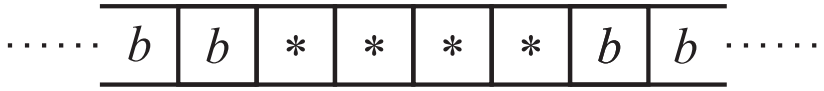


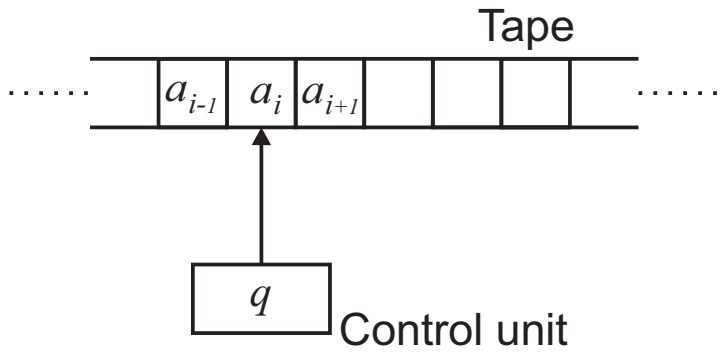
One element $b \in \Gamma$ is specified as the **blank symbol**.

- A tape content is the **blank tape** if every cell contains b.



- A tape content is **finite** if the number of cells with a non-b symbol is finite.





There is also a **control unit**, or a processor, that has access to one tape cell at any time. The unit is in some state q that is an element of a finite **state set** S.

TM operates at discrete time steps: depending on the

- current state, and
- the tape symbol currently in the accessed tape cell,

the TM may

- change its state,
- replace the tape symbol in the accesses tape cell, and
- change the accessed cell by moving the control unit one cell position to the left or right on the tape.

The action is specified by the **transition function**

$$\delta : S \times \Gamma \longrightarrow S \times \Gamma \times \{-1, +1\}.$$

The interpretation of

$$\delta(q, x) = (p, y, d)$$

is that if

- the current state is q, and
- the tape symbol at the current location i is x

then the machine

- changes the state into p,
- replaces x by y on the tape, and
- moves to location $i + d$ (one position left or right) on the tape, based on d.

A **configuration** (or **instataneous description, ID**) of the TM is an element of the set

$$S \times \mathbb{Z} \times \Gamma^{\mathbb{Z}}.$$

Configuration (q, i, f) specifies

- the current state $q \in S$,
- the current position $i \in \mathbb{Z}$ on the tape, and
- the current tape content $f : \mathbb{Z} \longrightarrow \Gamma$.

A **configuration** (or **instataneous description, ID**) of the TM is an element of the set

$$S \times \mathbb{Z} \times \Gamma^{\mathbb{Z}}.$$

Configuration (q, i, f) specifies

- the current state $q \in S$,
- the current position $i \in \mathbb{Z}$ on the tape, and
- the current tape content $f : \mathbb{Z} \longrightarrow \Gamma$.

We can now define one move of the machine formally: Configuration (q, i, f) is transformed in one move into the configuration (p, j, g) if

- $\delta(q, f(i)) = (p, y, d)$,
- $g(i) = y$, $g(k) = f(k)$ for all $k \neq i$,
- $j = i + d$.

We denote this move by

$$(q, i, f) \vdash (p, j, g).$$

Notations: We denote

- $(q, i, t) \vdash^n (q', i', t')$ if (q, i, t) turns into (q', i', t') using n moves,

- $(q, i, t) \vdash^+ (q', i', t')$ if (q, i, t) turns into (q', i', t') using some positive number of moves,

- $(q, i, t) \vdash^* (q', i', t')$ if (q, i, t) turns into (q', i', t') using zero or more moves.

The Turing machine can be understood as a **dynamical system** where the transformation $\vdash$ is applied repeatedly on configurations.

But Turing machines can also be used to recognize languages over a specified **input alphabet**

$$\Sigma \subseteq \Gamma \setminus \{b\}.$$

Note that the blank symbol b is not in the input alphabet.

In the beginning of the computation the Turing machine is in one specific state $q_0 \in S$ called the **initial state**, and two other states $q_a \in S$ and $q_r \in S$ are specified as the **accepting** and **rejecting** states.

The Turing machine **halts** when the control unit enters state q_a of q_r. The transitions from states q_a and q_r are then irrelevant, and in the following we assume that for all $x \in \Gamma$

$$\delta(q_a, x) = (q_a, x, 1) \text{ and } \delta(q_r, x) = (q_r, x, 1).$$

For any $w \in \Sigma^*$ we denote by $t_w \in \Gamma^{\mathbb{Z}}$ the tape content where we have written word w in cells $1, 2, \ldots, |w|$ and all other cells contain the blank symbol b. The machine **accepts** the input word w if and only if

$$(q_0, 1, t_w) \vdash^* (q_a, i, t)$$

for some $i \in \mathbb{Z}$ and $t \in \Gamma^{\mathbb{Z}}$.

Rejection can then happen in two ways: Either the machine halts in the rejecting state q_r or it never halts.

The set of accepted words is the language $L(M) \subseteq \Sigma^*$ **recognized** by the Turing machine M.

So, to specify a Turing machine we provide eight items. We say that a Turing machine is an 8-tuple

$$M = (S, \Gamma, \Sigma, \delta, q_0, q_a, q_r, b)$$

where S, Γ and Σ are finite sets with $\Sigma \subseteq \Gamma$, items $q_0, q_a, q_r \in S$ and $b \in \Gamma \setminus \Sigma$ are elements of those sets, and $\delta : S \times \Gamma \longrightarrow S \times \Gamma \times \{-1, +1\}$ is a function.

So, to specify a Turing machine we provide eight items. We say that a Turing machine is an 8-tuple

$$M = (S, \Gamma, \Sigma, \delta, q_0, q_a, q_r, b)$$

where S, Γ and Σ are finite sets with $\Sigma \subseteq \Gamma$, items $q_0, q_a, q_r \in S$ and $b \in \Gamma \setminus \Sigma$ are elements of those sets, and $\delta : S \times \Gamma \longrightarrow S \times \Gamma \times \{-1, +1\}$ is a function.

Example. Consider the following TM

$$M = (\{s, t, h, r\}, \{a, b\}, \emptyset, \delta, s, h, r, b)$$

where

$$\begin{aligned} \delta(s, a) &= (t, a, -1) \\ \delta(s, b) &= (t, a, +1) \\ \delta(t, a) &= (h, a, -1) \\ \delta(t, b) &= (s, a, -1) \end{aligned}$$

(and $\delta(h, x) = (h, x, 1)$ and $\delta(r, x) = (r, x, 1)$ for all x.)

The operation of M from the initial blank tape:

So, to specify a Turing machine we provide eight items. We say that a Turing machine is an 8-tuple

$$M = (S, \Gamma, \Sigma, \delta, q_0, q_a, q_r, b)$$

where S, Γ and Σ are finite sets with $\Sigma \subseteq \Gamma$, items $q_0, q_a, q_r \in S$ and $b \in \Gamma \setminus \Sigma$ are elements of those sets, and $\delta : S \times \Gamma \longrightarrow S \times \Gamma \times \{-1, +1\}$ is a function.

Example. Consider the following TM

$$M = (\{s, t, h, r\}, \{a, b\}, \emptyset, \delta, s, h, r, b)$$

where

$$\begin{aligned} \delta(s, a) &= (t, a, -1) \\ \delta(s, b) &= (t, a, +1) \\ \delta(t, a) &= (h, a, -1) \\ \delta(t, b) &= (s, a, -1) \end{aligned}$$

(and $\delta(h, x) = (h, x, 1)$ and $\delta(r, x) = (r, x, 1)$ for all x.)

This is the two-state **Busy Beaver**: when started on the blank tape it runs for the longest (finite) time before halting among all TM with two non-halting states and two tape letters.

The language $L(M)$ recognized by a Turing machine M is clearly **recursively enumerable**: A semi-algorithm simulates M step-by-step on any given input word until (if ever) the word gets accepted. If the word is not accepted then the semi-algorithm may not halt.

The language $L(M)$ recognized by a Turing machine M is clearly **recursively enumerable**: A semi-algorithm simulates M step-by-step on any given input word until (if ever) the word gets accepted. If the word is not accepted then the semi-algorithm may not halt.

Also the converse is true: every recursively enumerable language is recognized by a Turing machine. This is based on the fact that Turing machines — despite the simplicity of their individual moves — can simulate arbitrary semi-algorithms, as formalized by the next proposition.

Proposition. Given a (semi)algorithm A whose inputs are words over an alphabet Σ one can effectively construct a Turing Machine M with input alphabet Σ such that on every input $w \in \Sigma^*$

- if A returns "yes" then M accepts w (enters its accepting state q_a),
- if A returns "no" then M rejects w by entering its rejecting state q_r,
- if A does not halt then M rejects w by not halting.

Proposition. Given a (semi)algorithm A whose inputs are words over an alphabet Σ one can effectively construct a Turing Machine M with input alphabet Σ such that on every input $w \in \Sigma^*$

- if A returns "yes" then M accepts w (enters its accepting state q_a),
- if A returns "no" then M rejects w by entering its rejecting state q_r,
- if A does not halt then M rejects w by not halting.

Corollary. A language is recursively enumerable if and only if there is a Turing machine that recognizes it.

Corollary. A language is recursive if and only if it is recognized by a Turing machine that halts on every input word.

Proposition. Given a (semi)algorithm A whose inputs are words over an alphabet Σ one can effectively construct a Turing Machine M with input alphabet Σ such that on every input $w \in \Sigma^*$

- if A returns "yes" then M accepts w (enters its accepting state q_a),
- if A returns "no" then M rejects w by entering its rejecting state q_r,
- if A does not halt then M rejects w by not halting.

Corollary. A language is recursively enumerable if and only if there is a Turing machine that recognizes it.

Corollary. A language is recursive if and only if it is recognized by a Turing machine that halts on every input word.

Remark: Often one takes Turing machines as the formal definition of (semi)algorithms. In this case the above corollaries are definitions, and theorems state how (semi)algorithms defined under other models (C-programs, for example) are equivalent to Turing machines.

We did the process in reverse: (semi-)algorithms were defined in terms of C-programs, and the Corollaries above tie them to Turing machines.

Recall the decision problem Semi-algorithm halting that we proved to be undecidable.

We can **reduce** this problem to decision problems concerning Turing machines, thus proving these new problems to be undecidable as well.

Recall the decision problem Semi-algorithm halting that we proved to be undecidable.

We can **reduce** this problem to decision problems concerning Turing machines, thus proving these new problems to be undecidable as well.

Turing reductions: If we want to prove that a decision problem P is undecidable we describe an algorithm for some known undecidable problem U that makes (possibly several) calls to a hypothetical algorithm for P. Since U is previously known to be undecidable such algorithm cannot exist. Therefore the hypothetical algorithm for P cannot exist either.

Recall the decision problem Semi-algorithm halting that we proved to be undecidable.

We can **reduce** this problem to decision problems concerning Turing machines, thus proving these new problems to be undecidable as well.

Turing reductions: If we want to prove that a decision problem P is undecidable we describe an algorithm for some known undecidable problem U that makes (possibly several) calls to a hypothetical algorithm for P. Since U is previously known to be undecidable such algorithm cannot exist. Therefore the hypothetical algorithm for P cannot exist either.

Many-one reductions: Describe an algorithm that converts an arbitrary input u to a known undecidable problem U into an equivalent instance p of the decision problem P. (Equivalent means that p is a positive instance of P if and only if u is a positive instance of U.)

A many-one reduction is a restricted type of a Turing reduction: An algorithm to solve U converts its input u into the equivalent input p to P and calls the hypothetical algorithm for P with input p.

Example of a reduction: Let us show that the following decision problem is undecidable

TM halting on blank tape

Instance: A Turing machine M

Positive instance: M halts when started on the blank tape

by making a many-one reduction from the undecidable problem

Semi-algorithm halting

Instance: Two binary words: a semi-algorithm $\langle A \rangle$ and an input string w

Positive instance: A halts on input w

Example. Consider the following **Busy beaver** function

$$BB : \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$$

where $BB(n, m)$ is the largest number t such that there is a Turing machine with n non-halting states and m tape symbols that makes t moves on the blank tape and then halts.

It is known that

- $BB(2, 2) = 6$,
- $BB(3, 2) = 21$,
- $BB(4, 2) = 107$,
- $BB(5, 2) = 47176870$,
- $BB(3, 3) \geq 119\ 112\ 334\ 170\ 342\ 541$.
- $BB(6, 2) > 2 \uparrow\uparrow (2 \uparrow\uparrow (2 \uparrow\uparrow 10))$ (recall: $a \uparrow\uparrow b$ is the exponentiation tower $a^{a^{\cdot^{\cdot^{\cdot^{a}}}}}$ of height b.)

Example. Consider the following **Busy beaver** function

$$BB : \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$$

where $BB(n, m)$ is the largest number t such that there is a Turing machine with n non-halting states and m tape symbols that makes t moves on the blank tape and then halts.

Function BB cannot be upper bounded by any computable function: If there exists a computable function $f : \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$ such that $BB(n, m) \leq f(n, m)$ for all n, m then we can algorithmically solve TM halting on blank tape.