Computability II

Martin Hoefer

(based on material by Walter Unger)

- More problems that can not be solved by the computer.

The Post correspondence problem (1)

The **Post correspondence problem** (PCP) is a domino puzzle.

- Each domino piece is labeled with two words over an alphabet Σ, one word in the upper half and one word in the lower half.

The Post correspondence problem (1)

The **Post correspondence problem** (PCP) is a domino puzzle.

- Each domino piece is labeled with two words over an alphabet Σ, one word in the upper half and one word in the lower half.
- The input consists of a set K of domino pieces.
  Each piece may be used arbitrarily often.

The Post correspondence problem (1)

The **Post correspondence problem** (PCP) is a domino puzzle.

- Each domino piece is labeled with two words over an alphabet Σ, one word in the upper half and one word in the lower half.
- The input consists of a set K of domino pieces.
  Each piece may be used arbitrarily often.
- The goal is to find a **corresponding sequence** of dominoes in K, for which the concatenation of the upper words equals the concatenation of the lower words.

The Post correspondence problem (1)

The **Post correspondence problem** (PCP) is a domino puzzle.

- Each domino piece is labeled with two words over an alphabet Σ, one word in the upper half and one word in the lower half.
- The input consists of a set K of domino pieces.
  Each piece may be used arbitrarily often.
- The goal is to find a **corresponding sequence** of dominoes in K, for which the concatenation of the upper words equals the concatenation of the lower words.
- The sequence must contain at least one domino.

The Post correspondence problem (2a)

Example A

For the domino set

$$K = \left\{ \begin{bmatrix} b \\ \hline ca \end{bmatrix}, \begin{bmatrix} a \\ \hline ab \end{bmatrix}, \begin{bmatrix} ca \\ \hline a \end{bmatrix}, \begin{bmatrix} dbd \\ \hline cef \end{bmatrix} \begin{bmatrix} abc \\ \hline c \end{bmatrix} \right\}$$

there exists the corresponding sequence $\langle 2, 1, 3, 2, 5 \rangle$ with

$$\begin{bmatrix} a \\ \hline ab \end{bmatrix} \begin{bmatrix} b \\ \hline ca \end{bmatrix} \begin{bmatrix} ca \\ \hline a \end{bmatrix} \begin{bmatrix} a \\ \hline ab \end{bmatrix} \begin{bmatrix} abc \\ \hline c \end{bmatrix}$$

The Post correspondence problem (2b)

Example B

Not every domino set K allows a corresponding sequence, as for instance the domino set

$$K = \left\{ \left[\frac{abc}{ca}\right], \left[\frac{b}{aa}\right], \left[\frac{abcb}{abc}\right], \left[\frac{abc}{bc}\right] \right\}$$

Why is there no solution?

The Post correspondence problem (2c)

As an exercise, you may want to find the shortest corresponding sequences for the following three PCPs (you will need a computer for doing so):

Example C

$$K_1 = \left\{ \left[\frac{aaba}{a}\right], \left[\frac{baab}{aa}\right], \left[\frac{a}{aab}\right] \right\}$$

$$K_2 = \left\{ \left[\frac{aaa}{aab}\right], \left[\frac{baa}{a}\right], \left[\frac{ab}{abb}\right], \left[\frac{b}{aa}\right] \right\}$$

$$K_3 = \left\{ \left[\frac{aab}{a}\right], \left[\frac{a}{ba}\right], \left[\frac{b}{aab}\right] \right\}$$

The Post correspondence problem (3)

Formal Definition (Post correspondence problem; PCP for short)

An **instance** of the PCP consists of a finite set

$$K = \left\{ \left[\frac{x_1}{y_1}\right], \ldots, \left[\frac{x_k}{y_k}\right] \right\}$$

where $x_1, \ldots, x_k$ and $y_1, \ldots, y_k$ are non-empty words over some finite alphabet Σ.

The problem consists in deciding, whether there exists some **corresponding sequence** $\langle i_1, \ldots, i_n \rangle$ of indices in $\{1, \ldots, k\}$ with

$$x_{i_1} x_{i_2} \ldots x_{i_n} = y_{i_1} y_{i_2} \ldots y_{i_n}$$

The elements of K are called **domino pieces** or **dominoes**.

The Post correspondence problem (4)

In 1946, the American mathematician Emil Leon Post established the following theorem:

Theorem

The Post correspondence problem is undecidable.

The Post correspondence problem (4)

In 1946, the American mathematician Emil Leon Post established the following theorem:

Theorem

The Post correspondence problem is undecidable.

The proof essentially applies the sub-program technique, and shows that an algorithm for the PCP would imply the existence of an algorithm for the halting program.
The proof is long, somewhat tedious, and omitted in this course.

Context-free grammars (1)

In week 3 of this course, you have seen the following:

Definition

A **context-free grammar** (CFG) G is a quadruple (N, Σ, P, S), where

- N = set of non-terminal symbols
- Σ = terminal alphabet
- P = set of rules of the form $A \to w$
  where $A \in N$ and $w \in (\Sigma \cup N)^*$
- S = special element of N (startsymbol)

Note: In every rule in P the left hand side consists of a single non-terminal symbol

Context-free grammars (2)

Example

- $N = \{S\}$
- $\Sigma = \{a, b, c\}$
- P: $S \to aSa \mid bSb \mid cSc$; $S \to a \mid b \mid c$; $S \to aa \mid bb \mid cc$

Derivation:

$$S \to aSa \to acSca \to acaSaca \to acaaSaaca \to acaabbaaca$$

Derivation:

$$S \to bSb \to bbSbb \to bbbSbbb \to bbbbbbbb$$

Definition

$L(G)$ is the set of all words over the terminal alphabet Σ,
that can be derived from the startsymbol S
by repeated application of rules in P.

Decision problems for CFGs

The following problems for CFGs are decidable:

- Given CFG $\langle G \rangle$ and $w \in \Sigma^*$, does $w \in L(G)$ hold?
- Given CFG $\langle G \rangle$, is $L(G)$ empty?
- Given CFG $\langle G \rangle$, is $L(G)$ finite?

Decision problems for CFGs

The following problems for CFGs are decidable:

- Given CFG $\langle G \rangle$ and $w \in \Sigma^*$, does $w \in L(G)$ hold?
- Given CFG $\langle G \rangle$, is $L(G)$ empty?
- Given CFG $\langle G \rangle$, is $L(G)$ finite?

The following problems for CFGs are undecidable:

- Given CFG $\langle G \rangle$, does $L(G) = \Sigma^*$ hold?
- Given CFG $\langle G \rangle$, is $L(G)$ regular?
- Given CFGs $\langle G_1 \rangle$ and $\langle G_2 \rangle$, does $L(G_1) \subseteq L(G_2)$ hold?
- Given CFGs $\langle G_1 \rangle$ and $\langle G_2 \rangle$, is $L(G_1) \cap L(G_2)$ empty?

Recall: All these problems are decidable for regular languages

Empty intersection

Theorem

It is undecidable, whether the languages generated by two given CFGs G_1 and G_2 have empty intersection.

Empty intersection

Theorem

It is undecidable, whether the languages generated by two given CFGs G_1 and G_2 have empty intersection.

Sketch of proof:

- Consider an arbitrary PCP instance $\left\{ \left[\frac{x_1}{y_1}\right], \ldots, \left[\frac{x_k}{y_k}\right] \right\}$
- Let $b_1, \ldots, b_k$ be letters that do not occur in x_i and y_i

Empty intersection

Theorem

It is undecidable, whether the languages generated by two given CFGs G_1 and G_2 have empty intersection.

Sketch of proof:

- Consider an arbitrary PCP instance $\left\{ \left[\frac{x_1}{y_1}\right], \ldots, \left[\frac{x_k}{y_k}\right] \right\}$
- Let $b_1, \ldots, b_k$ be letters that do not occur in x_i and y_i
- Construct CFGs G_1 and G_2 with the following rules:

$$G_1 : \; S \to x_1 S b_1 \mid x_2 S b_2 \mid \cdots \mid x_k S b_k \mid x_1 b_1 \mid \cdots \mid x_k b_k$$
$$G_2 : \; S \to y_1 S b_1 \mid y_2 S b_2 \mid \cdots \mid y_k S b_k \mid y_1 b_1 \mid \cdots \mid y_k b_k$$

- PCP is solvable if and only if $L(G_1) \cap L(G_2)$ is non-empty

$$G_1 : \quad S \xrightarrow{*} x_8\, x_1\, x_4\, x_2\, x_5\, x_1\, x_4\; b_4\, b_1\, b_5\, b_2\, b_4\, b_1\, b_8$$
$$G_2 : \quad S \xrightarrow{*} y_8\, y_1\, y_4\, y_2\, y_5\, y_1\, y_4\; b_4\, b_1\, b_5\, b_2\, b_4\, b_1\, b_8$$

Turing-recognizable languages (1)

Recall:

A language L is **decided** by a C++ program P

- if P halts on every input, and
- if P exactly accepts the words in L.

Such a language L is called **Turing-decidable**, or simply **decidable**.

Turing-recognizable languages (1)

Recall:

A language L is **decided** by a C++ program P

- if P halts on every input, and
- if P exactly accepts the words in L.

Such a language L is called **Turing-decidable**, or simply **decidable**.

A language L is **recognized** by a C++ program P

- if P accepts every word in L, and
- if P does not accept any word not in L.

Such a language L is called **Turing-recognizable**, or simply **recognizable**.

Turing-recognizable languages (2): Example

Example

The halting problem $H = \{\langle P \rangle w \mid P \text{ halts on } w\}$ is not Turing-decidable, but Turing-recognizable.

Turing-recognizable languages (2): Example

Example

The halting problem $H = \{\langle P \rangle w \mid P \text{ halts on } w\}$ is not Turing-decidable, but Turing-recognizable.

Proof

The following C++ program P_H recognizes language H:

If P_H receives a syntactically incorrect input,

- then P_H rejects the input.

If P_H receives an input of the form $\langle P \rangle w$,

- then P_H simulates P on input w
- and accepts, if (as soon as) P halts on w.

Turing-recognizable languages (3): Exercise

Exercise

The Post Correspondence Problem (PCP) is not Turing-decidable, but Turing-recognizable.

Why?

Enumerators

Definition

An **enumerator** for a language $L \subseteq \Sigma^*$ is a variant of a C++ program with a **printer** attached to it.

The printer prints its output on an infinitely long piece of paper.

Enumerators

Definition

An **enumerator** for a language $L \subseteq \Sigma^*$ is a variant of a C++ program with a **printer** attached to it.

The printer prints its output on an infinitely long piece of paper.

- The enumerator program starts without input, and over time prints all the words in L (possibly with repetitions) on the printer.
- Every printed word is printed in a separate line.
- The enumerator only prints words from L.

Recursively enumerable languages

Definition

If a language L possesses an enumerator,
then L is called **recursively enumerable**, or simply **enumerable**.

Recursively enumerable languages

Definition

If a language L possesses an enumerator,
then L is called **recursively enumerable**, or simply **enumerable**.

Theorem (recognizability $\equiv$ enumerability)

A language L is **recursively enumerable**,
if and only if L is **Turing-recognizable**.

Proof (1)

Suppose that L is recursively enumerable by an enumerator E.
We construct a C++ program P that recognizes L.

On input word w the program P works as follows:

- P starts simulates E.
- Everytime E prints a new word, P compares the new word against w and accepts if the two words coincide.

Proof (1)

Suppose that L is recursively enumerable by an enumerator E.
We construct a C++ program P that recognizes L.

On input word w the program P works as follows:

- P starts simulates E.
- Everytime E prints a new word, P compares the new word against w and accepts if the two words coincide.

Correctness:

- If $w \in L$, then w eventually will be printed.
  At that point in time it will be accepted by P.
- If $w \notin L$, then w will never be printed and hence will never be accepted by P.

Proof (2)

Suppose that L is recognized by a C++ program P.
We construct an enumerator E for language L.

In the k-th round (with $k = 1, 2, 3, \ldots$)

- the enumerator simulates the execution of exactly k statements of program P on each of the words $w_1, \ldots, w_k$.
- Whenever the simulation accepts one of the words, this word is printed by the enumerator.

Proof (2)

Suppose that L is recognized by a C++ program P.
We construct an enumerator E for language L.

In the k-th round (with $k = 1, 2, 3, \ldots$)

- the enumerator simulates the execution of exactly k statements of program P on each of the words $w_1, \ldots, w_k$.
- Whenever the simulation accepts one of the words, this word is printed by the enumerator.

Correctness:
The enumerator E obviously prints only words from L.
But does it really print all the words from L ?

Proof (2)

Suppose that L is recognized by a C++ program P.
We construct an enumerator E for language L.

In the k-th round (with $k = 1, 2, 3, \ldots$)

- the enumerator simulates the execution of exactly k statements of program P on each of the words $w_1, \ldots, w_k$.
- Whenever the simulation accepts one of the words, this word is printed by the enumerator.

Correctness:
The enumerator E obviously prints only words from L.
But does it really print all the words from L ?

- Consider some word w_i in language L.

Proof (2)

Suppose that L is recognized by a C++ program P.
We construct an enumerator E for language L.

In the k-th round (with $k = 1, 2, 3, \ldots$)

- the enumerator simulates the execution of exactly k statements of program P on each of the words $w_1, \ldots, w_k$.
- Whenever the simulation accepts one of the words, this word is printed by the enumerator.

Correctness:
The enumerator E obviously prints only words from L.
But does it really print all the words from L ?

- Consider some word w_i in language L.
- Then w_i will be recognized/accepted by P after a finite number t_i of executed statements.

Proof (2)

Suppose that L is recognized by a C++ program P.
We construct an enumerator E for language L.

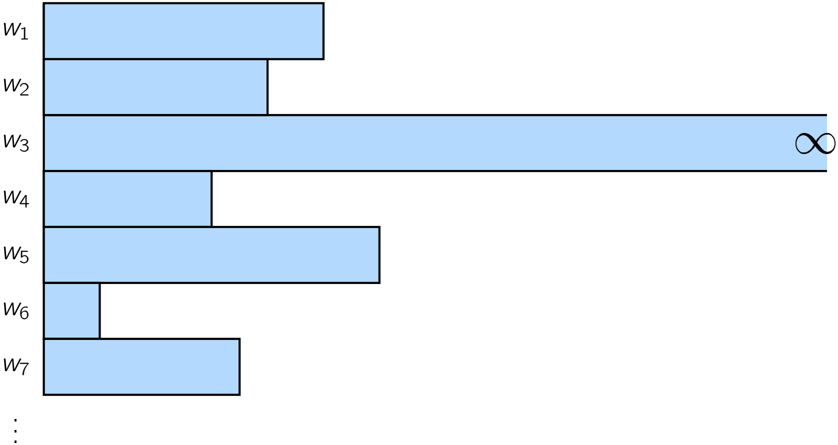
In the k-th round (with $k = 1, 2, 3, \ldots$)

- the enumerator simulates the execution of exactly k statements of program P on each of the words $w_1, \ldots, w_k$.
- Whenever the simulation accepts one of the words, this word is printed by the enumerator.

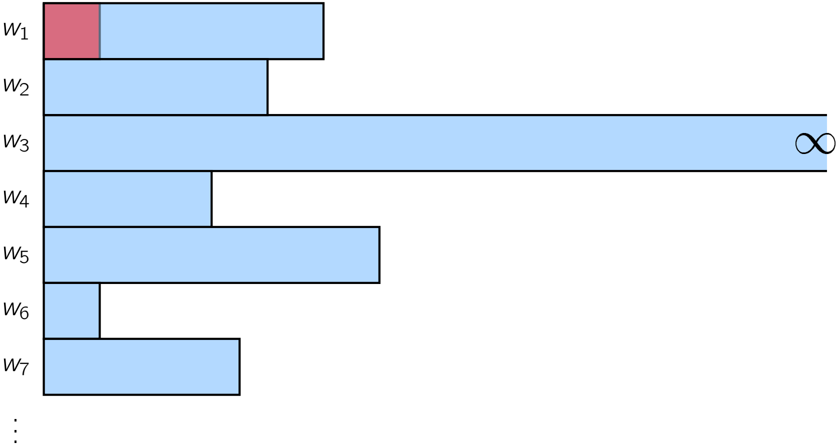
Correctness:
The enumerator E obviously prints only words from L.
But does it really print all the words from L ?

- Consider some word w_i in language L.
- Then w_i will be recognized/accepted by P after a finite number t_i of executed statements.
- Therefore the enumerator is going to print word w_i in every round k with $k \geq \max\{i, t_i\}$.

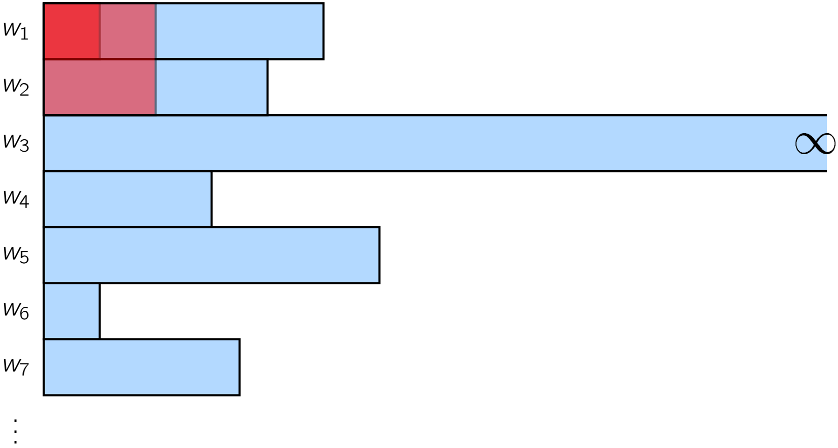
Number of steps that program P makes on input w_i: $\longrightarrow$



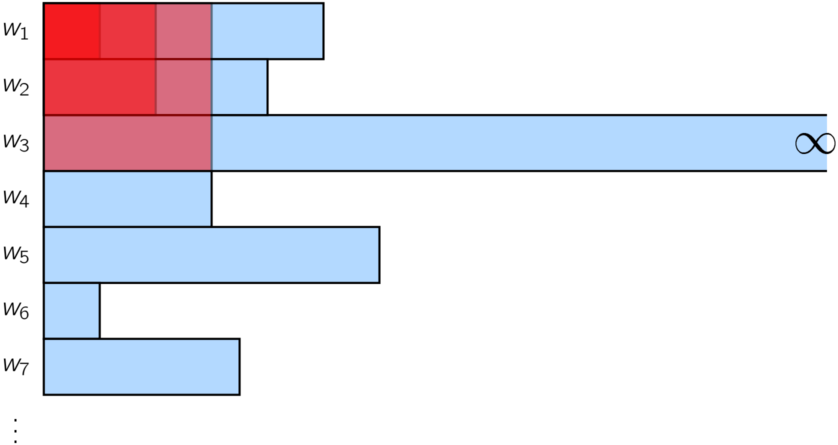
Number of steps that program P makes on input w_i: $\longrightarrow$



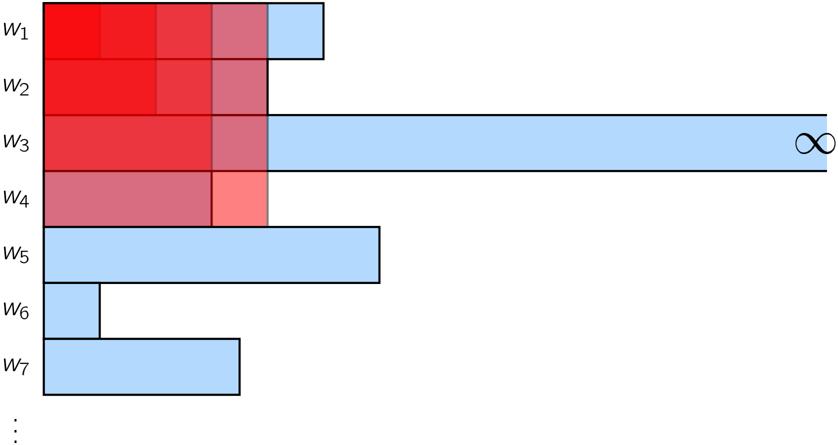
Number of steps that program P makes on input w_i: $\longrightarrow$



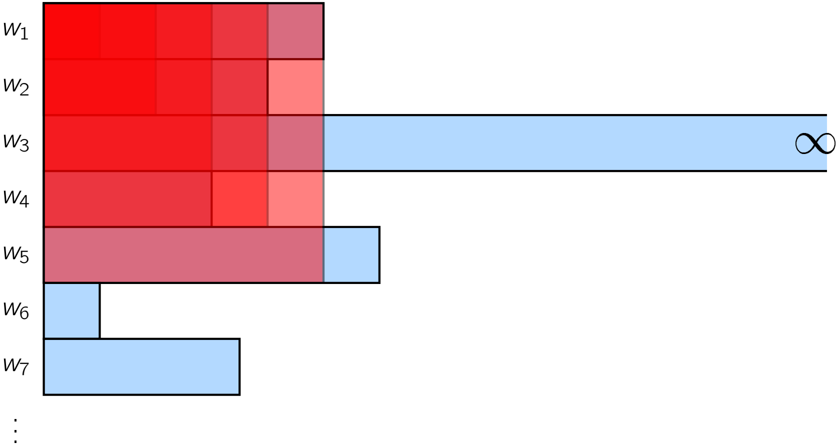
Number of steps that program P makes on input w_i: $\longrightarrow$



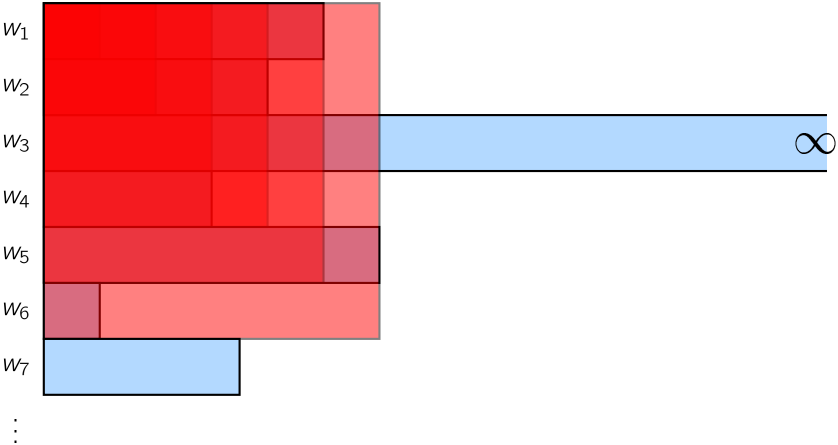
Number of steps that program P makes on input w_i: $\longrightarrow$



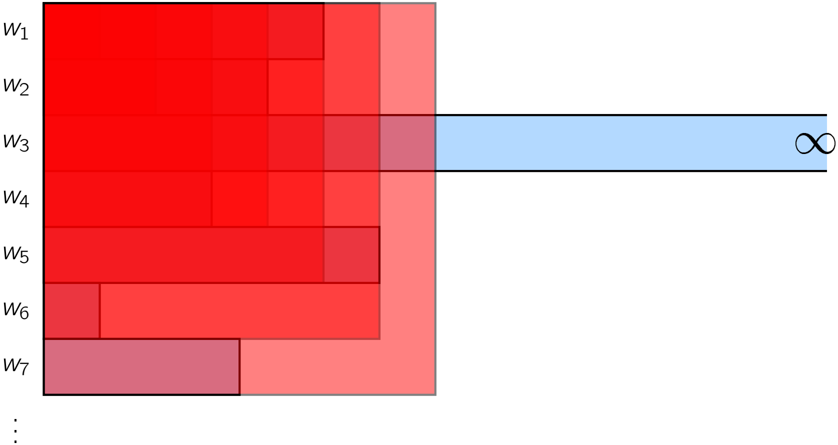
Number of steps that program P makes on input w_i: $\longrightarrow$



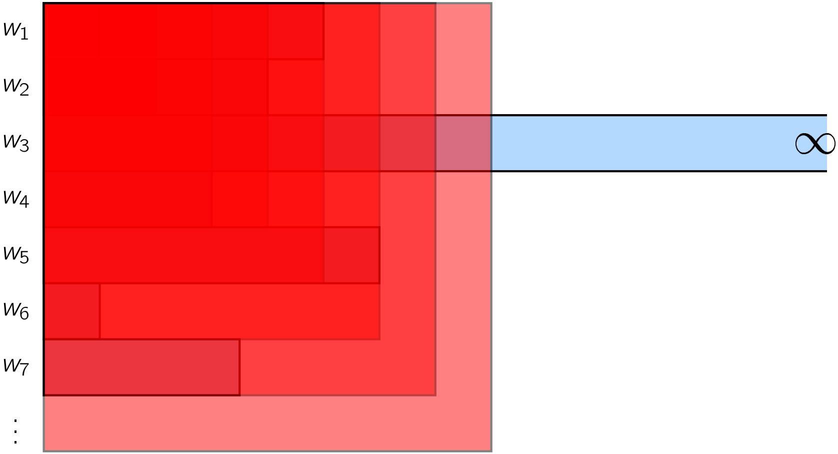
Number of steps that program P makes on input w_i: $\longrightarrow$



Number of steps that program P makes on input w_i: $\longrightarrow$



Number of steps that program P makes on input w_i: $\longrightarrow$



Intersection (1)

Theorem

(a) If both languages L_1 and L_2 are decidable, then also their intersection $L_1 \cap L_2$ is decidable.

(b) If both languages L_1 and L_2 are recursively enumerable, then also their intersection $L_1 \cap L_2$ is recursively enumerable.

Intersection (2a): Proof of part (a)

Let P_1 and P_2 be C++ programs, that respectively **decide** L_1 and L_2.

Intersection (2a): Proof of part (a)

Let P_1 and P_2 be C++ programs, that respectively **decide** L_1 and L_2.

New C++ program P

- On input w, program P first simulates the behavior of P_1 on w and then the behavior of P_2 on w.
- If P_1 and P_2 both accept w, then also P does accept; otherwise P rejects.

Intersection (2a): Proof of part (a)

Let P_1 and P_2 be C++ programs, that respectively **decide** L_1 and L_2.

New C++ program P

- On input w, program P first simulates the behavior of P_1 on w and then the behavior of P_2 on w.
- If P_1 and P_2 both accept w, then also P does accept; otherwise P rejects.

Correctness:

- If $w \in L_1 \cap L_2$, then w is accepted.
- Otherwise w is rejected.

Intersection (2b): Proof of part (b)

Now let P_1 and P_2 be C++ programs, that **recognize** L_1 and L_2.
We re-use the construction from part (a).

Intersection (2b): Proof of part (b)

Now let P_1 and P_2 be C++ programs, that **recognize** L_1 and L_2.
We re-use the construction from part (a).

New C++ program P

- On input w, program P first simulates the behavior of P_1 on w and then the behavior of P_2 on w.
- If P_1 and P_2 both accept w, then also P does accept w.

Union

Theorem

(a) If both languages L_1 and L_2 are decidable, then also $L_1 \cup L_2$ is decidable.

(b) If both languages L_1 and L_2 are recursively enumerable, then also $L_1 \cup L_2$ is recursively enumerable.

- Exercise: Prove these statements
- Attention: The proof of part (b) needs a small trick (programs P_1 and P_2 are to be executed in parallel)

Complement (1)

Lemma

If language $L \subseteq \Sigma^*$ and its complement $\overline{L} := \Sigma^* \setminus L$ are both recursively enumerable, then L is decidable.

Complement (1)

Lemma

If language $L \subseteq \Sigma^*$ and its complement $\overline{L} := \Sigma^* \setminus L$ are both recursively enumerable, then L is decidable.

Proof

- Let P and $\overline{P}$ be two C++ programs that recognize language L respectively language $\overline{L}$.

Complement (1)

Lemma

If language $L \subseteq \Sigma^*$ and its complement $\overline{L} := \Sigma^* \setminus L$ are both recursively enumerable, then L is decidable.

Proof

- Let P and $\overline{P}$ be two C++ programs that recognize language L respectively language $\overline{L}$.
- For an input word w, the new C++ program P' simulates the behavior of P on w and the behavior of $\overline{P}$ on w in parallel (one statement of P, one statement of $\overline{P}$, and so on)

Complement (1)

Lemma

If language $L \subseteq \Sigma^*$ and its complement $\overline{L} := \Sigma^* \setminus L$ are both recursively enumerable, then L is decidable.

Proof

- Let P and $\overline{P}$ be two C++ programs that recognize language L respectively language $\overline{L}$.
- For an input word w, the new C++ program P' simulates the behavior of P on w and the behavior of $\overline{P}$ on w in parallel (one statement of P, one statement of $\overline{P}$, and so on)
- If P accepts, then P' accepts.
  If $\overline{P}$ accepts, then P' rejects.

Complement (1)

Lemma

If language $L \subseteq \Sigma^*$ and its complement $\overline{L} := \Sigma^* \setminus L$ are both recursively enumerable, then L is decidable.

Proof

- Let P and $\overline{P}$ be two C++ programs that recognize language L respectively language $\overline{L}$.
- For an input word w, the new C++ program P' simulates the behavior of P on w and the behavior of $\overline{P}$ on w in parallel (one statement of P, one statement of $\overline{P}$, and so on)
- If P accepts, then P' accepts.
  If $\overline{P}$ accepts, then P' rejects.
- Since exactly one of $w \in L$ and $w \notin L$ holds, program $\overline{P}$ will terminate after finitely many steps.

Complement (2)

Observation 1

If language L is decidable, then also its complement $\overline{L}$ is decidable.

(Proof: Negate Yes/No acceptance behavior of a C++ program for L.)

Complement (2)

Observation 1

If language L is decidable, then also its complement $\overline{L}$ is decidable.

(Proof: Negate Yes/No acceptance behavior of a C++ program for L.)

Observation 2

The set of recursively enumerable languages is not closed under complement.

Complement (2)

Observation 1

If language L is decidable, then also its complement $\overline{L}$ is decidable.

(Proof: Negate Yes/No acceptance behavior of a C++ program for L.)

Observation 2

The set of recursively enumerable languages is not closed under complement.

Example

- The halting problem H is recursively enumerable.

Complement (2)

Observation 1

If language L is decidable, then also its complement $\overline{L}$ is decidable.

(Proof: Negate Yes/No acceptance behavior of a C++ program for L.)

Observation 2

The set of recursively enumerable languages is not closed under complement.

Example

- The halting problem H is recursively enumerable.
- If also the complement $\overline{H}$ is recursively enumerable, then language H would be decidable.

Complement (2)

Observation 1

If language L is decidable, then also its complement $\overline{L}$ is decidable.

(Proof: Negate Yes/No acceptance behavior of a C++ program for L.)

Observation 2

The set of recursively enumerable languages is not closed under complement.

Example

- The halting problem H is recursively enumerable.
- If also the complement $\overline{H}$ is recursively enumerable, then language H would be decidable.
- Hence $\overline{H}$ is not recursively enumerable.

The computability landscape (1)

Observation

Every language L belongs to exactly one of the following four families:

(1) L is decidable, and L as well as $\overline{L}$ are recursively enumerable.

(2) L is recursively enumerable, but $\overline{L}$ is not recursively enumerable

(3) $\overline{L}$ is recursively enumerable, but L is not recursively enumerable

(4) Neither L nor $\overline{L}$ are recursively enumerable

The computability landscape (1)

Observation

Every language L belongs to exactly one of the following four families:

(1) L is decidable, and L as well as $\overline{L}$ are recursively enumerable.

(2) L is recursively enumerable, but $\overline{L}$ is not recursively enumerable

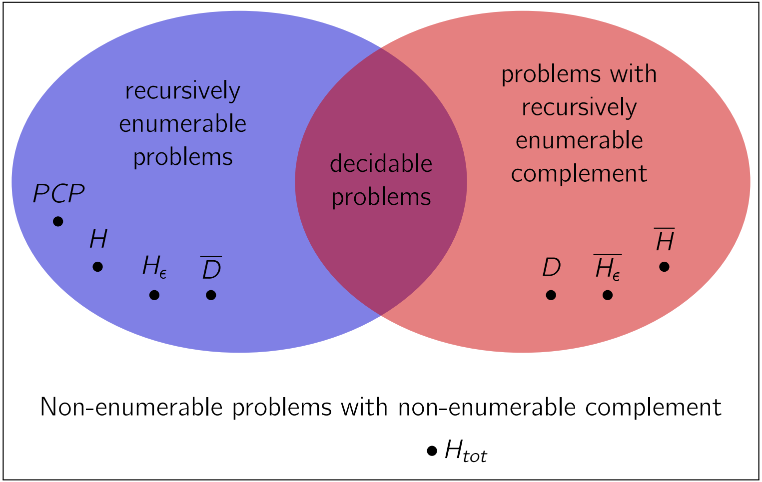
(3) $\overline{L}$ is recursively enumerable, but L is not recursively enumerable

(4) Neither L nor $\overline{L}$ are recursively enumerable

Examples

- Family 1: Graph connectivity; Hamiltonian cycle
- Family 2: H, H_ϵ, $\overline{D}$
- Family 3: $\overline{H}$, $\overline{H_\epsilon}$, D,
- Family 4: $H_{tot} = \{\langle P \rangle \mid P \text{ halts on every input}\}$

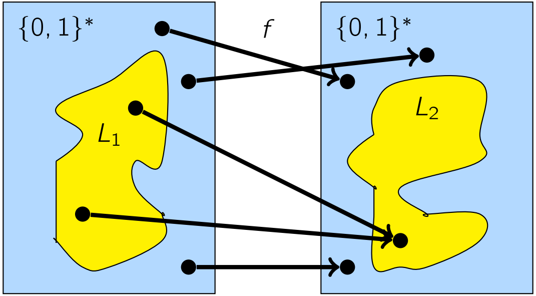
The computability landscape (2)



Reductions (1)

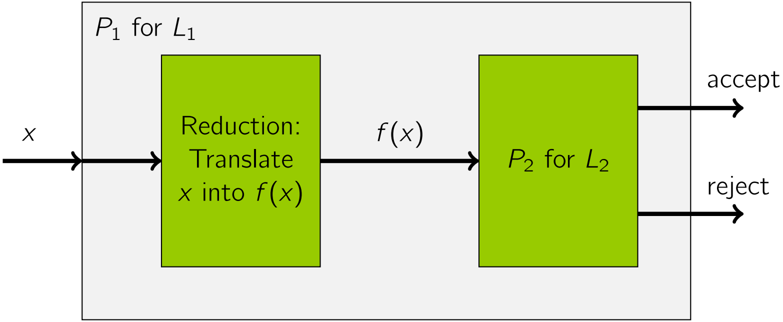
Definition

Let L_1 and L_2 be two languages over alphabet Σ.
Then L_1 is **reducible to** L_2 (with the notation $L_1 \leq L_2$),
if there exists a computable function $f\colon \Sigma^* \to \Sigma^*$,
so that for all $x \in \Sigma^*$ we have: $x \in L_1 \;\Leftrightarrow\; f(x) \in L_2$.



Reductions (2)

A reduction is an algorithm,
that formulates instances of a starting problem
as special cases of a target problem.



Reductions (3)

Theorem

If $L_1 \leq L_2$ and if L_2 is recursively enumerable,
then also L_1 is recursively enumerable.

Reductions (3)

Theorem

If $L_1 \leq L_2$ and if L_2 is recursively enumerable,
then also L_1 is recursively enumerable.

Proof: We construct a new C++ program P_1, that recognizes L_1, by using a sub-program P_2 that recognizes L_2:

- For an input x, program P_1 first computes $f(x)$.
- Then P_1 simulates P_2 on input $f(x)$.
- P_1 accepts input x, if P_2 accepts input $f(x)$.

Reductions (3)

Theorem

If $L_1 \leq L_2$ and if L_2 is recursively enumerable, then also L_1 is recursively enumerable.

Proof: We construct a new C++ program P_1, that recognizes L_1, by using a sub-program P_2 that recognizes L_2:

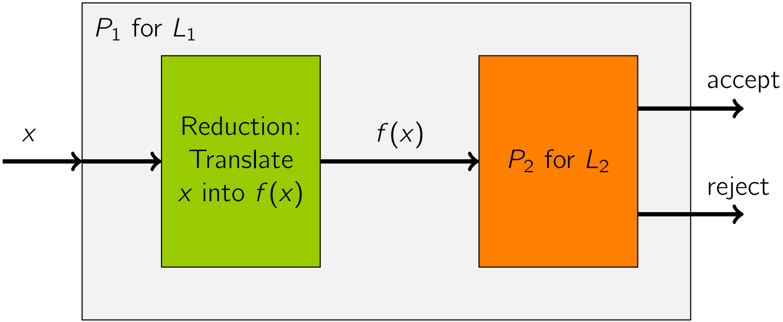
- For an input x, program P_1 first computes $f(x)$.
- Then P_1 simulates P_2 on input $f(x)$.
- P_1 accepts input x, if P_2 accepts input $f(x)$.

$$
\begin{aligned}
P_1 \text{ accepts } x \quad &\Leftrightarrow \quad P_2 \text{ accepts } f(x) \\
&\Leftrightarrow \quad f(x) \in L_2 \\
&\Leftrightarrow \quad x \in L_1
\end{aligned}
$$

Reductions (4)

The proven theorem and its logically equivalent reverse

- If $L_1 \leq L_2$ and if L_2 is recursively enumerable, then also L_1 is recursively enumerable.
- If $L_1 \leq L_2$ and if L_1 is **not** recursively enumerable, then also L_2 is **not** recursively enumerable,



The total halting problem

Definition (Total halting problem)

$$H_{\text{tot}} = \{\langle P\rangle \mid P \text{ halts for every input}\}$$

The total halting problem

Definition (Total halting problem)

$$H_{\text{tot}} = \{\langle P \rangle \mid P \text{ halts for every input}\}$$

We already know: H_ϵ is undecidable, but recursively enumerable.
This implies: $\overline{H}_\epsilon$ is not recursively enumerable.

We will show:

Claim A: $\overline{H}_\epsilon \le \overline{H}_{\text{tot}}$

Claim B: $\overline{H}_\epsilon \le H_{\text{tot}}$

These two reductions will then together imply:

Theorem

Neither $\overline{H}_{\text{tot}}$ nor H_{tot} is recursively enumerable.

Claim A: $\overline{H_\epsilon} \leq \overline{H_{\mathrm{tot}}}$ Proof (1)

We describe a computable function f,
that maps YES-instances of $\overline{H_\epsilon}$ into YES-instances of $\overline{H_{\mathrm{tot}}}$ and
that maps NO-instances of $\overline{H_\epsilon}$ into NO-instances of $\overline{H_{\mathrm{tot}}}$.

Claim A: $\overline{H}_\epsilon \leq \overline{H}_{\text{tot}}$ Proof (1)

We describe a computable function f,
that maps YES-instances of $\overline{H}_\epsilon$ into YES-instances of $\overline{H}_{\text{tot}}$ and
that maps NO-instances of $\overline{H}_\epsilon$ into NO-instances of $\overline{H}_{\text{tot}}$.

Let w be an input for $\overline{H}_\epsilon$.

- If w is not of the form $\langle P \rangle$, then we set $f(w) = w$.
- If $w = \langle P \rangle$ for some C++ program P, then we set $f(w) := \langle P_\epsilon^* \rangle$, where the C++ program P_ϵ^* behaves as follows:

  P_ϵ^* ignores the input and simulates P on input ϵ.

The described function f is computable. (Why?)

Claim A: $\overline{H}_\epsilon \leq \overline{H}_{\text{tot}}$ Proof (2)

For the correctness, we will show:

(a) $w \in \overline{H}_\epsilon \;\Rightarrow\; f(w) \in \overline{H}_{\textbf{tot}}$

(b) $w \notin \overline{H}_\epsilon \;\Rightarrow\; f(w) \notin \overline{H}_{\text{tot}}$

Claim A: $\overline{H}_\epsilon \leq \overline{H}_{\text{tot}}$ Proof (2)

For the correctness, we will show:

(a) $w \in \overline{H}_\epsilon \;\Rightarrow\; f(w) \in \overline{H}_{\textbf{tot}}$

(b) $w \notin \overline{H}_\epsilon \;\Rightarrow\; f(w) \notin \overline{H}_{\text{tot}}$

If w is not a C++ program, then $w \in \overline{H}_\epsilon$ and $f(w) \in \overline{H}_{\text{tot}}$.
This subcase of (a) has been handled correctly.

Claim A: $\overline{H_\epsilon} \leq \overline{H_{\text{tot}}}$ Proof (2)

For the correctness, we will show:

(a) $w \in \overline{H_\epsilon} \;\Rightarrow\; f(w) \in \overline{H_{\textbf{tot}}}$

(b) $w \notin \overline{H_\epsilon} \;\Rightarrow\; f(w) \notin \overline{H_{\text{tot}}}$

If w is not a C++ program, then $w \in \overline{H_\epsilon}$ and $f(w) \in \overline{H_{\text{tot}}}$.
This subcase of (a) has been handled correctly.

If $w = \langle P \rangle$ for some C++ program P, then we consider $f(w) = \langle P_\epsilon^* \rangle$.
Then:

$$w \in \overline{H_\epsilon} \quad \Rightarrow \quad P \text{ does not halt on input } \epsilon$$

Claim A: $\overline{H}_\epsilon \le \overline{H}_{\mathrm{tot}}$ Proof (2)

For the correctness, we will show:

(a) $w \in \overline{H}_\epsilon \;\Rightarrow\; f(w) \in \overline{H}_{\mathbf{tot}}$

(b) $w \notin \overline{H}_\epsilon \;\Rightarrow\; f(w) \notin \overline{H}_{\mathrm{tot}}$

If w is not a C++ program, then $w \in \overline{H}_\epsilon$ and $f(w) \in \overline{H}_{\mathrm{tot}}$.
This subcase of (a) has been handled correctly.

If $w = \langle P \rangle$ for some C++ program P, then we consider $f(w) = \langle P^*_\epsilon \rangle$.
Then:

$$\begin{aligned} w \in \overline{H}_\epsilon \quad &\Rightarrow \quad P \text{ does not halt on input } \epsilon \\ &\Rightarrow \quad P^*_\epsilon \text{ does not halt on any input} \end{aligned}$$

Claim A: $\overline{H}_\epsilon \leq \overline{H}_{\text{tot}}$ — Proof (2)

For the correctness, we will show:

(a) $w \in \overline{H}_\epsilon \;\Rightarrow\; f(w) \in \overline{H}_{\textbf{tot}}$

(b) $w \notin \overline{H}_\epsilon \;\Rightarrow\; f(w) \notin \overline{H}_{\text{tot}}$

If w is not a C++ program, then $w \in \overline{H}_\epsilon$ and $f(w) \in \overline{H}_{\text{tot}}$.
This subcase of (a) has been handled correctly.

If $w = \langle P \rangle$ for some C++ program P, then we consider $f(w) = \langle P^*_\epsilon \rangle$.
Then:

$$\begin{aligned} w \in \overline{H}_\epsilon \quad &\Rightarrow \quad P \text{ does not halt on input } \epsilon \\ &\Rightarrow \quad P^*_\epsilon \text{ does not halt on any input} \\ &\Rightarrow \quad \langle P^*_\epsilon \rangle \notin H_{\text{tot}} \end{aligned}$$

Claim A: $\overline{H_\epsilon} \leq \overline{H}_{\text{tot}}$ Proof (2)

For the correctness, we will show:

(a) $w \in \overline{H}_\epsilon \;\Rightarrow\; f(w) \in \overline{H}_{\textbf{tot}}$

(b) $w \notin \overline{H}_\epsilon \;\Rightarrow\; f(w) \notin \overline{H}_{\text{tot}}$

If w is not a C++ program, then $w \in \overline{H}_\epsilon$ and $f(w) \in \overline{H}_{\text{tot}}$.
This subcase of (a) has been handled correctly.

If $w = \langle P \rangle$ for some C++ program P, then we consider $f(w) = \langle P^*_\epsilon \rangle$.
Then:

$$
\begin{aligned}
w \in \overline{H}_\epsilon \quad &\Rightarrow \quad P \text{ does not halt on input } \epsilon \\
&\Rightarrow \quad P^*_\epsilon \text{ does not halt on any input} \\
&\Rightarrow \quad \langle P^*_\epsilon \rangle \notin H_{\text{tot}} \\
&\Rightarrow \quad f(w) = \langle P^*_\epsilon \rangle \in \overline{H}_{\text{tot}} \qquad \text{and (a) is correct.}
\end{aligned}
$$

Claim A: $\overline{H_\epsilon} \leq \overline{H_{\text{tot}}}$ Proof (3)

For the correctness, we will show:

(a) $w \in \overline{H_\epsilon} \;\Rightarrow\; f(w) \in \overline{H_{\text{tot}}}$

(b) $w \notin \overline{H_\epsilon} \;\Rightarrow\; f(w) \notin \overline{H_{\mathbf{tot}}}$

Claim A: $\overline{H_\epsilon} \leq \overline{H_{\text{tot}}}$ Proof (3)

For the correctness, we will show:

(a) $w \in \overline{H_\epsilon} \;\Rightarrow\; f(w) \in \overline{H_{\text{tot}}}$

(b) $w \notin \overline{H_\epsilon} \;\Rightarrow\; f(w) \notin \overline{H_{\textbf{tot}}}$

If $w = \langle P \rangle$ for some C++ program P, then we consider $f(w) = \langle P^*_\epsilon \rangle$.
Then:

$$\begin{aligned} w \notin \overline{H_\epsilon} \quad &\Rightarrow \quad w \in H_\epsilon \\ &\Rightarrow \quad P \text{ halts on input } \epsilon \end{aligned}$$

Claim A: $\overline{H}_\epsilon \leq \overline{H}_{\text{tot}}$ Proof (3)

For the correctness, we will show:

(a) $w \in \overline{H}_\epsilon \;\Rightarrow\; f(w) \in \overline{H}_{\text{tot}}$

(b) $w \notin \overline{H}_\epsilon \;\Rightarrow\; f(w) \notin \overline{H}_{\textbf{tot}}$

If $w = \langle P \rangle$ for some C++ program P, then we consider $f(w) = \langle P^*_\epsilon \rangle$.
Then:

$$\begin{aligned} w \notin \overline{H}_\epsilon \quad &\Rightarrow \quad w \in H_\epsilon \\ &\Rightarrow \quad P \text{ halts on input } \epsilon \\ &\Rightarrow \quad P^*_\epsilon \text{ halts on every input} \end{aligned}$$

Claim A: $\overline{H}_\epsilon \leq \overline{H}_{\text{tot}}$ — Proof (3)

For the correctness, we will show:

(a) $w \in \overline{H}_\epsilon \;\Rightarrow\; f(w) \in \overline{H}_{\text{tot}}$

(b) $w \notin \overline{H}_\epsilon \;\Rightarrow\; f(w) \notin \overline{H}_{\textbf{tot}}$

If $w = \langle P \rangle$ for some C++ program P, then we consider $f(w) = \langle P^*_\epsilon \rangle$.
Then:

$$
\begin{aligned}
w \notin \overline{H}_\epsilon \quad &\Rightarrow \quad w \in H_\epsilon \\
&\Rightarrow \quad P \text{ halts on input } \epsilon \\
&\Rightarrow \quad P^*_\epsilon \text{ halts on every input} \\
&\Rightarrow \quad \langle P^*_\epsilon \rangle \in H_{\text{tot}}
\end{aligned}
$$

Claim A: $\overline{H_\epsilon} \leq \overline{H_{\text{tot}}}$ — Proof (3)

For the correctness, we will show:

(a) $w \in \overline{H_\epsilon} \;\Rightarrow\; f(w) \in \overline{H_{\text{tot}}}$

(b) $w \notin \overline{H_\epsilon} \;\Rightarrow\; f(w) \notin \overline{H_{\textbf{tot}}}$

If $w = \langle P \rangle$ for some C++ program P, then we consider $f(w) = \langle P_\epsilon^* \rangle$.
Then:

$$
\begin{aligned}
w \notin \overline{H_\epsilon} \quad &\Rightarrow \quad w \in H_\epsilon \\
&\Rightarrow \quad P \text{ halts on input } \epsilon \\
&\Rightarrow \quad P_\epsilon^* \text{ halts on every input} \\
&\Rightarrow \quad \langle P_\epsilon^* \rangle \in H_{\text{tot}} \\
&\Rightarrow \quad f(w) = \langle P_\epsilon^* \rangle \notin \overline{H_{\text{tot}}} \qquad \text{and (b) is correct.}
\end{aligned}
$$

Claim A: $\overline{H}_\epsilon \leq \overline{H}_{\text{tot}}$ Proof (3)

For the correctness, we will show:

(a) $w \in \overline{H}_\epsilon \;\Rightarrow\; f(w) \in \overline{H}_{\text{tot}}$

(b) $w \notin \overline{H}_\epsilon \;\Rightarrow\; f(w) \notin \overline{H}_{\textbf{tot}}$

If $w = \langle P \rangle$ for some C++ program P, then we consider $f(w) = \langle P_\epsilon^* \rangle$.
Then:

$$
\begin{aligned}
w \notin \overline{H}_\epsilon \quad &\Rightarrow \quad w \in H_\epsilon \\
&\Rightarrow \quad P \text{ halts on input } \epsilon \\
&\Rightarrow \quad P_\epsilon^* \text{ halts on every input} \\
&\Rightarrow \quad \langle P_\epsilon^* \rangle \in H_{\text{tot}} \\
&\Rightarrow \quad f(w) = \langle P_\epsilon^* \rangle \notin \overline{H}_{\text{tot}} \qquad \text{and (b) is correct.}
\end{aligned}
$$

This completes the proof of Claim A. □

Claim B: $\overline{H}_\epsilon \leq H_{\text{tot}}$ Proof (1)

We describe a computable function f,
that maps YES-instances of $\overline{H}_\epsilon$ into YES-instances of H_{tot} and
that maps NO-instances of $\overline{H}_\epsilon$ into NO-instances of H_{tot}.

Claim B: $\overline{H}_\epsilon \le H_{\text{tot}}$ Proof (1)

We describe a computable function f,
that maps YES-instances of $\overline{H}_\epsilon$ into YES-instances of H_{tot} and
that maps NO-instances of $\overline{H}_\epsilon$ into NO-instances of H_{tot}.

Let w be some input $\overline{H}_\epsilon$. Let w' be some word in H_{tot}.

- If w is not a C++ program, then we set $f(w) = w'$.
- If $w = \langle P \rangle$ for some C++ program P, then we set $f(w) := \langle P' \rangle$ where the C++ program P' behaves as follows on inputs of length ℓ:

  P' simulates the first ℓ steps of P on input ϵ.
  If P halts within these ℓ steps, then P' enters an endless-loop;
  otherwise P' halts.

The described function f is computable. (Why?)

Claim B: $\overline{H}_\epsilon \leq H_{\text{tot}}$ Proof (2)

For the correctness, we will show:

(a) $w \in \overline{H}_\epsilon \;\Rightarrow\; f(w) \in H_{\textbf{tot}}$

(b) $w \notin \overline{H}_\epsilon \;\Rightarrow\; f(w) \notin H_{\text{tot}}$

Claim B: $\overline{H}_\epsilon \leq H_{\text{tot}}$ — Proof (2)

For the correctness, we will show:

(a) $w \in \overline{H}_\epsilon \;\Rightarrow\; f(w) \in H_{\textbf{tot}}$

(b) $w \notin \overline{H}_\epsilon \;\Rightarrow\; f(w) \notin H_{\text{tot}}$

If w is not a C++ program, then $w \in \overline{H}_\epsilon$ and $f(w) = w' \in H_{\text{tot}}$.
This subcase of (a) has been handled correctly.

Claim B: $\overline{H_\epsilon} \leq H_{\text{tot}}$ Proof (2)

For the correctness, we will show:

(a) $w \in \overline{H_\epsilon} \;\Rightarrow\; f(w) \in H_{\textbf{tot}}$

(b) $w \notin \overline{H_\epsilon} \;\Rightarrow\; f(w) \notin H_{\text{tot}}$

If w is not a C++ program, then $w \in \overline{H_\epsilon}$ and $f(w) = w' \in H_{\text{tot}}$.
This subcase of (a) has been handled correctly.
If $w = \langle P \rangle$ for some C++ program P, then we consider $f(w) = \langle P' \rangle$.
Then:

$$w \in \overline{H_\epsilon} \quad \Rightarrow \quad P \text{ does not halt on input } \epsilon$$

Claim B: $\overline{H}_\epsilon \leq H_{\text{tot}}$ Proof (2)

For the correctness, we will show:

(a) $w \in \overline{H}_\epsilon \;\Rightarrow\; f(w) \in H_{\textbf{tot}}$

(b) $w \notin \overline{H}_\epsilon \;\Rightarrow\; f(w) \notin H_{\text{tot}}$

If w is not a C++ program, then $w \in \overline{H}_\epsilon$ and $f(w) = w' \in H_{\text{tot}}$.
This subcase of (a) has been handled correctly.
If $w = \langle P \rangle$ for some C++ program P, then we consider $f(w) = \langle P' \rangle$.
Then:

$$
\begin{aligned}
w \in \overline{H}_\epsilon \quad &\Rightarrow \quad P \text{ does not halt on input } \epsilon \\
&\Rightarrow \quad \neg\exists i\colon\ P \text{ halts within } i \text{ steps on input } \epsilon
\end{aligned}
$$

Claim B: $\overline{H_\epsilon} \leq H_{\text{tot}}$ Proof (2)

For the correctness, we will show:

(a) $w \in \overline{H_\epsilon} \;\Rightarrow\; f(w) \in H_{\textbf{tot}}$

(b) $w \notin \overline{H_\epsilon} \;\Rightarrow\; f(w) \notin H_{\text{tot}}$

If w is not a C++ program, then $w \in \overline{H_\epsilon}$ and $f(w) = w' \in H_{\text{tot}}$.
This subcase of (a) has been handled correctly.
If $w = \langle P \rangle$ for some C++ program P, then we consider $f(w) = \langle P' \rangle$.
Then:

$$
\begin{aligned}
w \in \overline{H_\epsilon} \quad &\Rightarrow \quad P \text{ does not halt on input } \epsilon \\
&\Rightarrow \quad \neg\exists i\colon\; P \text{ halts within } i \text{ steps on input } \epsilon \\
&\Rightarrow \quad \forall i\colon\; P \text{ does not halt within } i \text{ steps on input } \epsilon
\end{aligned}
$$

Claim B: $\overline{H_\epsilon} \leq H_{\text{tot}}$ Proof (2)

For the correctness, we will show:

(a) $w \in \overline{H_\epsilon} \;\Rightarrow\; f(w) \in H_{\textbf{tot}}$

(b) $w \notin \overline{H_\epsilon} \;\Rightarrow\; f(w) \notin H_{\text{tot}}$

If w is not a C++ program, then $w \in \overline{H_\epsilon}$ and $f(w) = w' \in H_{\text{tot}}$.
This subcase of (a) has been handled correctly.
If $w = \langle P \rangle$ for some C++ program P, then we consider $f(w) = \langle P' \rangle$.
Then:

$$
\begin{aligned}
w \in \overline{H_\epsilon} \quad &\Rightarrow \quad P \text{ does not halt on input } \epsilon \\
&\Rightarrow \quad \neg\exists i\colon\; P \text{ halts within } i \text{ steps on input } \epsilon \\
&\Rightarrow \quad \forall i\colon\; P \text{ does not halt within } i \text{ steps on input } \epsilon \\
&\Rightarrow \quad \forall i\colon\; P' \text{ halts on all inputs of length } i
\end{aligned}
$$

Claim B: $\overline{H}_\epsilon \leq H_{\text{tot}}$ Proof (2)

For the correctness, we will show:

(a) $w \in \overline{H}_\epsilon \;\Rightarrow\; f(w) \in H_{\textbf{tot}}$

(b) $w \notin \overline{H}_\epsilon \;\Rightarrow\; f(w) \notin H_{\text{tot}}$

If w is not a C++ program, then $w \in \overline{H}_\epsilon$ and $f(w) = w' \in H_{\text{tot}}$.
This subcase of (a) has been handled correctly.
If $w = \langle P \rangle$ for some C++ program P, then we consider $f(w) = \langle P' \rangle$.
Then:

$$
\begin{aligned}
w \in \overline{H}_\epsilon \quad &\Rightarrow \quad P \text{ does not halt on input } \epsilon \\
&\Rightarrow \quad \neg\exists i\colon\ P \text{ halts within } i \text{ steps on input } \epsilon \\
&\Rightarrow \quad \forall i\colon\ P \text{ does not halt within } i \text{ steps on input } \epsilon \\
&\Rightarrow \quad \forall i\colon\ P' \text{ halts on all inputs of length } i \\
&\Rightarrow \quad P' \text{ halts on every input}
\end{aligned}
$$

Claim B: $\overline{H_\epsilon} \leq H_{\text{tot}}$ — Proof (2)

For the correctness, we will show:

(a) $w \in \overline{H_\epsilon} \;\Rightarrow\; f(w) \in H_{\textbf{tot}}$

(b) $w \notin \overline{H_\epsilon} \;\Rightarrow\; f(w) \notin H_{\text{tot}}$

If w is not a C++ program, then $w \in \overline{H_\epsilon}$ and $f(w) = w' \in H_{\text{tot}}$.
This subcase of (a) has been handled correctly.
If $w = \langle P \rangle$ for some C++ program P, then we consider $f(w) = \langle P' \rangle$.
Then:

$$
\begin{aligned}
w \in \overline{H_\epsilon} \quad &\Rightarrow \quad P \text{ does not halt on input } \epsilon \\
&\Rightarrow \quad \neg\exists i\colon\; P \text{ halts within } i \text{ steps on input } \epsilon \\
&\Rightarrow \quad \forall i\colon\; P \text{ does not halt within } i \text{ steps on input } \epsilon \\
&\Rightarrow \quad \forall i\colon\; P' \text{ halts on all inputs of length } i \\
&\Rightarrow \quad P' \text{ halts on every input} \\
&\Rightarrow \quad f(w) = \langle P' \rangle \in H_{\text{tot}} \qquad \text{and (a) is correct.}
\end{aligned}
$$

Claim B: $\overline{H_\epsilon} \leq H_{\text{tot}}$ Proof (3)

For the correctness, we will show:

(a) $w \in \overline{H_\epsilon} \;\Rightarrow\; f(w) \in H_{\text{tot}}$

(b) $w \notin \overline{H_\epsilon} \;\Rightarrow\; f(w) \notin H_{\textbf{tot}}$

If $w = \langle P \rangle$ for some C++ program P, then we consider $f(w) = \langle P' \rangle$.
Then:

$$w \notin \overline{H_\epsilon} \;\Rightarrow\; P \text{ halts on input } \epsilon.$$

Claim B: $\overline{H_\epsilon} \leq H_{\text{tot}}$ — Proof (3)

For the correctness, we will show:

(a) $w \in \overline{H_\epsilon} \;\Rightarrow\; f(w) \in H_{\text{tot}}$

(b) $w \notin \overline{H_\epsilon} \;\Rightarrow\; f(w) \notin H_{\textbf{tot}}$

If $w = \langle P \rangle$ for some C++ program P, then we consider $f(w) = \langle P' \rangle$.
Then:

$$
\begin{aligned}
w \notin \overline{H_\epsilon} \;&\Rightarrow\; P \text{ halts on input } \epsilon. \\
&\Rightarrow\; \exists i\colon\ P \text{ halts within } i \text{ steps on input } \epsilon
\end{aligned}
$$

Claim B: $\overline{H_\epsilon} \leq H_{\text{tot}}$ — Proof (3)

For the correctness, we will show:

(a) $w \in \overline{H_\epsilon} \;\Rightarrow\; f(w) \in H_{\text{tot}}$

(b) $w \notin \overline{H_\epsilon} \;\Rightarrow\; f(w) \notin H_{\textbf{tot}}$

If $w = \langle P \rangle$ for some C++ program P, then we consider $f(w) = \langle P' \rangle$.
Then:

$$
\begin{aligned}
w \notin \overline{H_\epsilon} \;&\Rightarrow\; P \text{ halts on input } \epsilon. \\
&\Rightarrow\; \exists i\colon\; P \text{ halts within } i \text{ steps on input } \epsilon \\
&\Rightarrow\; \exists i\colon\; P' \text{ does not halt on any input of length} \geq i
\end{aligned}
$$

Claim B: $\overline{H}_\epsilon \le H_{\text{tot}}$ Proof (3)

For the correctness, we will show:

(a) $w \in \overline{H}_\epsilon \;\Rightarrow\; f(w) \in H_{\text{tot}}$

(b) $w \notin \overline{H}_\epsilon \;\Rightarrow\; f(w) \notin H_{\textbf{tot}}$

If $w = \langle P \rangle$ for some C++ program P, then we consider $f(w) = \langle P' \rangle$.
Then:

$$
\begin{aligned}
w \notin \overline{H}_\epsilon \;&\Rightarrow\; P \text{ halts on input } \epsilon. \\
&\Rightarrow\; \exists i\colon\; P \text{ halts within } i \text{ steps on input } \epsilon \\
&\Rightarrow\; \exists i\colon\; P' \text{ does not halt on any input of length} \ge i \\
&\Rightarrow\; P' \text{ does not halt on every input}
\end{aligned}
$$

Claim B: $\overline{H_\epsilon} \leq H_{\text{tot}}$ Proof (3)

For the correctness, we will show:

(a) $w \in \overline{H_\epsilon} \;\Rightarrow\; f(w) \in H_{\text{tot}}$

(b) $w \notin \overline{H_\epsilon} \;\Rightarrow\; f(w) \notin H_{\textbf{tot}}$

If $w = \langle P \rangle$ for some C++ program P, then we consider $f(w) = \langle P' \rangle$.
Then:

$$
\begin{aligned}
w \notin \overline{H_\epsilon} \;&\Rightarrow\; P \text{ halts on input } \epsilon. \\
&\Rightarrow\; \exists i\colon\; P \text{ halts within } i \text{ steps on input } \epsilon \\
&\Rightarrow\; \exists i\colon\; P' \text{ does not halt on any input of length} \geq i \\
&\Rightarrow\; P' \text{ does not halt on every input} \\
&\Rightarrow\; f(w) = \langle P' \rangle \notin H_{\text{tot}} \qquad \text{and (b) is correct.}
\end{aligned}
$$

Claim B: $\overline{H_\epsilon} \leq H_{\text{tot}}$ Proof (3)

For the correctness, we will show:

(a) $w \in \overline{H_\epsilon} \;\Rightarrow\; f(w) \in H_{\text{tot}}$

(b) $w \notin \overline{H_\epsilon} \;\Rightarrow\; f(w) \notin H_{\textbf{tot}}$

If $w = \langle P \rangle$ for some C++ program P, then we consider $f(w) = \langle P' \rangle$.
Then:

$$
\begin{aligned}
w \notin \overline{H_\epsilon} \;&\Rightarrow\; P \text{ halts on input } \epsilon. \\
&\Rightarrow\; \exists i\colon\; P \text{ halts within } i \text{ steps on input } \epsilon \\
&\Rightarrow\; \exists i\colon\; P' \text{ does not halt on any input of length} \geq i \\
&\Rightarrow\; P' \text{ does not halt on every input} \\
&\Rightarrow\; f(w) = \langle P' \rangle \notin H_{\text{tot}} \qquad \text{and (b) is correct.}
\end{aligned}
$$

This completes the proof of Claim B. □

Integration in closed form (1)

Example: Some indefinite integrals

$$\int 4x^3 + 3x^2 + 2x + 7 \, \mathrm{d}x \; = \; x^4 + x^3 + x^2 + 7x$$

Integration in closed form (1)

Example: Some indefinite integrals

$$\int 4x^3 + 3x^2 + 2x + 7 \; \mathrm{d}x \;=\; x^4 + x^3 + x^2 + 7x + C$$

Integration in closed form (1)

Example: Some indefinite integrals

$$\int 4x^3 + 3x^2 + 2x + 7 \, \mathrm{d}x = x^4 + x^3 + x^2 + 7x + C$$

$$\int x \sin(x) \, \mathrm{d}x =$$

Integration in closed form (1)

Example: Some indefinite integrals

$$\int 4x^3 + 3x^2 + 2x + 7 \,\mathrm{d}x = x^4 + x^3 + x^2 + 7x + C$$

$$\int x \sin(x) \,\mathrm{d}x = -x \cos(x) + \sin(x) + C$$

Integration in closed form (1)

Example: Some indefinite integrals

$$\int 4x^3 + 3x^2 + 2x + 7 \,\mathrm{d}x = x^4 + x^3 + x^2 + 7x + C$$

$$\int x \sin(x) \,\mathrm{d}x = -x\cos(x) + \sin(x) + C$$

$$\int \frac{\sin(x)}{x} \,\mathrm{d}x =$$

Integration in closed form (1)

Example: Some indefinite integrals

$$
\begin{aligned}
\int 4x^3 + 3x^2 + 2x + 7 \,\mathrm{d}x &= x^4 + x^3 + x^2 + 7x + C \\
\int x \sin(x) \,\mathrm{d}x &= -x\cos(x) + \sin(x) + C \\
\int \frac{\sin(x)}{x} \,\mathrm{d}x &= \;???
\end{aligned}
$$

Integration in closed form (1)

Example: Some indefinite integrals

$$\int 4x^3 + 3x^2 + 2x + 7 \,\mathrm{d}x = x^4 + x^3 + x^2 + 7x + C$$

$$\int x \sin(x) \,\mathrm{d}x = -x\cos(x) + \sin(x) + C$$

$$\int \frac{\sin(x)}{x} \,\mathrm{d}x = ???$$

$$\int e^{-x^2} \,\mathrm{d}x = ???$$

Integration in closed form (2)

Rational functions always are integrable in closed form:

Theorem (Liouville, 1838)

If a function $f(x)$ is the ratio of two polynomials $P(x)$ and $Q(x)$, then $f(x)$ can be written as the sum of several terms of the form

$$\frac{a}{(x-b)^n} \qquad \text{and} \qquad \frac{ax+b}{((x-c)^2+d^2)^n}.$$

Since every such term is integrable in closed form,
every rational function $f(x)$ is integrable in closed form.

Richardson's theorem

A function is called **elementary**, if it can be constructed by combining addition, subtraction, multiplication, division, exponentiation, taking roots, taking logarithms, taking absolute values, and trigonometric functions.

Richardson's theorem

A function is called **elementary**, if it can be constructed by combining
addition, subtraction, multiplication, division,
exponentiation, taking roots, taking logarithms, taking absolute values,
and trigonometric functions.

In 1968 the British mathematician Daniel Richardson proved the following undecidability result:

Richardson's theorem (1968)

It is undecidable, whether a given elementary function has an elementary antiderivative.

The proof reduces the halting problem (for Turing machines) to the integrability problem.

David Hilbert (1862–1943)

Wikipedia: David Hilbert was a German mathematician. He is recognized as one of the most influential and universal mathematicians of the 20th century. Hilbert discovered and developed a broad range of fundamental ideas in many areas, including invariant theory and the axiomatization of geometry.

In 1920 Hilbert proposed a research project that became known as Hilbert's program. He wanted mathematics to be formulated on a solid and complete logical foundation. He believed that in principle this could be done, by showing that (1) all of mathematics follows from a correctly chosen finite system of axioms; and (2) that one such axiom system is provably consistent.

Hilbert's tenth problem

At the International Congress of Mathematicians in Paris in 1900, David Hilbert presented a list with 23 mathematical problems.

Hilbert's tenth problem

At the International Congress of Mathematicians in Paris in 1900, David Hilbert presented a list with 23 mathematical problems.

Hilbert's tenth problem (original wording)

Given a Diophantine equation with any number of unknown quantities and with rational integral numerical coefficients: To devise a process according to which it can be determined in a finite number of operations whether the equation is solvable in rational integers.

- By "rational integers", Hilbert simply meant our integers in $\mathbb{Z}$
- A "Diophantine equation" is a polynomial equation in one or more variables

Diophantine equations

- A **term** is a product of variables and an integer coefficient.
  For instance

  $$6 \cdot x \cdot x \cdot x \cdot y \cdot z \cdot z \quad \text{bzw.} \quad 6x^3yz^2$$

  is a term over the variables x, y, z with the coefficient 6.

Diophantine equations

- A **term** is a product of variables and an integer coefficient. For instance

$$6 \cdot x \cdot x \cdot x \cdot y \cdot z \cdot z \quad \text{bzw.} \quad 6x^3yz^2$$

  is a term over the variables x, y, z with the coefficient 6.
- A **polynomial** is a sum of terms, as for instance

$$6x^3yz^2 + 3xy^2 - x^3 - 10$$

Diophantine equations

- A **term** is a product of variables and an integer coefficient.
  For instance

  $$6 \cdot x \cdot x \cdot x \cdot y \cdot z \cdot z \quad \text{bzw.} \quad 6x^3yz^2$$

  is a term over the variables x, y, z with the coefficient 6.
- A **polynomial** is a sum of terms, as for instance

  $$6x^3yz^2 + 3xy^2 - x^3 - 10$$

- A **Diophantine equation** sets a polynomial equal to zero.
  In other words, the solutions of the equation are the roots of the polynomial. The above polynomial has the root

  $$(x, y, z) = (5, 3, 0)$$

Examples (1)

Example 28

Is there an integer solution for the equation $x^3 + y^3 + z^3 = \mathbf{28}$?

Examples (1)

Example 28

Is there an integer solution for the equation $x^3 + y^3 + z^3 = \mathbf{28}$?

Yes, for example $(x, y, z) = (0, 1, 3)$

Examples (1)

Example 28

Is there an integer solution for the equation $x^3 + y^3 + z^3 = \mathbf{28}$?

Yes, for example $(x, y, z) = (0, 1, 3)$

Example 29

Is there an integer solution for the equation $x^3 + y^3 + z^3 = \mathbf{29}$?

Examples (1)

Example 28

Is there an integer solution for the equation $x^3 + y^3 + z^3 = \mathbf{28}$?

Yes, for example $(x, y, z) = (0, 1, 3)$

Example 29

Is there an integer solution for the equation $x^3 + y^3 + z^3 = \mathbf{29}$?

Yes, for example $(x, y, z) = (1, 1, 3)$

Examples (1)

Example 28

Is there an integer solution for the equation $x^3 + y^3 + z^3 = \mathbf{28}$?

Yes, for example $(x, y, z) = (0, 1, 3)$

Example 29

Is there an integer solution for the equation $x^3 + y^3 + z^3 = \mathbf{29}$?

Yes, for example $(x, y, z) = (1, 1, 3)$

Example 30

Is there an integer solution for the equation $x^3 + y^3 + z^3 = \mathbf{30}$?

Examples (1)

Example 28

Is there an integer solution for the equation $x^3 + y^3 + z^3 = \mathbf{28}$?

Yes, for example $(x, y, z) = (0, 1, 3)$

Example 29

Is there an integer solution for the equation $x^3 + y^3 + z^3 = \mathbf{29}$?

Yes, for example $(x, y, z) = (1, 1, 3)$

Example 30

Is there an integer solution for the equation $x^3 + y^3 + z^3 = \mathbf{30}$?

Yes, for example $(x, y, z) = (-283059965, -2218888517, 2220422932)$
(Discovered by: Beck, Mine, Tarrant & Yarbrough, 2007)

Examples (2)

Example 31

Is there an integer solution for the equation $x^3 + y^3 + z^3 = \mathbf{31}$?

Examples (2)

Example 31

Is there an integer solution for the equation $x^3 + y^3 + z^3 = \mathbf{31}$?

No!!

Examples (2)

Example 31

Is there an integer solution for the equation $x^3 + y^3 + z^3 = \mathbf{31}$?

No!!

- Modulo 9 an integer n can only take the remainders $0, \pm 1, \pm 2, \pm 3, \pm 4$.

Examples (2)

Example 31

Is there an integer solution for the equation $x^3 + y^3 + z^3 = \mathbf{31}$?

No!!

- Modulo 9 an integer n can only take the remainders $0, \pm 1, \pm 2, \pm 3, \pm 4$.
- Modulo 9 a cube n^3 can only take the remainders $0^3, (\pm 1)^3, (\pm 2)^3, (\pm 3)^3, (\pm 4)^3$.

Examples (2)

Example 31

Is there an integer solution for the equation $x^3 + y^3 + z^3 = \mathbf{31}$?

No!!

- Modulo 9 an integer n can only take the remainders $0, \pm 1, \pm 2, \pm 3, \pm 4$.
- Modulo 9 a cube n^3 can only take the remainders $0^3, (\pm 1)^3, (\pm 2)^3, (\pm 3)^3, (\pm 4)^3$.
- Modulo 9 a cube n^3 can only take the remainders $0, \pm 1$.

Examples (2)

Example 31

Is there an integer solution for the equation $x^3 + y^3 + z^3 = \mathbf{31}$?

No!!

- Modulo 9 an integer n can only take the remainders $0, \pm 1, \pm 2, \pm 3, \pm 4$.
- Modulo 9 a cube n^3 can only take the remainders $0^3, (\pm 1)^3, (\pm 2)^3, (\pm 3)^3, (\pm 4)^3$.
- Modulo 9 a cube n^3 can only take the remainders $0, \pm 1$.

- Modulo 9 a sum $x^3 + y^3 + z^3$ of three cubes can only take the remainders $0, \pm 1, \pm 2, \pm 3$, but never the remainders ± 4.
- Since $31 \equiv 4 \pmod 9$, there is no integer solution to this case.

Examples (3)

Example 32

Is there an integer solution for the equation $x^3 + y^3 + z^3 = \mathbf{32}$?

Examples (3)

Example 32

Is there an integer solution for the equation $x^3 + y^3 + z^3 = \mathbf{32}$?

No! Since $32 \equiv -4 \pmod 9$, there is no integer solution to this case.

Examples (3)

Example 32

Is there an integer solution for the equation $x^3 + y^3 + z^3 = \mathbf{32}$?

No! Since $32 \equiv -4 \pmod 9$, there is no integer solution to this case.

Example 33

Is there an integer solution for the equation $x^3 + y^3 + z^3 = \mathbf{33}$?

Examples (3)

Example 32

Is there an integer solution for the equation $x^3 + y^3 + z^3 = \mathbf{32}$?

No! Since $32 \equiv -4 \pmod 9$, there is no integer solution to this case.

Example 33

Is there an integer solution for the equation $x^3 + y^3 + z^3 = \mathbf{33}$?

Nobody knows. Open problem. No solutions with $|x|, |y|, |z| \leq 10^{12}$.

Examples (3)

Example 32

Is there an integer solution for the equation $x^3 + y^3 + z^3 = \mathbf{32}$?

No! Since $32 \equiv -4 \pmod 9$, there is no integer solution to this case.

Example 33

Is there an integer solution for the equation $x^3 + y^3 + z^3 = \mathbf{33}$?

Nobody knows. Open problem. No solutions with $|x|, |y|, |z| \leq 10^{12}$.
That's actually the old answer (valid till February 2019).

The new answer (valid since March 2019) is:
Yes: $(8866128975287528, -8778405442862239, -2736111468807040)$
Discovered in March 2019 by Andrew Booker (University of Bristol)

Examples (4)

Some Turing-decidable examples

- Quadratic Equation $5x^2 - 3x + 6 = 0$.
  Can be solved by highschool methods. Hence it is easy to check whether the equation has integer solutions.

Examples (4)

Some Turing-decidable examples

- Quadratic Equation $5x^2 - 3x + 6 = 0$.
  Can be solved by highschool methods. Hence it is easy to check whether the equation has integer solutions.
- Diophantine equations in a single variable x:

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_2 x^2 + a_1 x + a_0 = 0$$

  Each integer solution x is a divisor of a_0.
  Hence it is sufficient to work through all integers x with $|x| \leq |a_0|$.

Examples (4)

Some Turing-decidable examples

- Quadratic Equation $5x^2 - 3x + 6 = 0$.
  Can be solved by highschool methods. Hence it is easy to check whether the equation has integer solutions.
- Diophantine equations in a single variable x:

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_2 x^2 + a_1 x + a_0 = 0$$

  Each integer solution x is a divisor of a_0.
  Hence it is sufficient to work through all integers x with $|x| \leq |a_0|$.
- Fermat's equation $x^n + y^n = z^n$ with $n \geq 3$.
  This Diophantine equation has no positive integers solutions.
  (Fermat's last theorem; theorem of Sir Andrew Wiles)

Formulation as decision problem

Hilbert's tenth problem (original wording)

Given a Diophantine equation with any number of unknown quantities and with rational integral numerical coefficients: To devise a process according to which it can be determined in a finite number of operations whether the equation is solvable in rational integers.

Hilbert's tenth problem (modern formulation)

Describe an algorithm that decides, whether a given polynomial with integer coefficients has an integer root.

Formulation as decision problem

Hilbert's tenth problem (original wording)

Given a Diophantine equation with any number of unknown quantities and with rational integral numerical coefficients: To devise a process according to which it can be determined in a finite number of operations whether the equation is solvable in rational integers.

Hilbert's tenth problem (modern formulation)

Describe an algorithm that decides, whether a given polynomial with integer coefficients has an integer root.

Hilbert's tenth problem (in our language)

Describe a C++ program that decides the following language:

$$\text{Dioph} = \{ \langle p \rangle \mid p \text{ is a polynomial with integer coefficients and with an integer root} \}$$

Recursive enumerability of Dioph

For a polynomial p in ℓ variables,
the range of p corresponds to the countably infinite language $\mathbb{Z}^{\ell}$.

Recursive enumerability of Dioph

For a polynomial p in ℓ variables,
the range of p corresponds to the countably infinite language $\mathbb{Z}^\ell$.

The following algorithm recognizes Dioph:

- Enumerate the ℓ-tuples in $\mathbb{Z}^\ell$ in their canonical ordering and evaluate p for each such tuple.
- Accept, as soon as one of these evaluations yields the value 0.

Recursive enumerability of Dioph

For a polynomial p in ℓ variables,
the range of p corresponds to the countably infinite language $\mathbb{Z}^\ell$.

The following algorithm recognizes Dioph:

- Enumerate the ℓ-tuples in $\mathbb{Z}^\ell$ in their canonical ordering and evaluate p for each such tuple.
- Accept, as soon as one of these evaluations yields the value 0.

We conclude:

Theorem

The language Dioph is recursively enumerable.

Decidability of Dioph (1)

- If we had a hard upper bound on the absolute values of the roots, then we could simply enumerate all the finite set of ℓ-tuples whose components satisfy this upper bound. This would make problem Dioph decidable.

Decidability of Dioph (1)

- If we had a hard upper bound on the absolute values of the roots, then we could simply enumerate all the finite set of ℓ-tuples whose components satisfy this upper bound. This would make problem Dioph decidable.
- We have seen:
  For polynomials $p(x) = a_k x^k + a_{k-1}x^{k-1} + \cdots + a_1 x + a_0$ in a single variable such an upper bound is given by $|a_0|$.

Decidability of Dioph (1)

- If we had a hard upper bound on the absolute values of the roots, then we could simply enumerate all the finite set of ℓ-tuples whose components satisfy this upper bound. This would make problem Dioph decidable.
- We have seen:
  For polynomials $p(x) = a_k x^k + a_{k-1}x^{k-1} + \cdots + a_1 x + a_0$ in a single variable such an upper bound is given by $|a_0|$.
- For polynomials in two or more variables, however, there are no such upper bounds on the absolute values of the roots: Consider for example the polynomial $x + y$.

Decidability of Dioph (1)

- If we had a hard upper bound on the absolute values of the roots, then we could simply enumerate all the finite set of ℓ-tuples whose components satisfy this upper bound. This would make problem Dioph decidable.
- We have seen:
  For polynomials $p(x) = a_k x^k + a_{k-1}x^{k-1} + \cdots + a_1 x + a_0$ in a single variable such an upper bound is given by $|a_0|$.
- For polynomials in two or more variables, however, there are no such upper bounds on the absolute values of the roots: Consider for example the polynomial $x + y$.
- On the other hand: We wouldn't even need such a strong bound that is valid for **all** the roots. It would be sufficient, if a **single one** of the roots would be bounded.

Decidability of Dioph (1)

- If we had a hard upper bound on the absolute values of the roots, then we could simply enumerate all the finite set of ℓ-tuples whose components satisfy this upper bound. This would make problem Dioph decidable.
- We have seen:
  For polynomials $p(x) = a_k x^k + a_{k-1} x^{k-1} + \cdots + a_1 x + a_0$ in a single variable such an upper bound is given by $|a_0|$.
- For polynomials in two or more variables, however, there are no such upper bounds on the absolute values of the roots: Consider for example the polynomial $x + y$.
- On the other hand: We wouldn't even need such a strong bound that is valid for **all** the roots. It would be sufficient, if a **single one** of the roots would be bounded.
- Does such an upper bound exist?
- Or is there perhaps a completely different approach for attacking Hilbert's tenth problem?

Decidability of Dioph (2)

It took seventy years to answer Hilbert's tenth problem. The Russian mathematician Yuri Matijasevich solved it with the following theorem:

Theorem of Matijasevich (1970)

The problem, whether a given polynomial with integer coefficients possesses an integer root, ist undecidable.

Decidability of Dioph (2)

It took seventy years to answer Hilbert's tenth problem. The Russian mathematician Yuri Matijasevich solved it with the following theorem:

Theorem of Matijasevich (1970)

The problem, whether a given polynomial with integer coefficients possesses an integer root, ist undecidable.

- The proof is based on a long chain of reductions, that altogether reduces the halting problem H to the integer root problem Dioph.
- Yuri Matijasevich completed the last piece of that chain.
- Other important pieces of that chain had been constructed before by Martin Davis, Julia Robinson and Hilary Putnam, in the years 1950–1970.

Decidability of Dioph (3)

In fact, the proof chain yields a stronger result:

Theorem (Davis, Robinson, Putnam & Matijasevich)

For every subset $Y \subseteq \mathbb{Z}$ of the integers, the following two statements are equivalent:

- Y is recursively enumerable
- There exists a $(k+1)$-variate polynomial $p(x_1, \ldots, x_k, y)$ with integer coefficients so that $Y = \{y \in \mathbb{Z} \mid \exists x_1, \ldots, x_k \in \mathbb{Z} \text{ mit } p(x_1, \ldots, x_k, y) = 0\}$

Decidability of Dioph (3)

In fact, the proof chain yields a stronger result:

Theorem (Davis, Robinson, Putnam & Matijasevich)

For every subset $Y \subseteq \mathbb{Z}$ of the integers, the following two statements are equivalent:

- Y is recursively enumerable
- There exists a $(k+1)$-variate polynomial $p(x_1, \ldots, x_k, y)$ with integer coefficients so that $Y = \{y \in \mathbb{Z} \mid \exists x_1, \ldots, x_k \in \mathbb{Z} \text{ mit } p(x_1, \ldots, x_k, y) = 0\}$

- Hence integer polynomials are equally powerful as C++ programs and as Turing machines.
- The proof is technical, difficult, and long (and hence omitted).

The computability landscape