The Grand Unified Theory of Computation

Vahan Mkrtchyan [1]

[1] Lane Department of Computer Science and Electrical Engineering
West Virginia University

April 20, 2015

Outline

1. Babbage's Vision and Hilbert's Dream

Outline

1. Babbage's Vision and Hilbert's Dream
2. Universality and Undecidability

Outline

1. Babbage's Vision and Hilbert's Dream
2. Universality and Undecidability
3. Building Blocks: Recursive Functions

Problems

Problems

Formulation of an Algorithmic Problem

Problems

Formulation of an Algorithmic Problem

In a typical algorithmic problem (decision problem), we are given a certain input x,

Problems

Formulation of an Algorithmic Problem

In a typical algorithmic problem (decision problem), we are given a certain input x, and we are asked to check if a certain property P is true.

Problems

Formulation of an Algorithmic Problem

In a typical algorithmic problem (decision problem), we are given a certain input x, and we are asked to check if a certain property P is true.

TSP

Problems

Formulation of an Algorithmic Problem

In a typical algorithmic problem (decision problem), we are given a certain input x, and we are asked to check if a certain property P is true.

TSP

Given a complete graph K_n, together with an edge-weight function $c : E(K_n) \longrightarrow N$ and a bound B,

Problems

Formulation of an Algorithmic Problem

In a typical algorithmic problem (decision problem), we are given a certain input x, and we are asked to check if a certain property P is true.

TSP

Given a complete graph K_n, together with an edge-weight function $c : E(K_n) \longrightarrow N$ and a bound B, the goal is to check whether there is a Hamiltonian cycle of weight at most B.

Algorithms

Algorithms

An Algorithm solving the Problem

Algorithms

An Algorithm solving the Problem

In this course, we have dealt with problems that are solvable with some algorithm.

Algorithms

An Algorithm solving the Problem

In this course, we have dealt with problems that are solvable with some algorithm.

We have addressed the issue of solving these problems efficiently, or showing that this kind of algorithms may not exist

Algorithms

An Algorithm solving the Problem

In this course, we have dealt with problems that are solvable with some algorithm.

We have addressed the issue of solving these problems efficiently, or showing that this kind of algorithms may not exist (**NP-completeness, NP-hardness**).

Algorithms

An Algorithm solving the Problem

In this course, we have dealt with problems that are solvable with some algorithm.

We have addressed the issue of solving these problems efficiently, or showing that this kind of algorithms may not exist (**NP-completeness, NP-hardness**).

What is the algorithm that solves TSP?

Algorithms

An Algorithm solving the Problem

In this course, we have dealt with problems that are solvable with some algorithm.

We have addressed the issue of solving these problems efficiently, or showing that this kind of algorithms may not exist (**NP-completeness, NP-hardness**).

What is the algorithm that solves TSP? How many Hamiltonian cycles K_n has?

Algorithms

An Algorithm solving the Problem

In this course, we have dealt with problems that are solvable with some algorithm.

We have addressed the issue of solving these problems efficiently, or showing that this kind of algorithms may not exist (**NP-completeness, NP-hardness**).

What is the algorithm that solves TSP? How many Hamiltonian cycles K_n has?

What is the description of the algorithm that solves the general decision problem?

Linear Programming

Linear Programming

Does the same trick imply that Linear Programming is solvable?

Linear Programming

Does the same trick imply that Linear Programming is solvable?

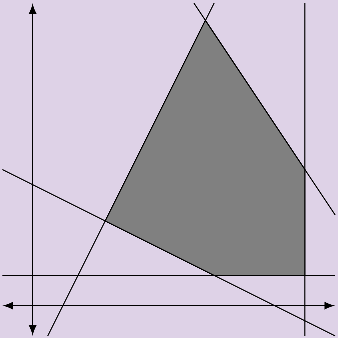

Figure: The feasible region

Question

Question

Are there algorithmic problems that are unsolvable?

Question

Are there algorithmic problems that are unsolvable?

Definition

An algorithmic problem is *decidable* or *computable* or *solvable*, if there is an algorithm that solves it in some finite amount of time.

Question

Are there algorithmic problems that are unsolvable?

Definition

An algorithmic problem is *decidable* or *computable* or *solvable*, if there is an algorithm that solves it in some finite amount of time.

Remark

We place no bounds whatsoever on how long the algorithm takes,

Question

Are there algorithmic problems that are unsolvable?

Definition

An algorithmic problem is *decidable* or *computable* or *solvable*, if there is an algorithm that solves it in some finite amount of time.

Remark

We place no bounds whatsoever on how long the algorithm takes, we just know that it will halt eventually.

Babbage's ideas

Babbage's ideas

Babbage was probably the first, who attempted to construct a mechanical computer.

Babbage's ideas

Babbage was probably the first, who attempted to construct a mechanical computer.

A mechanical device that could calculate the value of a polynomial at any point.

Babbage's ideas

Babbage was probably the first, who attempted to construct a mechanical computer.

A mechanical device that could calculate the value of a polynomial at any point.

Since he was aware of Taylor series, he was expecting to compute the value of any function approximately.

The Foundations of Mathematics

The Foundations of Mathematics

Hilbert's ideas

The Foundations of Mathematics

Hilbert's ideas

Mathematicians from Euclid to Gauss have been thinking about algorithms for millennia.

The Foundations of Mathematics

Hilbert's ideas

Mathematicians from Euclid to Gauss have been thinking about algorithms for millennia.

The idea of algorithms as well-defined mathematical objects, worthy of investigation in and of themselves, did not emerge until the dawn of the 20th century.

The Foundations of Mathematics

Hilbert's ideas

Mathematicians from Euclid to Gauss have been thinking about algorithms for millennia.

The idea of algorithms as well-defined mathematical objects, worthy of investigation in and of themselves, did not emerge until the dawn of the 20th century.

In 1900, David Hilbert delivered an address to the International Congress of Mathematicians, and asked for the solution of the following problem:

The Foundations of Mathematics

Hilbert's ideas

Mathematicians from Euclid to Gauss have been thinking about algorithms for millennia.

The idea of algorithms as well-defined mathematical objects, worthy of investigation in and of themselves, did not emerge until the dawn of the 20th century.

In 1900, David Hilbert delivered an address to the International Congress of Mathematicians, and asked for the solution of the following problem:

Problem

Specify a procedure which, in a finite number of operations, enables one to determine whether a given Diophantine equation (a polynomial equation with integer coefficients) with an arbitrary number of variables has an integer solution.

An example

An example

A Diophantine equation

An example

A Diophantine equation

$$3 \cdot x^2 \cdot y^4 \cdot z^6 + 13 \cdot x \cdot y \cdot z^2 - 53 \cdot x^4 \cdot y^3 \cdot z^4 + 12 \cdot x + 15 \cdot z - 3 = 0.$$

An example

A Diophantine equation

$$3 \cdot x^2 \cdot y^4 \cdot z^6 + 13 \cdot x \cdot y \cdot z^2 - 53 \cdot x^4 \cdot y^3 \cdot z^4 + 12 \cdot x + 15 \cdot z - 3 = 0.$$

A consequence

An example

A Diophantine equation

$$3 \cdot x^2 \cdot y^4 \cdot z^6 + 13 \cdot x \cdot y \cdot z^2 - 53 \cdot x^4 \cdot y^3 \cdot z^4 + 12 \cdot x + 15 \cdot z - 3 = 0.$$

A consequence

Were there such an algorithm, we could have asked it to solve Fermat's Last Theorem for each fixed value of $n \geq 3$:

An example

A Diophantine equation

$$3 \cdot x^2 \cdot y^4 \cdot z^6 + 13 \cdot x \cdot y \cdot z^2 - 53 \cdot x^4 \cdot y^3 \cdot z^4 + 12 \cdot x + 15 \cdot z - 3 = 0.$$

A consequence

Were there such an algorithm, we could have asked it to solve Fermat's Last Theorem for each fixed value of $n \geq 3$:

$$x^n + y^n = z^n.$$

Hilbert's optimism

Hilbert's optimism

Entscheidungsproblem

Hilbert's optimism

Entscheidungsproblem

Hilbert showed even more optimism about the power of algorithms in 1928, when he challenged his fellow mathematicians with the Entscheidungsproblem (in English, the decision problem):

Hilbert's optimism

Entscheidungsproblem

Hilbert showed even more optimism about the power of algorithms in 1928, when he challenged his fellow mathematicians with the Entscheidungsproblem (in English, the decision problem):

Problem

The Entscheidungsproblem is solved if one knows a procedure that allows one to decide the validity of a given logical expression by a finite number of operations.

An example

An example

A consequence

An example

A consequence

Were there such an algorithm, we could have asked it to decide whether Fermat's Last Theorem is true:

An example

A consequence

Were there such an algorithm, we could have asked it to decide whether Fermat's Last Theorem is true:

$$\exists x, y, z \in Z \backslash \{0\}, x^n + y^n = z^n.$$

Mathematics and its axioms

Mathematics and its axioms

Axiomatic Systems

Mathematics and its axioms

Axiomatic Systems

Where one needs to prove this statement?

Mathematics and its axioms

Axiomatic Systems

Where one needs to prove this statement? Mathematics?

Mathematics and its axioms

Axiomatic Systems

Where one needs to prove this statement? Mathematics?

What are the axioms of Mathematics and what are the inference rules?

Mathematics and its axioms

Axiomatic Systems

Where one needs to prove this statement? Mathematics?

What are the axioms of Mathematics and what are the inference rules?

Mathematicians have been proving theorems without asking these questions since Ancient Greek.

Mathematics and its axioms

Axiomatic Systems

Where one needs to prove this statement? Mathematics?

What are the axioms of Mathematics and what are the inference rules?

Mathematicians have been proving theorems without asking these questions since Ancient Greek.

Only in the end of 19th, and in the beginning of 20th century, mathematicians started to think in the direction of building an axiomatic foundation for mathematics.

Mathematics and its axioms

Axiomatic Systems

Where one needs to prove this statement? Mathematics?

What are the axioms of Mathematics and what are the inference rules?

Mathematicians have been proving theorems without asking these questions since Ancient Greek.

Only in the end of 19th, and in the beginning of 20th century, mathematicians started to think in the direction of building an axiomatic foundation for mathematics.

Their goal was to reduce all of mathematics to set theory and logic, creating a formal system powerful enough to prove all the mathematical facts we know.

Mathematics and its axioms

Axiomatic Systems

Where one needs to prove this statement? Mathematics?

What are the axioms of Mathematics and what are the inference rules?

Mathematicians have been proving theorems without asking these questions since Ancient Greek.

Only in the end of 19th, and in the beginning of 20th century, mathematicians started to think in the direction of building an axiomatic foundation for mathematics.

Their goal was to reduce all of mathematics to set theory and logic, creating a formal system powerful enough to prove all the mathematical facts we know.

At the turn of the century, several paradoxes shook these foundations, showing that a naive approach to set theory could lead to contradictions.

Paradoxes in Mathematics

Paradoxes in Mathematics

Russell's paradox

Paradoxes in Mathematics

Russell's paradox

Sets can be elements of other sets,

Paradoxes in Mathematics

Russell's paradox

Sets can be elements of other sets, for instance, consider the set of all intervals on the real line, each of which is a set of real numbers.

Paradoxes in Mathematics

Russell's paradox

Sets can be elements of other sets, for instance, consider the set of all intervals on the real line, each of which is a set of real numbers.

So, it seems reasonable to ask which sets are elements of themselves, and which are not.

Paradoxes in Mathematics

Russell's paradox

Sets can be elements of other sets, for instance, consider the set of all intervals on the real line, each of which is a set of real numbers.

So, it seems reasonable to ask which sets are elements of themselves, and which are not.

Consider the set R defined as follows:

Paradoxes in Mathematics

Russell's paradox

Sets can be elements of other sets, for instance, consider the set of all intervals on the real line, each of which is a set of real numbers.

So, it seems reasonable to ask which sets are elements of themselves, and which are not.

Consider the set R defined as follows:

$$R = \{S : S \notin S\}.$$

Paradoxes in Mathematics

Russell's paradox

Sets can be elements of other sets, for instance, consider the set of all intervals on the real line, each of which is a set of real numbers.

So, it seems reasonable to ask which sets are elements of themselves, and which are not.

Consider the set R defined as follows:

$$R = \{S : S \notin S\}.$$

Remark

It can be easily seen that

$$R \in R \text{ if and only if } R \notin R.$$

Paradoxes in Mathematics

Paradoxes in Mathematics

A number theoretic paradox

Paradoxes in Mathematics

A number theoretic paradox

In order to specify a natural number $n \geq 1$, we need some number of words in English.

Paradoxes in Mathematics

A number theoretic paradox

In order to specify a natural number $n \geq 1$, we need some number of words in English.

For each $n \geq 1$, there exists a smallest number $h(n)$, so that any specification of n requires at least $h(n)$ words in English.

Paradoxes in Mathematics

A number theoretic paradox

In order to specify a natural number $n \geq 1$, we need some number of words in English.

For each $n \geq 1$, there exists a smallest number $h(n)$, so that any specification of n requires at least $h(n)$ words in English.

Consider the smallest number k which requires at least 1000 words for its specification.

Universal Programs and Interpreters

Universal Programs and Interpreters

Universal Programs

Universal Programs and Interpreters

Universal Programs

The most basic fact about modern computers is their universality.

Universal Programs and Interpreters

Universal Programs

The most basic fact about modern computers is their universality.

They can carry out any program we give to them.

Universal Programs and Interpreters

Universal Programs

The most basic fact about modern computers is their universality.

They can carry out any program we give to them.

In particular, there are programs that run other programs.

Universal Programs and Interpreters

Universal Programs

The most basic fact about modern computers is their universality.

They can carry out any program we give to them.

In particular, there are programs that run other programs.

A computer's operating system is a program that runs and manages many programs at once.

Universal Programs and Interpreters

Universal Programs

The most basic fact about modern computers is their universality.

They can carry out any program we give to them.

In particular, there are programs that run other programs.

A computer's operating system is a program that runs and manages many programs at once.

Interpreters

Universal Programs and Interpreters

Universal Programs

The most basic fact about modern computers is their universality.

They can carry out any program we give to them.

In particular, there are programs that run other programs.

A computer's operating system is a program that runs and manages many programs at once.

Interpreters

In any programming language, one can write an *interpreter* or *universal program*,

Universal Programs and Interpreters

Universal Programs

The most basic fact about modern computers is their universality.

They can carry out any program we give to them.

In particular, there are programs that run other programs.

A computer's operating system is a program that runs and manages many programs at once.

Interpreters

In any programming language, one can write an *interpreter* or *universal program*, a program that takes the source code of another program as input,

Universal Programs and Interpreters

Universal Programs

The most basic fact about modern computers is their universality.

They can carry out any program we give to them.

In particular, there are programs that run other programs.

A computer's operating system is a program that runs and manages many programs at once.

Interpreters

In any programming language, one can write an *interpreter* or *universal program*, a program that takes the source code of another program as input, and runs it step-by-step, keeping track of its variables and which instruction to perform next.

Universal Programs and Interpreters

Universal Programs

The most basic fact about modern computers is their universality.

They can carry out any program we give to them.

In particular, there are programs that run other programs.

A computer's operating system is a program that runs and manages many programs at once.

Interpreters

In any programming language, one can write an *interpreter* or *universal program*, a program that takes the source code of another program as input, and runs it step-by-step, keeping track of its variables and which instruction to perform next.

Symbolically, we can define this universal program like this:

$$U(\Pi, x) = \Pi(x).$$

Diagonalization and Halting

Diagonalization and Halting

Some programs cannot halt

Diagonalization and Halting

Some programs cannot halt

Let $U(\Pi, x)$ be a universal program.

Diagonalization and Halting

Some programs cannot halt

Let $U(\Pi, x)$ be a universal program. Consider the special case where $x = \Pi$. Then, we will have the following:

$$U(\Pi, \Pi) = \Pi(\Pi).$$

Diagonalization and Halting

Some programs cannot halt

Let $U(\Pi, x)$ be a universal program. Consider the special case where $x = \Pi$. Then, we will have the following:

$$U(\Pi, \Pi) = \Pi(\Pi).$$

Now suppose, for simplicity, that the programs in question returns a Boolean value, true or false.

Diagonalization and Halting

Some programs cannot halt

Let $U(\Pi, x)$ be a universal program. Consider the special case where $x = \Pi$. Then, we will have the following:

$$U(\Pi, \Pi) = \Pi(\Pi).$$

Now suppose, for simplicity, that the programs in question returns a Boolean value, true or false. Then we can define a new program V which runs Π on itself, and negates the result:

$$V(\Pi) = \overline{\Pi(\Pi)}.$$

Diagonalization and Halting

Some programs cannot halt

Let $U(\Pi, x)$ be a universal program. Consider the special case where $x = \Pi$. Then, we will have the following:

$$U(\Pi, \Pi) = \Pi(\Pi).$$

Now suppose, for simplicity, that the programs in question returns a Boolean value, true or false. Then we can define a new program V which runs Π on itself, and negates the result:

$$V(\Pi) = \overline{\Pi(\Pi)}.$$

Now, if we feed V its own source code, an apparent contradiction arises, since

$$V(V) = \overline{V(V)}.$$

Diagonalization and Halting

Some programs cannot halt

Let $U(\Pi, x)$ be a universal program. Consider the special case where $x = \Pi$. Then, we will have the following:

$$U(\Pi, \Pi) = \Pi(\Pi).$$

Now suppose, for simplicity, that the programs in question returns a Boolean value, true or false. Then we can define a new program V which runs Π on itself, and negates the result:

$$V(\Pi) = \overline{\Pi(\Pi)}.$$

Now, if we feed V its own source code, an apparent contradiction arises, since

$$V(V) = \overline{V(V)}.$$

The only way to resolve this paradox is if $V(V)$ is undefined.

Diagonalization and Halting

Some programs cannot halt

Let $U(\Pi, x)$ be a universal program. Consider the special case where $x = \Pi$. Then, we will have the following:

$$U(\Pi, \Pi) = \Pi(\Pi).$$

Now suppose, for simplicity, that the programs in question returns a Boolean value, true or false. Then we can define a new program V which runs Π on itself, and negates the result:

$$V(\Pi) = \overline{\Pi(\Pi)}.$$

Now, if we feed V its own source code, an apparent contradiction arises, since

$$V(V) = \overline{V(V)}.$$

The only way to resolve this paradox is if $V(V)$ is undefined. In other words, when given its own source code as input, V runs forever, and never returns any output.

Diagonalization and Halting

Diagonalization and Halting

Universality implies non-halting programs

Diagonalization and Halting

Universality implies non-halting programs

This shows that any programming language powerful enough to express a universal program

Diagonalization and Halting

Universality implies non-halting programs

This shows that any programming language powerful enough to express a universal program possesses programs that never halt, at least when given certain inputs.

Diagonalization and Halting

Universality implies non-halting programs

This shows that any programming language powerful enough to express a universal program possesses programs that never halt, at least when given certain inputs.

In brief, universality implies non-halting programs.

Diagonalization and Halting

Universality implies non-halting programs

This shows that any programming language powerful enough to express a universal program possesses programs that never halt, at least when given certain inputs.

In brief, universality implies non-halting programs.

Thus any reasonable definition of computable functions includes partial functions, which are undefined for some values of their input,

Diagonalization and Halting

Universality implies non-halting programs

This shows that any programming language powerful enough to express a universal program possesses programs that never halt, at least when given certain inputs.

In brief, universality implies non-halting programs.

Thus any reasonable definition of computable functions includes partial functions, which are undefined for some values of their input, in addition to total ones, which are always well-defined.

Diagonalization and Cantor

Diagonalization and Cantor

Definition

Two sets A and B are said to be equicardinal, if there is a one-to-one mapping between the elements of these sets.

Diagonalization and Cantor

Definition

Two sets A and B are said to be equicardinal, if there is a one-to-one mapping between the elements of these sets.

Equicardinal Sets

Diagonalization and Cantor

Definition

Two sets A and B are said to be equicardinal, if there is a one-to-one mapping between the elements of these sets.

Equicardinal Sets

Natural numbers and even numbers are equicardinal.

Diagonalization and Cantor

Definition

Two sets A and B are said to be equicardinal, if there is a one-to-one mapping between the elements of these sets.

Equicardinal Sets

Natural numbers and even numbers are equicardinal.

Natural numbers and odd numbers are equicardinal.

Diagonalization and Cantor

Definition

Two sets A and B are said to be equicardinal, if there is a one-to-one mapping between the elements of these sets.

Equicardinal Sets

Natural numbers and even numbers are equicardinal.

Natural numbers and odd numbers are equicardinal.

Definition

If C is a set, let 2^C denote its power set,

Diagonalization and Cantor

Definition

Two sets A and B are said to be equicardinal, if there is a one-to-one mapping between the elements of these sets.

Equicardinal Sets

Natural numbers and even numbers are equicardinal.

Natural numbers and odd numbers are equicardinal.

Definition

If C is a set, let 2^C denote its power set, that is,

$$2^C = \{D : D \subseteq C\}.$$

Diagonalization and Cantor

Diagonalization and Cantor

Theorem

The set of natural numbers, N and its power set 2^N are not equicardinal.

Diagonalization and Cantor

Theorem

The set of natural numbers, N and its power set 2^N are not equicardinal.

Proof

Diagonalization and Cantor

Theorem

The set of natural numbers, N and its power set 2^N are not equicardinal.

Proof

Assume that we have some enumeration f of 2^N.

Diagonalization and Cantor

Theorem

The set of natural numbers, N and its power set 2^N are not equicardinal.

Proof

Assume that we have some enumeration f of 2^N.

Consider $C = \{x \in N : x \notin f(x)\}$.

Diagonalization and Cantor

Theorem

The set of natural numbers, N and its power set 2^N are not equicardinal.

Proof

Assume that we have some enumeration f of 2^N.

Consider $C = \{x \in N : x \notin f(x)\}$.

Since C is a subset of N, we have that there is a $c \in N$, such that $f(c) = C$.

Diagonalization and Cantor

Theorem

The set of natural numbers, N and its power set 2^N are not equicardinal.

Proof

Assume that we have some enumeration f of 2^N.

Consider $C = \{x \in N : x \notin f(x)\}$.

Since C is a subset of N, we have that there is a $c \in N$, such that $f(c) = C$.

If $c \in C$,

Diagonalization and Cantor

Theorem

The set of natural numbers, N and its power set 2^N are not equicardinal.

Proof

Assume that we have some enumeration f of 2^N.

Consider $C = \{x \in N : x \notin f(x)\}$.

Since C is a subset of N, we have that there is a $c \in N$, such that $f(c) = C$.

If $c \in C$, then $c \in f(c)$,

Diagonalization and Cantor

Theorem

The set of natural numbers, N and its power set 2^N are not equicardinal.

Proof

Assume that we have some enumeration f of 2^N.

Consider $C = \{x \in N : x \notin f(x)\}$.

Since C is a subset of N, we have that there is a $c \in N$, such that $f(c) = C$.

If $c \in C$, then $c \in f(c)$, hence $c \notin C$.

Diagonalization and Cantor

Theorem

The set of natural numbers, N and its power set 2^N are not equicardinal.

Proof

Assume that we have some enumeration f of 2^N.

Consider $C = \{x \in N : x \notin f(x)\}$.

Since C is a subset of N, we have that there is a $c \in N$, such that $f(c) = C$.

If $c \in C$, then $c \in f(c)$, hence $c \notin C$.

If $c \notin C$,

Diagonalization and Cantor

Theorem

The set of natural numbers, N and its power set 2^N are not equicardinal.

Proof

Assume that we have some enumeration f of 2^N.

Consider $C = \{x \in N : x \notin f(x)\}$.

Since C is a subset of N, we have that there is a $c \in N$, such that $f(c) = C$.

If $c \in C$, then $c \in f(c)$, hence $c \notin C$.

If $c \notin C$, then $c \notin f(c)$,

Diagonalization and Cantor

Theorem

The set of natural numbers, N and its power set 2^N are not equicardinal.

Proof

Assume that we have some enumeration f of 2^N.

Consider $C = \{x \in N : x \notin f(x)\}$.

Since C is a subset of N, we have that there is a $c \in N$, such that $f(c) = C$.

If $c \in C$, then $c \in f(c)$, hence $c \notin C$.

If $c \notin C$, then $c \notin f(c)$, hence $c \in C$.

Diagonalization and Cantor

Theorem

The set of natural numbers, N and its power set 2^N are not equicardinal.

Proof

Assume that we have some enumeration f of 2^N.

Consider $C = \{x \in N : x \notin f(x)\}$.

Since C is a subset of N, we have that there is a $c \in N$, such that $f(c) = C$.

If $c \in C$, then $c \in f(c)$, hence $c \notin C$.

If $c \notin C$, then $c \notin f(c)$, hence $c \in C$.

We have that $c \notin C$ if and only if $c \in C$,

Diagonalization and Cantor

Theorem

The set of natural numbers, N and its power set 2^N are not equicardinal.

Proof

Assume that we have some enumeration f of 2^N.

Consider $C = \{x \in N : x \notin f(x)\}$.

Since C is a subset of N, we have that there is a $c \in N$, such that $f(c) = C$.

If $c \in C$, then $c \in f(c)$, hence $c \notin C$.

If $c \notin C$, then $c \notin f(c)$, hence $c \in C$.

We have that $c \notin C$ if and only if $c \in C$, which is a contradiction.

The Halting Problem

The Halting Problem

Idea

The Halting Problem

Idea

Since some programs halt and others do not,

The Halting Problem

Idea

Since some programs halt and others do not, it would be nice to be able to tell which is which.

The Halting Problem

Idea

Since some programs halt and others do not, it would be nice to be able to tell which is which. Consider the following problem:

The Halting Problem

Idea

Since some programs halt and others do not, it would be nice to be able to tell which is which. Consider the following problem:

Problem

Given a program Π *and an input* x*,*

The Halting Problem

Idea

Since some programs halt and others do not, it would be nice to be able to tell which is which. Consider the following problem:

Problem

Given a program Π *and an input* x*, determine whether* Π *will halt when* x *is given as the input.*

The Halting Problem

Idea

Since some programs halt and others do not, it would be nice to be able to tell which is which. Consider the following problem:

Problem

Given a program Π and an input x, determine whether Π will halt when x is given as the input.

Theorem

The Halting Problem is undecidable.

The Halting Problem

The Halting Problem

Proof

The Halting Problem

Proof

Assume that there is a program A that solves the Halting Problem.

The Halting Problem

Proof

Assume that there is a program A that solves the Halting Problem.

A returns TRUE if Π halts on x,

The Halting Problem

Proof

Assume that there is a program A that solves the Halting Problem.

A returns TRUE if Π halts on x, and FALSE, otherwise.

The Halting Problem

Proof

Assume that there is a program A that solves the Halting Problem.

A returns TRUE if Π halts on x, and FALSE, otherwise.

Consider a program B that is defined as follows:

The Halting Problem

Proof

Assume that there is a program A that solves the Halting Problem.

A returns TRUE if Π halts on x, and FALSE, otherwise.

Consider a program B that is defined as follows: if $A(\Pi, \Pi) = TRUE$, then B goes to an infinite loop,

The Halting Problem

Proof

Assume that there is a program A that solves the Halting Problem.

A returns TRUE if Π halts on x, and FALSE, otherwise.

Consider a program B that is defined as follows: if $A(\Pi, \Pi) = TRUE$, then B goes to an infinite loop, otherwise B returns TRUE.

The Halting Problem

Proof

Assume that there is a program A that solves the Halting Problem.

A returns TRUE if Π halts on x, and FALSE, otherwise.

Consider a program B that is defined as follows: if $A(\Pi, \Pi) = TRUE$, then B goes to an infinite loop, otherwise B returns TRUE.

If $B(B)$ halts,

The Halting Problem

Proof

Assume that there is a program A that solves the Halting Problem.

A returns TRUE if Π halts on x, and FALSE, otherwise.

Consider a program B that is defined as follows: if $A(\Pi, \Pi) = \mathit{TRUE}$, then B goes to an infinite loop, otherwise B returns TRUE.

If $B(B)$ halts, then $A(B, B) = \mathit{FALSE}$,

The Halting Problem

Proof

Assume that there is a program A that solves the Halting Problem.

A returns TRUE if Π halts on x, and FALSE, otherwise.

Consider a program B that is defined as follows: if $A(\Pi, \Pi) = \mathit{TRUE}$, then B goes to an infinite loop, otherwise B returns TRUE.

If $B(B)$ halts, then $A(B, B) = \mathit{FALSE}$, hence $B(B)$ does not halt.

The Halting Problem

Proof

Assume that there is a program A that solves the Halting Problem.

A returns TRUE if Π halts on x, and FALSE, otherwise.

Consider a program B that is defined as follows: if $A(\Pi, \Pi) = TRUE$, then B goes to an infinite loop, otherwise B returns TRUE.

If $B(B)$ halts, then $A(B, B) = FALSE$, hence $B(B)$ does not halt.

If $B(B)$ does not halt,

The Halting Problem

Proof

Assume that there is a program A that solves the Halting Problem.

A returns TRUE if Π halts on x, and FALSE, otherwise.

Consider a program B that is defined as follows: if $A(\Pi, \Pi) = TRUE$, then B goes to an infinite loop, otherwise B returns TRUE.

If $B(B)$ halts, then $A(B, B) = FALSE$, hence $B(B)$ does not halt.

If $B(B)$ does not halt, then $A(B, B) = TRUE$,

The Halting Problem

Proof

Assume that there is a program A that solves the Halting Problem.

A returns TRUE if Π halts on x, and FALSE, otherwise.

Consider a program B that is defined as follows: if $A(\Pi, \Pi) = \mathit{TRUE}$, then B goes to an infinite loop, otherwise B returns TRUE.

If $B(B)$ halts, then $A(B, B) = \mathit{FALSE}$, hence $B(B)$ does not halt.

If $B(B)$ does not halt, then $A(B, B) = \mathit{TRUE}$, hence $B(B)$ halts.

The Halting Problem

Proof

Assume that there is a program A that solves the Halting Problem.

A returns TRUE if Π halts on x, and FALSE, otherwise.

Consider a program B that is defined as follows: if $A(\Pi, \Pi) = TRUE$, then B goes to an infinite loop, otherwise B returns TRUE.

If $B(B)$ halts, then $A(B, B) = FALSE$, hence $B(B)$ does not halt.

If $B(B)$ does not halt, then $A(B, B) = TRUE$, hence $B(B)$ halts.

In both cases we have a contradiction,

The Halting Problem

Proof

Assume that there is a program A that solves the Halting Problem.

A returns TRUE if Π halts on x, and FALSE, otherwise.

Consider a program B that is defined as follows: if $A(\Pi, \Pi) = \mathit{TRUE}$, then B goes to an infinite loop, otherwise B returns TRUE.

If $B(B)$ halts, then $A(B, B) = \mathit{FALSE}$, hence $B(B)$ does not halt.

If $B(B)$ does not halt, then $A(B, B) = \mathit{TRUE}$, hence $B(B)$ halts.

In both cases we have a contradiction, hence A cannot exist.

The 42 Problem

The 42 Problem

Idea

The 42 Problem

Idea

We have one undecidable problem.

The 42 Problem

Idea

We have one undecidable problem. We can prove that other problems are undecidable by reducing The Halting Problem to them.

The 42 Problem

Idea

We have one undecidable problem. We can prove that other problems are undecidable by reducing The Halting Problem to them. Consider the following problem:

The 42 Problem

Idea

We have one undecidable problem. We can prove that other problems are undecidable by reducing The Halting Problem to them. Consider the following problem:

Problem

Given a program Π.

The 42 Problem

Idea

We have one undecidable problem. We can prove that other problems are undecidable by reducing The Halting Problem to them. Consider the following problem:

Problem

Given a program Π. *Is there an input* x,

The 42 Problem

Idea

We have one undecidable problem. We can prove that other problems are undecidable by reducing The Halting Problem to them. Consider the following problem:

Problem

Given a program Π*. Is there an input* x*, such that* $\Pi(x)$ *halts and returns* 42*?*

The 42 Problem

Idea

We have one undecidable problem. We can prove that other problems are undecidable by reducing The Halting Problem to them. Consider the following problem:

Problem

Given a program Π. *Is there an input* x*, such that* $\Pi(x)$ *halts and returns* 42*?*

Theorem

The 42 *Problem is undecidable.*

The 42 Problem

The 42 Problem

Proof

The 42 Problem

Proof

Given a program Π and an input x,

The 42 Problem

Proof

Given a program Π and an input x, we can convert them to a program Π' which ignores its input, runs $\Pi(x)$ instead,

The 42 Problem

Proof

Given a program Π and an input x, we can convert them to a program Π' which ignores its input, runs $\Pi(x)$ instead, and returns 42 if it halts.

The 42 Problem

Proof

Given a program Π and an input x, we can convert them to a program Π' which ignores its input, runs $\Pi(x)$ instead, and returns 42 if it halts.

If $\Pi(x)$ halts, then $\Pi'(x')$ returns 42.

The 42 Problem

Proof

Given a program Π and an input x, we can convert them to a program Π' which ignores its input, runs $\Pi(x)$ instead, and returns 42 if it halts.

If $\Pi(x)$ halts, then $\Pi'(x')$ returns 42.

If $\Pi(x)$ does not halt, then neither does Π',

The 42 Problem

Proof

Given a program Π and an input x, we can convert them to a program Π' which ignores its input, runs $\Pi(x)$ instead, and returns 42 if it halts.

If $\Pi(x)$ halts, then $\Pi'(x')$ returns 42.

If $\Pi(x)$ does not halt, then neither does Π', no matter what input x' we give it.

The 42 Problem

Proof

Given a program Π and an input x, we can convert them to a program Π' which ignores its input, runs $\Pi(x)$ instead, and returns 42 if it halts.

If $\Pi(x)$ halts, then $\Pi'(x')$ returns 42.

If $\Pi(x)$ does not halt, then neither does Π', no matter what input x' we give it.

Thus, if the 42 Problem were decidable,

The 42 Problem

Proof

Given a program Π and an input x, we can convert them to a program Π' which ignores its input, runs $\Pi(x)$ instead, and returns 42 if it halts.

If $\Pi(x)$ halts, then $\Pi'(x')$ returns 42.

If $\Pi(x)$ does not halt, then neither does Π', no matter what input x' we give it.

Thus, if the 42 Problem were decidable, the Halting Problem would be too.

The 42 Problem

Proof

Given a program Π and an input x, we can convert them to a program Π' which ignores its input, runs $\Pi(x)$ instead, and returns 42 if it halts.

If $\Pi(x)$ halts, then $\Pi'(x')$ returns 42.

If $\Pi(x)$ does not halt, then neither does Π', no matter what input x' we give it.

Thus, if the 42 Problem were decidable, the Halting Problem would be too.

But we know that the Halting Problem is undecidable,

The 42 Problem

Proof

Given a program Π and an input x, we can convert them to a program Π' which ignores its input, runs $\Pi(x)$ instead, and returns 42 if it halts.

If $\Pi(x)$ halts, then $\Pi'(x')$ returns 42.

If $\Pi(x)$ does not halt, then neither does Π', no matter what input x' we give it.

Thus, if the 42 Problem were decidable, the Halting Problem would be too.

But we know that the Halting Problem is undecidable, hence the 42 Problem must be undecidable as well.

Reductions

Reductions

The idea

Reductions

The idea

The mapping that we have just constructed, maps the instances of the Halting Problem to those of the 42 Problem, so that the answers are the same.

Reductions

The idea

The mapping that we have just constructed, maps the instances of the Halting Problem to those of the 42 Problem, so that the answers are the same.

It shows that the 42 Problem is at least hard as the Halting Problem, that is

$$\text{The Halting Problem} \leq \text{The 42 problem.}$$

Reductions

The idea

The mapping that we have just constructed, maps the instances of the Halting Problem to those of the 42 Problem, so that the answers are the same.

It shows that the 42 Problem is at least hard as the Halting Problem, that is

$$\text{The Halting Problem} \leq \text{The 42 problem.}$$

The reductions that we used in the proof are *computable* reductions.

Reductions

The idea

The mapping that we have just constructed, maps the instances of the Halting Problem to those of the 42 Problem, so that the answers are the same.

It shows that the 42 Problem is at least hard as the Halting Problem, that is

$$\text{The Halting Problem} \;\leq\; \text{The 42 problem.}$$

The reductions that we used in the proof are *computable* reductions.

That is, a reduction can be any function from instances of A to instances of B that we can compute in finite time.

Reductions

The idea

The mapping that we have just constructed, maps the instances of the Halting Problem to those of the 42 Problem, so that the answers are the same.

It shows that the 42 Problem is at least hard as the Halting Problem, that is

$$\text{The Halting Problem} \ \leq \ \text{The 42 problem.}$$

The reductions that we used in the proof are *computable* reductions.

That is, a reduction can be any function from instances of A to instances of B that we can compute in finite time.

In this case, $A \leq B$ implies that if B is decidable then A is decidable, and conversely, if A is not decidable, then B is undecidable, too.

Recursive Enumerability

Recursive Enumerability

The idea

Recursive Enumerability

The idea

While the Halting Problem is undecidable, it has kind of one-sided decidability.

Recursive Enumerability

The idea

While the Halting Problem is undecidable, it has kind of one-sided decidability.

If the instance (Π, x) is a YES instance, then we can learn this fact in a finite amount of time, by simulating Π until it halts.

Recursive Enumerability

The idea

While the Halting Problem is undecidable, it has kind of one-sided decidability.

If the instance (Π, x) is a YES instance, then we can learn this fact in a finite amount of time, by simulating Π until it halts.

In other words, the Halting Problem can be represented as:

Recursive Enumerability

The idea

While the Halting Problem is undecidable, it has kind of one-sided decidability.

If the instance (Π, x) is a YES instance, then we can learn this fact in a finite amount of time, by simulating Π until it halts.

In other words, the Halting Problem can be represented as:

$$Halts(\Pi, x) = \exists t : HaltsInTime(\Pi, x, t),$$

Recursive Enumerability

The idea

While the Halting Problem is undecidable, it has kind of one-sided decidability.

If the instance (Π, x) is a YES instance, then we can learn this fact in a finite amount of time, by simulating Π until it halts.

In other words, the Halting Problem can be represented as:

$$Halts(\Pi, x) = \exists t : HaltsInTime(\Pi, x, t),$$

where $HaltsInTime(\Pi, x, t)$ is the property that Π, given x as input, halts on its tth step.

Recursive Enumerability

The idea

While the Halting Problem is undecidable, it has kind of one-sided decidability.

If the instance (Π, x) is a YES instance, then we can learn this fact in a finite amount of time, by simulating Π until it halts.

In other words, the Halting Problem can be represented as:

$$\mathit{Halts}(\Pi, x) = \exists t : \mathit{HaltsInTime}(\Pi, x, t),$$

where $\mathit{HaltsInTime}(\Pi, x, t)$ is the property that Π, given x as input, halts on its tth step.

$\mathit{HaltsInTime}(\Pi, x, t)$ is decidable?

Recursive Enumerability

The idea

While the Halting Problem is undecidable, it has kind of one-sided decidability.

If the instance (Π, x) is a YES instance, then we can learn this fact in a finite amount of time, by simulating Π until it halts.

In other words, the Halting Problem can be represented as:

$$\mathit{Halts}(\Pi, x) = \exists t : \mathit{HaltsInTime}(\Pi, x, t),$$

where $\mathit{HaltsInTime}(\Pi, x, t)$ is the property that Π, given x as input, halts on its tth step.

$\mathit{HaltsInTime}(\Pi, x, t)$ is decidable? Simulate Π for t steps.

Recursive Enumerability

The idea

While the Halting Problem is undecidable, it has kind of one-sided decidability.

If the instance (Π, x) is a YES instance, then we can learn this fact in a finite amount of time, by simulating Π until it halts.

In other words, the Halting Problem can be represented as:

$$Halts(\Pi, x) = \exists t : HaltsInTime(\Pi, x, t),$$

where $HaltsInTime(\Pi, x, t)$ is the property that Π, given x as input, halts on its tth step.

$HaltsInTime(\Pi, x, t)$ is decidable? Simulate Π for t steps.

Thus $Halts(\Pi, x)$ is a combination of a decidable problem with a single $\exists$.

Recursive Enumerability

Recursive Enumerability

Definition

Let **RE** denote the class of problems, that can be represented as a combination of a decidable problem with a single $\exists$.

Recursive Enumerability

Definition

Let **RE** denote the class of problems, that can be represented as a combination of a decidable problem with a single $\exists$.

Analogy with **P** and **NP**

Recursive Enumerability

Definition

Let **RE** denote the class of problems, that can be represented as a combination of a decidable problem with a single $\exists$.

Analogy with **P** and **NP**

In some ways, this is analogous to the relationship between **P** and **NP**.

Recursive Enumerability

Definition

Let **RE** denote the class of problems, that can be represented as a combination of a decidable problem with a single $\exists$.

Analogy with **P** and **NP**

In some ways, this is analogous to the relationship between **P** and **NP**.

Decidable problems are the analogues of **P**.

Recursive Enumerability

Definition

Let **RE** denote the class of problems, that can be represented as a combination of a decidable problem with a single $\exists$.

Analogy with **P** and **NP**

In some ways, this is analogous to the relationship between **P** and **NP**.

Decidable problems are the analogues of **P**.

Recall that a property A is in **NP**,

Recursive Enumerability

Definition

Let **RE** denote the class of problems, that can be represented as a combination of a decidable problem with a single $\exists$.

Analogy with **P** and **NP**

In some ways, this is analogous to the relationship between **P** and **NP**.

Decidable problems are the analogues of **P**.

Recall that a property A is in **NP**, if it can be written as

$$A(x) = \exists w : B(x, w),$$

Recursive Enumerability

Definition

Let **RE** denote the class of problems, that can be represented as a combination of a decidable problem with a single $\exists$.

Analogy with **P** and **NP**

In some ways, this is analogous to the relationship between **P** and **NP**.

Decidable problems are the analogues of **P**.

Recall that a property A is in **NP**, if it can be written as

$$A(x) = \exists w : B(x, w),$$

where B is in **P**.

Recursive Enumerability

Definition

Let **RE** denote the class of problems, that can be represented as a combination of a decidable problem with a single $\exists$.

Analogy with **P** and **NP**

In some ways, this is analogous to the relationship between **P** and **NP**.

Decidable problems are the analogues of **P**.

Recall that a property A is in **NP**, if it can be written as

$$A(x) = \exists w : B(x, w),$$

where B is in **P**.

In other words, x is a YES instance of A, if some witness w exists,

Recursive Enumerability

Definition

Let **RE** denote the class of problems, that can be represented as a combination of a decidable problem with a single $\exists$.

Analogy with **P** and **NP**

In some ways, this is analogous to the relationship between **P** and **NP**.

Decidable problems are the analogues of **P**.

Recall that a property A is in **NP**, if it can be written as

$$A(x) = \exists w : B(x, w),$$

where B is in **P**.

In other words, x is a YES instance of A, if some witness w exists, and the property $B(x, w)$, that w is a valid witness for x, can be checked in polynomial time.

Recursive Enumerability

Definition

Let **RE** denote the class of problems, that can be represented as a combination of a decidable problem with a single $\exists$.

Analogy with **P** and **NP**

In some ways, this is analogous to the relationship between **P** and **NP**.

Decidable problems are the analogues of **P**.

Recall that a property A is in **NP**, if it can be written as

$$A(x) = \exists w : B(x, w),$$

where B is in **P**.

In other words, x is a YES instance of A, if some witness w exists, and the property $B(x, w)$, that w is a valid witness for x, can be checked in polynomial time.

Similarly, t is a witness that Π halts,

Recursive Enumerability

Definition

Let **RE** denote the class of problems, that can be represented as a combination of a decidable problem with a single $\exists$.

Analogy with **P** and **NP**

In some ways, this is analogous to the relationship between **P** and **NP**.

Decidable problems are the analogues of **P**.

Recall that a property A is in **NP**, if it can be written as

$$A(x) = \exists w : B(x, w),$$

where B is in **P**.

In other words, x is a YES instance of A, if some witness w exists, and the property $B(x, w)$, that w is a valid witness for x, can be checked in polynomial time.

Similarly, t is a witness that Π halts, and we can check the validity of this witness in finite time.

Recursive Enumerability

Recursive Enumerability

More analogy with **P** and **NP**

Recursive Enumerability

More analogy with P and NP

Recall that **coNP** stands for the class of problems for which NO instances have witnesses,

Recursive Enumerability

More analogy with **P** and **NP**

Recall that **coNP** stands for the class of problems for which NO instances have witnesses, whose validity can be verified in polynomial time.

Recursive Enumerability

More analogy with P and NP

Recall that **coNP** stands for the class of problems for which NO instances have witnesses, whose validity can be verified in polynomial time.

Similarly, the class **coRE** stands for the class of problems, whose NO instances are in **RE**.

Recursive Enumerability and the **P** vs. **NP** Problem

Recursive Enumerability and the **P** vs. **NP** Problem

Some relations among the classes

Recursive Enumerability and the **P** vs. **NP** Problem

Some relations among the classes

Unlike the polynomial world, where the **P** vs. **NP** question remains unsolved,

Recursive Enumerability and the **P** vs. **NP** Problem

Some relations among the classes

Unlike the polynomial world, where the **P** vs. **NP** question remains unsolved, we know that **RE**, **coRE** and **Decidable** are different.

Recursive Enumerability and the **P** vs. **NP** Problem

Some relations among the classes

Unlike the polynomial world, where the **P** vs. **NP** question remains unsolved, we know that **RE**, **coRE** and **Decidable** are different.

Show that

$$\textbf{Decidable} = \textbf{RE} \cap \textbf{coRE}.$$

Recursive Enumerability and the **P** vs. **NP** Problem

Some relations among the classes

Unlike the polynomial world, where the **P** vs. **NP** question remains unsolved, we know that **RE**, **coRE** and **Decidable** are different.

Show that

$$\mathbf{Decidable} = \mathbf{RE} \cap \mathbf{coRE}.$$

In other words, if both S and $\bar{S}$ are in **RE**,

Recursive Enumerability and the **P** vs. **NP** Problem

Some relations among the classes

Unlike the polynomial world, where the **P** vs. **NP** question remains unsolved, we know that **RE**, **coRE** and **Decidable** are different.

Show that

$$\textbf{Decidable} = \textbf{RE} \cap \textbf{coRE}.$$

In other words, if both S and $\bar{S}$ are in **RE**, then S is decidable.

Recursive Enumerability and the **P** vs. **NP** Problem

Some relations among the classes

Unlike the polynomial world, where the **P** vs. **NP** question remains unsolved, we know that **RE**, **coRE** and **Decidable** are different.

Show that

$$\mathbf{Decidable} = \mathbf{RE} \cap \mathbf{coRE}.$$

In other words, if both S and $\bar{S}$ are in **RE**, then S is decidable.

From this one can conclude that $\mathbf{RE} \neq \mathbf{coRE}$.

Recursive Enumerability and the **P** vs. **NP** Problem

Some relations among the classes

Unlike the polynomial world, where the **P** vs. **NP** question remains unsolved, we know that **RE**, **coRE** and **Decidable** are different.

Show that

$$\textbf{Decidable} = \textbf{RE} \cap \textbf{coRE}.$$

In other words, if both S and $\bar{S}$ are in **RE**, then S is decidable.

From this one can conclude that **RE**$\neq$**coRE**.

In contrast, the questions whether **NP**$\neq$**coNP** and **P**$=$**NP**$\cap$**coNP** are still open.

Polynomial Hierarchy

Definition

Let **D** be a class of problems.

Polynomial Hierarchy

Definition

Let **D** be a class of problems. A problem L is in $\mathbf{P}^{\mathbf{D}}$,

Polynomial Hierarchy

Definition

Let $\mathbf{D}$ be a class of problems. A problem L is in $\mathbf{P}^{\mathbf{D}}$, if there exists a problem $L' \in \mathbf{D}$, such that L can be solved in polynomial time by an oracle program using an L' oracle.

Polynomial Hierarchy

Polynomial Hierarchy

Definition

Polynomial Hierarchy

Definition

The *polynomial hierarchy* is the following sequence of classes:

Polynomial Hierarchy

Definition

The *polynomial hierarchy* is the following sequence of classes:

1. $\Delta_0 P = \Sigma_0 P = \Pi_0 P = \mathbf{P}$

Polynomial Hierarchy

Definition

The *polynomial hierarchy* is the following sequence of classes:

1. $\Delta_0 P = \Sigma_0 P = \Pi_0 P = \mathbf{P}$
2. $\Delta_{i+1} P = \mathbf{P}^{\Sigma_i P}$

Polynomial Hierarchy

Definition

The *polynomial hierarchy* is the following sequence of classes:

1. $\Delta_0\mathrm{P} = \Sigma_0\mathrm{P} = \Pi_0\mathrm{P} = \mathbf{P}$
2. $\Delta_{i+1}\mathrm{P} = \mathbf{P}^{\Sigma_i P}$
3. $\Sigma_{i+1}\mathrm{P} = \mathbf{NP}^{\Sigma_i P}$

Polynomial Hierarchy

Definition

The *polynomial hierarchy* is the following sequence of classes:

1. $\Delta_0 \mathrm{P} = \Sigma_0 \mathrm{P} = \Pi_0 \mathrm{P} = \mathbf{P}$
2. $\Delta_{i+1} \mathrm{P} = \mathbf{P}^{\Sigma_i P}$
3. $\Sigma_{i+1} \mathrm{P} = \mathbf{NP}^{\Sigma_i P}$
4. $\Pi_{i+1} \mathrm{P} = \mathbf{coNP}^{\Sigma_i P}$

Polynomial Hierarchy

Definition

The *polynomial hierarchy* is the following sequence of classes:

1. $\Delta_0\mathrm{P} = \Sigma_0\mathrm{P} = \Pi_0\mathrm{P} = \mathbf{P}$
2. $\Delta_{i+1}\mathrm{P} = \mathbf{P}^{\Sigma_i P}$
3. $\Sigma_{i+1}\mathrm{P} = \mathbf{NP}^{\Sigma_i P}$
4. $\Pi_{i+1}\mathrm{P} = \mathbf{coNP}^{\Sigma_i P}$

For all $i \geq 0$.

Polynomial Hierarchy

Definition

The *polynomial hierarchy* is the following sequence of classes:

1. $\Delta_0\mathrm{P} = \Sigma_0\mathrm{P} = \Pi_0\mathrm{P} = \mathbf{P}$
2. $\Delta_{i+1}\mathrm{P} = \mathbf{P}^{\Sigma_i P}$
3. $\Sigma_{i+1}\mathrm{P} = \mathbf{NP}^{\Sigma_i P}$
4. $\Pi_{i+1}\mathrm{P} = \mathbf{coNP}^{\Sigma_i P}$

For all $i \geq 0$. We also define the collective class $\mathbf{PH} = \bigcup_{i \geq 0} \Sigma_i P$.

Polynomial Hierarchy

Definition

The *polynomial hierarchy* is the following sequence of classes:

1. $\Delta_0 P = \Sigma_0 P = \Pi_0 P = \mathbf{P}$
2. $\Delta_{i+1} P = \mathbf{P}^{\Sigma_i P}$
3. $\Sigma_{i+1} P = \mathbf{NP}^{\Sigma_i P}$
4. $\Pi_{i+1} P = \mathbf{coNP}^{\Sigma_i P}$

For all $i \geq 0$. We also define the collective class $\mathbf{PH} = \bigcup_{i \geq 0} \Sigma_i P$.

Observations

Note that because $\Sigma_0 P = \mathbf{P}$, we have that $\Sigma_1 P = \mathbf{NP}$, $\Delta_1 P = \mathbf{P}$, and $\Pi_1 P = \mathbf{coNP}$.

Polynomial Hierarchy

Definition

The *polynomial hierarchy* is the following sequence of classes:

1. $\Delta_0 \mathrm{P} = \Sigma_0 \mathrm{P} = \Pi_0 \mathrm{P} = \mathbf{P}$
2. $\Delta_{i+1} \mathrm{P} = \mathbf{P}^{\Sigma_i P}$
3. $\Sigma_{i+1} \mathrm{P} = \mathbf{NP}^{\Sigma_i P}$
4. $\Pi_{i+1} \mathrm{P} = \mathbf{coNP}^{\Sigma_i P}$

For all $i \geq 0$. We also define the collective class $\mathbf{PH} = \bigcup_{i \geq 0} \Sigma_i P$.

Observations

Note that because $\Sigma_0 \mathrm{P} = \mathbf{P}$, we have that $\Sigma_1 \mathrm{P} = \mathbf{NP}$, $\Delta_1 \mathrm{P} = \mathbf{P}$, and $\Pi_1 \mathrm{P} = \mathbf{coNP}$.

At each level the classes are believed to be distinct and are known to hold the same relationship as **P**, **NP** and **coNP**.

Polynomial Hierarchy

Definition

The *polynomial hierarchy* is the following sequence of classes:

1. $\Delta_0 \mathrm{P} = \Sigma_0 \mathrm{P} = \Pi_0 \mathrm{P} = \mathbf{P}$
2. $\Delta_{i+1} \mathrm{P} = \mathbf{P}^{\Sigma_i P}$
3. $\Sigma_{i+1} \mathrm{P} = \mathbf{NP}^{\Sigma_i P}$
4. $\Pi_{i+1} \mathrm{P} = \mathbf{coNP}^{\Sigma_i P}$

For all $i \geq 0$. We also define the collective class $\mathbf{PH} = \bigcup_{i \geq 0} \Sigma_i P$.

Observations

Note that because $\Sigma_0 \mathrm{P} = \mathbf{P}$, we have that $\Sigma_1 \mathrm{P} = \mathbf{NP}$, $\Delta_1 \mathrm{P} = \mathbf{P}$, and $\Pi_1 \mathrm{P} = \mathbf{coNP}$.

At each level the classes are believed to be distinct and are known to hold the same relationship as **P**, **NP** and **coNP**.

Also, each class at each level includes all classes at the previous levels.

Arithmetical Hierarchy

Arithmetical Hierarchy

Definition

Arithmetical Hierarchy

Definition

The *arithmetical hierarchy* is the following sequence of classes:

Arithmetical Hierarchy

Definition

The *arithmetical hierarchy* is the following sequence of classes:

1. $\Delta_0 D = \Sigma_0 P = \Pi_0 P =$ **Decidable**

Arithmetical Hierarchy

Definition

The *arithmetical hierarchy* is the following sequence of classes:

1. $\Delta_0\mathrm{D} = \Sigma_0\mathrm{P} = \Pi_0\mathrm{P} =$ **Decidable**
2. $\Delta_{i+1}\mathrm{D} = \textbf{Decidable}^{\Sigma_i D}$

Arithmetical Hierarchy

Definition

The *arithmetical hierarchy* is the following sequence of classes:

1. $\Delta_0\text{D} = \Sigma_0\text{P} = \Pi_0\text{P} = \textbf{Decidable}$
2. $\Delta_{i+1}\text{D} = \textbf{Decidable}^{\Sigma_i D}$
3. $\Sigma_{i+1}\text{D} = \textbf{RE}^{\Sigma_i D}$

Arithmetical Hierarchy

Definition

The *arithmetical hierarchy* is the following sequence of classes:

1. $\Delta_0\mathrm{D} = \Sigma_0\mathrm{P} = \Pi_0\mathrm{P} = \textbf{Decidable}$
2. $\Delta_{i+1}\mathrm{D} = \textbf{Decidable}^{\Sigma_i D}$
3. $\Sigma_{i+1}\mathrm{D} = \textbf{RE}^{\Sigma_i D}$
4. $\Pi_{i+1}\mathrm{D} = \textbf{coRE}^{\Sigma_i D}$

Arithmetical Hierarchy

Definition

The *arithmetical hierarchy* is the following sequence of classes:

1. $\Delta_0 \mathrm{D} = \Sigma_0 \mathrm{P} = \Pi_0 \mathrm{P} =$ **Decidable**
2. $\Delta_{i+1} \mathrm{D} = \mathbf{Decidable}^{\Sigma_i D}$
3. $\Sigma_{i+1} \mathrm{D} = \mathbf{RE}^{\Sigma_i D}$
4. $\Pi_{i+1} \mathrm{D} = \mathbf{coRE}^{\Sigma_i D}$

For all $i \geq 0$.

Arithmetical Hierarchy

Definition

The *arithmetical hierarchy* is the following sequence of classes:

1. $\Delta_0\mathrm{D} = \Sigma_0\mathrm{P} = \Pi_0\mathrm{P} =$ **Decidable**
2. $\Delta_{i+1}\mathrm{D} = \textbf{Decidable}^{\Sigma_i D}$
3. $\Sigma_{i+1}\mathrm{D} = \textbf{RE}^{\Sigma_i D}$
4. $\Pi_{i+1}\mathrm{D} = \textbf{coRE}^{\Sigma_i D}$

For all $i \geq 0$. We also define the collective class $\textbf{AH} = \bigcup_{i \geq 0} \Sigma_i D$.

Arithmetical Hierarchy

Definition

The *arithmetical hierarchy* is the following sequence of classes:

1. $\Delta_0\mathrm{D} = \Sigma_0\mathrm{P} = \Pi_0\mathrm{P} = \textbf{Decidable}$
2. $\Delta_{i+1}\mathrm{D} = \textbf{Decidable}^{\Sigma_i D}$
3. $\Sigma_{i+1}\mathrm{D} = \textbf{RE}^{\Sigma_i D}$
4. $\Pi_{i+1}\mathrm{D} = \textbf{coRE}^{\Sigma_i D}$

For all $i \geq 0$. We also define the collective class $\textbf{AH} = \bigcup_{i \geq 0} \Sigma_i D$.

What's Known

Arithmetical Hierarchy

Definition

The *arithmetical hierarchy* is the following sequence of classes:

1. Δ_0D = Σ_0P = Π_0P = **Decidable**
2. Δ_{i+1}D = $\textbf{Decidable}^{\Sigma_i D}$
3. Σ_{i+1}D = $\textbf{RE}^{\Sigma_i D}$
4. Π_{i+1}D = $\textbf{coRE}^{\Sigma_i D}$

For all $i \geq 0$. We also define the collective class $\textbf{AH} = \bigcup_{i\geq 0} \Sigma_i D$.

What's Known

Unlike the polynomial hierarchy, it is known that the levels of the arithmetical hierarchy are distinct.

Formal Systems

Formal Systems

The idea

Formal Systems

The idea

A *formal system* has a finite set of axioms including rules of inference such as modus ponens.

Formal Systems

The idea

A *formal system* has a finite set of axioms including rules of inference such as modus ponens. A and $A \longrightarrow B$ implies B.

Formal Systems

The idea

A *formal system* has a finite set of axioms including rules of inference such as modus ponens. A and $A \longrightarrow B$ implies B.

A theorem is a statement that can be proved, with some finite chain of reasoning, from the axioms.

Formal Systems

The idea

A *formal system* has a finite set of axioms including rules of inference such as modus ponens. A and $A \rightarrow B$ implies B.

A theorem is a statement that can be proved, with some finite chain of reasoning, from the axioms.

A formal system is consistent,

Formal Systems

The idea

A *formal system* has a finite set of axioms including rules of inference such as modus ponens. A and $A \rightarrow B$ implies B.

A theorem is a statement that can be proved, with some finite chain of reasoning, from the axioms.

A formal system is consistent, if there is no statement T such that both T and $\bar{T}$ are theorems.

Formal Systems

The idea

A *formal system* has a finite set of axioms including rules of inference such as modus ponens. A and $A \rightarrow B$ implies B.

A theorem is a statement that can be proved, with some finite chain of reasoning, from the axioms.

A formal system is consistent, if there is no statement T such that both T and $\bar{T}$ are theorems.

A formal system is complete,

Formal Systems

The idea

A *formal system* has a finite set of axioms including rules of inference such as modus ponens. A and $A \longrightarrow B$ implies B.

A theorem is a statement that can be proved, with some finite chain of reasoning, from the axioms.

A formal system is consistent, if there is no statement T such that both T and $\bar{T}$ are theorems.

A formal system is complete, if for each statement T, at least one of T and $\bar{T}$ is a theorem.

Formal Systems

The idea

A *formal system* has a finite set of axioms including rules of inference such as modus ponens. A and $A \rightarrow B$ implies B.

A theorem is a statement that can be proved, with some finite chain of reasoning, from the axioms.

A formal system is consistent, if there is no statement T such that both T and $\bar{T}$ are theorems.

A formal system is complete, if for each statement T, at least one of T and $\bar{T}$ is a theorem.

We can define a statement as true or false,

Formal Systems

The idea

A *formal system* has a finite set of axioms including rules of inference such as modus ponens. A and $A \to B$ implies B.

A theorem is a statement that can be proved, with some finite chain of reasoning, from the axioms.

A formal system is consistent, if there is no statement T such that both T and $\bar{T}$ are theorems.

A formal system is complete, if for each statement T, at least one of T and $\bar{T}$ is a theorem.

We can define a statement as true or false, by interpreting the symbols of the formal system in some standard way.

Formal Systems

The idea

A *formal system* has a finite set of axioms including rules of inference such as modus ponens. A and $A \rightarrow B$ implies B.

A theorem is a statement that can be proved, with some finite chain of reasoning, from the axioms.

A formal system is consistent, if there is no statement T such that both T and $\bar{T}$ are theorems.

A formal system is complete, if for each statement T, at least one of T and $\bar{T}$ is a theorem.

We can define a statement as true or false, by interpreting the symbols of the formal system in some standard way. $\exists$ - there exists,

Formal Systems

The idea

A *formal system* has a finite set of axioms including rules of inference such as modus ponens. A and $A \rightarrow B$ implies B.

A theorem is a statement that can be proved, with some finite chain of reasoning, from the axioms.

A formal system is consistent, if there is no statement T such that both T and $\bar{T}$ are theorems.

A formal system is complete, if for each statement T, at least one of T and $\bar{T}$ is a theorem.

We can define a statement as true or false, by interpreting the symbols of the formal system in some standard way. $\exists$ - there exists, $\wedge$- and,

Formal Systems

The idea

A *formal system* has a finite set of axioms including rules of inference such as modus ponens. A and $A \rightarrow B$ implies B.

A theorem is a statement that can be proved, with some finite chain of reasoning, from the axioms.

A formal system is consistent, if there is no statement T such that both T and $\bar{T}$ are theorems.

A formal system is complete, if for each statement T, at least one of T and $\bar{T}$ is a theorem.

We can define a statement as true or false, by interpreting the symbols of the formal system in some standard way. $\exists$ - there exists, $\wedge$- and, so on, and assuming that its variables refer to a specific set of mathematical objects such as integers.

David Hilbert and Kurt Gödel

David Hilbert and Kurt Gödel

David Hilbert

David Hilbert and Kurt Gödel

David Hilbert

The ideal system would be consistent and complete,

David Hilbert and Kurt Gödel

David Hilbert

The ideal system would be consistent and complete, in that all its theorems are true, and all true statements are theorems.

David Hilbert and Kurt Gödel

David Hilbert

The ideal system would be consistent and complete, in that all its theorems are true, and all true statements are theorems.

Such a system would fulfill Hilbert's dream of an axiomatic foundation for mathematics.

David Hilbert and Kurt Gödel

David Hilbert

The ideal system would be consistent and complete, in that all its theorems are true, and all true statements are theorems.

Such a system would fulfill Hilbert's dream of an axiomatic foundation for mathematics.

It would be powerful enough to prove all truths, and yet be free from paradoxes.

David Hilbert and Kurt Gödel

David Hilbert

The ideal system would be consistent and complete, in that all its theorems are true, and all true statements are theorems.

Such a system would fulfill Hilbert's dream of an axiomatic foundation for mathematics.

It would be powerful enough to prove all truths, and yet be free from paradoxes.

Kurt Gödel

David Hilbert and Kurt Gödel

David Hilbert

The ideal system would be consistent and complete, in that all its theorems are true, and all true statements are theorems.

Such a system would fulfill Hilbert's dream of an axiomatic foundation for mathematics.

It would be powerful enough to prove all truths, and yet be free from paradoxes.

Kurt Gödel

In 1931 Gödel dashed Hilbert's hopes.

David Hilbert and Kurt Gödel

David Hilbert

The ideal system would be consistent and complete, in that all its theorems are true, and all true statements are theorems.

Such a system would fulfill Hilbert's dream of an axiomatic foundation for mathematics.

It would be powerful enough to prove all truths, and yet be free from paradoxes.

Kurt Gödel

In 1931 Gödel dashed Hilbert's hopes.

He proved that no sufficiently powerful system is both consistent and complete.

David Hilbert and Kurt Gödel

David Hilbert

The ideal system would be consistent and complete, in that all its theorems are true, and all true statements are theorems.

Such a system would fulfill Hilbert's dream of an axiomatic foundation for mathematics.

It would be powerful enough to prove all truths, and yet be free from paradoxes.

Kurt Gödel

In 1931 Gödel dashed Hilbert's hopes.

He proved that no sufficiently powerful system is both consistent and complete.

He did this by constructing a self-referential statement, which can be interpreted as:

David Hilbert and Kurt Gödel

David Hilbert

The ideal system would be consistent and complete, in that all its theorems are true, and all true statements are theorems.

Such a system would fulfill Hilbert's dream of an axiomatic foundation for mathematics.

It would be powerful enough to prove all truths, and yet be free from paradoxes.

Kurt Gödel

In 1931 Gödel dashed Hilbert's hopes.

He proved that no sufficiently powerful system is both consistent and complete.

He did this by constructing a self-referential statement, which can be interpreted as:

This statement cannot be proved.

Proof of Gödel's Theorem

Proof of Gödel's Theorem

Idea

Proof of Gödel's Theorem

Idea

If this statement is false,

Proof of Gödel's Theorem

Idea

If this statement is false, then it can be proved,

Proof of Gödel's Theorem

Idea

If this statement is false, then it can be proved, hence it would violate the consistency.

Proof of Gödel's Theorem

Idea

If this statement is false, then it can be proved, hence it would violate the consistency.

Thus, it must be true,

Proof of Gödel's Theorem

Idea

If this statement is false, then it can be proved, hence it would violate the consistency.

Thus, it must be true, hence unprovable,

Proof of Gödel's Theorem

Idea

If this statement is false, then it can be proved, hence it would violate the consistency.

Thus, it must be true, hence unprovable, therefore there are truths that cannot be proved.

Proof of Gödel's Theorem

Idea

If this statement is false, then it can be proved, hence it would violate the consistency.

Thus, it must be true, hence unprovable, therefore there are truths that cannot be proved.

Remark

Proof of Gödel's Theorem

Idea

If this statement is false, then it can be proved, hence it would violate the consistency.

Thus, it must be true, hence unprovable, therefore there are truths that cannot be proved.

Remark

What we did demonstrates that the problem is in English.

Proof of Gödel's Theorem

Idea

If this statement is false, then it can be proved, hence it would violate the consistency.

Thus, it must be true, hence unprovable, therefore there are truths that cannot be proved.

Remark

What we did demonstrates that the problem is in English.

Gödel did something more, he showed that one can get similar statements in mathematics.

Proof of Gödel's Theorem

Idea

If this statement is false, then it can be proved, hence it would violate the consistency.

Thus, it must be true, hence unprovable, therefore there are truths that cannot be proved.

Remark

What we did demonstrates that the problem is in English.

Gödel did something more, he showed that one can get similar statements in mathematics.

Below we derive this theorem, as a consequence of the undecidability of the Halting Problem.

Proof of Gödel's Theorem

Proof of Gödel's Theorem

Proof

Proof of Gödel's Theorem

Proof

Let $\mathit{Theorem}(T)$ be the property that a statement T is provable.

Proof of Gödel's Theorem

Proof

Let *Theorem*(T) be the property that a statement T is provable.

Then it can be written as:

$$\mathit{Theorem}(T) = \exists P : \mathit{Proof}(P, T),$$

Proof of Gödel's Theorem

Proof

Let $\mathit{Theorem}(T)$ be the property that a statement T is provable.

Then it can be written as:

$$\mathit{Theorem}(T) = \exists P : \mathit{Proof}(P, T),$$

where $\mathit{Proof}(P, T)$ is the property that P is a valid proof of T.

Proof of Gödel's Theorem

Proof

Let $Theorem(T)$ be the property that a statement T is provable.

Then it can be written as:

$$Theorem(T) = \exists P : Proof(P, T),$$

where $Proof(P, T)$ is the property that P is a valid proof of T.

$Proof(P, T)$ is decidable,

Proof of Gödel's Theorem

Proof

Let $\mathit{Theorem}(T)$ be the property that a statement T is provable.

Then it can be written as:

$$\mathit{Theorem}(T) = \exists P : \mathit{Proof}(P, T),$$

where $\mathit{Proof}(P, T)$ is the property that P is a valid proof of T.

$\mathit{Proof}(P, T)$ is decidable, because we can check the proof line by line.

Proof of Gödel's Theorem

Proof

Let $Theorem(T)$ be the property that a statement T is provable.

Then it can be written as:

$$Theorem(T) = \exists P : Proof(P, T),$$

where $Proof(P, T)$ is the property that P is a valid proof of T.

$Proof(P, T)$ is decidable, because we can check the proof line by line.

Thus, the set of theorems is in **RE**.

Proof of Gödel's Theorem

Proof

Let $\mathit{Theorem}(T)$ be the property that a statement T is provable.

Then it can be written as:

$$\mathit{Theorem}(T) = \exists P : \mathit{Proof}(P, T),$$

where $\mathit{Proof}(P, T)$ is the property that P is a valid proof of T.

$\mathit{Proof}(P, T)$ is decidable, because we can check the proof line by line.

Thus, the set of theorems is in **RE**.

We assume that our formal system is powerful enough to talk about computation.

Proof of Gödel's Theorem

Proof

Let $Theorem(T)$ be the property that a statement T is provable.

Then it can be written as:

$$Theorem(T) = \exists P : Proof(P, T),$$

where $Proof(P, T)$ is the property that P is a valid proof of T.

$Proof(P, T)$ is decidable, because we can check the proof line by line.

Thus, the set of theorems is in **RE**.

We assume that our formal system is powerful enough to talk about computation.

We assume that it includes quantifiers like $\forall$ and $\exists$.

Proof of Gödel's Theorem

Proof

Let $Theorem(T)$ be the property that a statement T is provable.

Then it can be written as:

$$Theorem(T) = \exists P : Proof(P, T),$$

where $Proof(P, T)$ is the property that P is a valid proof of T.

$Proof(P, T)$ is decidable, because we can check the proof line by line.

Thus, the set of theorems is in **RE**.

We assume that our formal system is powerful enough to talk about computation.

We assume that it includes quantifiers like $\forall$ and $\exists$.

We assume that the theory can express statements like $Halts(\Pi, x)$.

Proof of Gödel's Theorem

Proof

Let $\mathit{Theorem}(T)$ be the property that a statement T is provable.

Then it can be written as:

$$\mathit{Theorem}(T) = \exists P : \mathit{Proof}(P, T),$$

where $\mathit{Proof}(P, T)$ is the property that P is a valid proof of T.

$\mathit{Proof}(P, T)$ is decidable, because we can check the proof line by line.

Thus, the set of theorems is in **RE**.

We assume that our formal system is powerful enough to talk about computation.

We assume that it includes quantifiers like $\forall$ and $\exists$.

We assume that the theory can express statements like $\mathit{Halts}(\Pi, x)$.

We assume that the axioms of the theory are strong enough to derive each step of a computation from the previous one.

Proof of Gödel's Theorem

Proof of Gödel's Theorem

Proof

Proof of Gödel's Theorem

Proof

We will get a truth about halting and non-halting programs that cannot be proved.

Proof of Gödel's Theorem

Proof

We will get a truth about halting and non-halting programs that cannot be proved.

If $\Pi(x)$ halts on its t-th step,

Proof of Gödel's Theorem

Proof

We will get a truth about halting and non-halting programs that cannot be proved.

If $\Pi(x)$ halts on its t-th step, then its computation is a proof of this fact, about t lines long.

Proof of Gödel's Theorem

Proof

We will get a truth about halting and non-halting programs that cannot be proved.

If $\Pi(x)$ halts on its t-th step, then its computation is a proof of this fact, about t lines long.

Thus,

$$\text{if } \mathit{Halts}(\Pi, x) \text{ is true, then it is provable.}$$

Proof of Gödel's Theorem

Proof

We will get a truth about halting and non-halting programs that cannot be proved.

If $\Pi(x)$ halts on its t-th step, then its computation is a proof of this fact, about t lines long.

Thus,

$$\text{if } \mathit{Halts}(\Pi, x) \text{ is true, then it is provable.}$$

What if Π does not hold?

Proof of Gödel's Theorem

Proof

We will get a truth about halting and non-halting programs that cannot be proved.

If $\Pi(x)$ halts on its t-th step, then its computation is a proof of this fact, about t lines long.

Thus,

$$\text{if } \mathit{Halts}(\Pi, x) \text{ is true, then it is provable.}$$

What if Π does not hold? Assume that all statements of the form $\overline{\mathit{Halts}(\Pi, x)}$ are provable.

Proof of Gödel's Theorem

Proof

We will get a truth about halting and non-halting programs that cannot be proved.

If $\Pi(x)$ halts on its t-th step, then its computation is a proof of this fact, about t lines long.

Thus,

$$\text{if } \mathit{Halts}(\Pi, x) \text{ is true, then it is provable.}$$

What if Π does not hold? Assume that all statements of the form $\overline{\mathit{Halts}(\Pi, x)}$ are provable.

Then we can solve the Halting Problem by doing two things in parallel:

Proof of Gödel's Theorem

Proof

We will get a truth about halting and non-halting programs that cannot be proved.

If $\Pi(x)$ halts on its t-th step, then its computation is a proof of this fact, about t lines long.

Thus,

$$\text{if } \mathit{Halts}(\Pi, x) \text{ is true, then it is provable.}$$

What if Π does not hold? Assume that all statements of the form $\overline{\mathit{Halts}(\Pi, x)}$ are provable.

Then we can solve the Halting Problem by doing two things in parallel: run $\Pi(x)$ to see if it halts,

Proof of Gödel's Theorem

Proof

We will get a truth about halting and non-halting programs that cannot be proved.

If $\Pi(x)$ halts on its t-th step, then its computation is a proof of this fact, about t lines long.

Thus,

$$\text{if } Halts(\Pi, x) \text{ is true, then it is provable.}$$

What if Π does not hold? Assume that all statements of the form $\overline{Halts(\Pi, x)}$ are provable.

Then we can solve the Halting Problem by doing two things in parallel: run $\Pi(x)$ to see if it halts, and looking for the proof that it will not.

Proof of Gödel's Theorem

Proof

We will get a truth about halting and non-halting programs that cannot be proved.

If $\Pi(x)$ halts on its t-th step, then its computation is a proof of this fact, about t lines long.

Thus,

$$\text{if } \mathit{Halts}(\Pi, x) \text{ is true, then it is provable.}$$

What if Π does not hold? Assume that all statements of the form $\overline{\mathit{Halts}(\Pi, x)}$ are provable.

Then we can solve the Halting Problem by doing two things in parallel: run $\Pi(x)$ to see if it halts, and looking for the proof that it will not.

Since the Halting Problem is undecidable, there must exist a statement of the form $\overline{\mathit{Halts}(\Pi, x)}$ that is not provable.

Proof of Gödel's Theorem

Proof

We will get a truth about halting and non-halting programs that cannot be proved.

If $\Pi(x)$ halts on its t-th step, then its computation is a proof of this fact, about t lines long.

Thus,

$$\text{if } \mathit{Halts}(\Pi, x) \text{ is true, then it is provable.}$$

What if Π does not hold? Assume that all statements of the form $\overline{\mathit{Halts}(\Pi, x)}$ are provable.

Then we can solve the Halting Problem by doing two things in parallel: run $\Pi(x)$ to see if it halts, and looking for the proof that it will not.

Since the Halting Problem is undecidable, there must exist a statement of the form $\overline{\mathit{Halts}(\Pi, x)}$ that is not provable.

In this case, neither $\mathit{Halts}(\Pi, x)$ nor $\overline{\mathit{Halts}(\Pi, x)}$ is a theorem.

Proof of Gödel's Theorem

Proof

We will get a truth about halting and non-halting programs that cannot be proved.

If $\Pi(x)$ halts on its t-th step, then its computation is a proof of this fact, about t lines long.

Thus,

$$\text{if } \mathit{Halts}(\Pi, x) \text{ is true, then it is provable.}$$

What if Π does not hold? Assume that all statements of the form $\overline{\mathit{Halts}(\Pi, x)}$ are provable.

Then we can solve the Halting Problem by doing two things in parallel: run $\Pi(x)$ to see if it halts, and looking for the proof that it will not.

Since the Halting Problem is undecidable, there must exist a statement of the form $\overline{\mathit{Halts}(\Pi, x)}$ that is not provable.

In this case, neither $\mathit{Halts}(\Pi, x)$ nor $\overline{\mathit{Halts}(\Pi, x)}$ is a theorem. It is independent of the axioms.

Possible Remedy

Possible Remedy

Idea

Possible Remedy

Idea

How about adding $\overline{Halts(\Pi, x)}$ to our list of axioms?

Possible Remedy

Idea

How about adding $\overline{Halts(\Pi, x)}$ to our list of axioms?

Then the fact that Π does not hold on input x becomes, trivially, a theorem of the system.

Possible Remedy

Idea

How about adding $\overline{Halts(\Pi, x)}$ to our list of axioms?

Then the fact that Π does not hold on input x becomes, trivially, a theorem of the system.

But then there will be another program Π' and an input x', such that $\overline{Halts(\Pi', x')}$ is true, but not provable in the new system, and so on.

Possible Remedy

Idea

How about adding $\overline{Halts(\Pi, x)}$ to our list of axioms?

Then the fact that Π does not hold on input x becomes, trivially, a theorem of the system.

But then there will be another program Π' and an input x', such that $\overline{Halts(\Pi', x')}$ is true, but not provable in the new system, and so on.

No finite set of axioms captures all the non-halting programs.

Possible Remedy

Idea

How about adding $\overline{Halts(\Pi, x)}$ to our list of axioms?

Then the fact that Π does not hold on input x becomes, trivially, a theorem of the system.

But then there will be another program Π' and an input x', such that $\overline{Halts(\Pi', x')}$ is true, but not provable in the new system, and so on.

No finite set of axioms captures all the non-halting programs.

For any formal system, there will be some truth that it cannot prove.

Clear definition of the notion of algorithm

Clear definition of the notion of algorithm

What is an algorithm?

Clear definition of the notion of algorithm

What is an algorithm?

How can we give a clear definition of the algorithm?

Clear definition of the notion of algorithm

What is an algorithm?

How can we give a clear definition of the algorithm?

Intuitively, a function is computable if it can be defined in terms of simpler functions, which are computable, too.

Clear definition of the notion of algorithm

What is an algorithm?

How can we give a clear definition of the algorithm?

Intuitively, a function is computable if it can be defined in terms of simpler functions, which are computable, too.

These simpler functions are defined in turn in terms of even simpler ones, and so on.

Clear definition of the notion of algorithm

What is an algorithm?

How can we give a clear definition of the algorithm?

Intuitively, a function is computable if it can be defined in terms of simpler functions, which are computable, too.

These simpler functions are defined in turn in terms of even simpler ones, and so on.

With this we reach a set of basic functions, for which no further explanation is necessary.

Clear definition of the notion of algorithm

What is an algorithm?

How can we give a clear definition of the algorithm?

Intuitively, a function is computable if it can be defined in terms of simpler functions, which are computable, too.

These simpler functions are defined in turn in terms of even simpler ones, and so on.

With this we reach a set of basic functions, for which no further explanation is necessary.

These basic functions form the atoms of computation.

Clear definition of the notion of algorithm

What is an algorithm?

How can we give a clear definition of the algorithm?

Intuitively, a function is computable if it can be defined in terms of simpler functions, which are computable, too.

These simpler functions are defined in turn in terms of even simpler ones, and so on.

With this we reach a set of basic functions, for which no further explanation is necessary.

These basic functions form the atoms of computation.

In terms of programming, they are the elementary operations that we can carry out in a single step.

Basic functions

Basic functions

The constant 0 and the successor function

Basic functions

The constant 0 and the successor function

The first basic function is:

$$0(x) = 0.$$

Basic functions

The constant 0 and the successor function

The first basic function is:

$$0(x) = 0.$$

The second basic function is:

$$S(x) = x + 1.$$

Basic functions

The constant 0 and the successor function

The first basic function is:

$$0(x) = 0.$$

The second basic function is:

$$S(x) = x + 1.$$

Remark

Basic functions

The constant 0 and the successor function

The first basic function is:

$$0(x) = 0.$$

The second basic function is:

$$S(x) = x + 1.$$

Remark

Strictly speaking, in order to use x on the right-side we also need to include the identity function $I(x) = x$.

Schemes

Schemes

Generating new functions

Schemes

Generating new functions

We need some schemes by which we can construct new functions from old ones.

Schemes

Generating new functions

We need some schemes by which we can construct new functions from old ones.

Composition

Schemes

Generating new functions

We need some schemes by which we can construct new functions from old ones.

Composition

If f and g are already defined, we can define a new function $h = f \circ g$ by

$$h(x_1, ..., x_n) = f(g(x_1, ..., x_n)).$$

Schemes

Generating new functions

We need some schemes by which we can construct new functions from old ones.

Composition

If f and g are already defined, we can define a new function $h = f \circ g$ by

$$h(x_1, ..., x_n) = f(g(x_1, ..., x_n)).$$

More generally, we allow functions to access each of their variables.

Schemes

Generating new functions

We need some schemes by which we can construct new functions from old ones.

Composition

If f and g are already defined, we can define a new function $h = f \circ g$ by

$$h(x_1, ..., x_n) = f(g(x_1, ..., x_n)).$$

More generally, we allow functions to access each of their variables. For instance, if $f(x_1, x_2)$, $g(x_1, x_2)$ and $m(x_1, x_2)$ are already defined, we can define

$$h(x_1, x_2, x_3) = f(g(x_1, x_2), m(x_3, x_1)).$$

Schemes

Generating new functions

We need some schemes by which we can construct new functions from old ones.

Composition

If f and g are already defined, we can define a new function $h = f \circ g$ by

$$h(x_1, ..., x_n) = f(g(x_1, ..., x_n)).$$

More generally, we allow functions to access each of their variables. For instance, if $f(x_1, x_2)$, $g(x_1, x_2)$ and $m(x_1, x_2)$ are already defined, we can define

$$h(x_1, x_2, x_3) = f(g(x_1, x_2), m(x_3, x_1)).$$

Remark

In terms of programming, composition lets us call previously defined functions as subroutines, using the output of one as the input of the other.

Schemes

Schemes

Primitive recursion

Schemes

Primitive recursion

If $f(x_1, ..., x_n)$ and $g(x_1, ..., x_n, y, z)$ are already defined,

Schemes

Primitive recursion

If $f(x_1, ..., x_n)$ and $g(x_1, ..., x_n, y, z)$ are already defined, we can define a new function $h(x_1, ..., x_n, y)$ as follows:

Schemes

Primitive recursion

If $f(x_1, \ldots, x_n)$ and $g(x_1, \ldots, x_n, y, z)$ are already defined, we can define a new function $h(x_1, \ldots, x_n, y)$ as follows:

$$h(x_1, \ldots, x_n, 0) = f(x_1, \ldots, x_n),$$

Schemes

Primitive recursion

If $f(x_1, \ldots, x_n)$ and $g(x_1, \ldots, x_n, y, z)$ are already defined, we can define a new function $h(x_1, \ldots, x_n, y)$ as follows:

$$h(x_1, \ldots, x_n, 0) = f(x_1, \ldots, x_n), \text{ and } h(x_1, \ldots, x_n, y + 1) = g(x_1, \ldots, x_n, y, h(x_1, \ldots, x_n, y)).$$

Schemes

Primitive recursion

If $f(x_1, ..., x_n)$ and $g(x_1, ..., x_n, y, z)$ are already defined, we can define a new function $h(x_1, ..., x_n, y)$ as follows:

$$h(x_1, ..., x_n, 0) = f(x_1, ..., x_n), \text{ and } h(x_1, ..., x_n, y + 1) = g(x_1, ..., x_n, y, h(x_1, ..., x_n, y)).$$

Remark

In terms of programming, this corresponds to a **for** *loop, when one iterates through the values of y.*

Examples

Examples

The addition function is computable

Examples

The addition function is computable

$$add(x, 0) = x$$

Examples

The addition function is computable

$$add(x, 0) = x \text{ and } add(x, y + 1) = S(add(x, y)).$$

Examples

The addition function is computable

$$add(x, 0) = x \text{ and } add(x, y + 1) = S(add(x, y)).$$

In standard language this will look as follows:

Examples

The addition function is computable

$$add(x, 0) = x \text{ and } add(x, y + 1) = S(add(x, y)).$$

In standard language this will look as follows:

$$x + 0 = x$$

Examples

The addition function is computable

$$add(x, 0) = x \text{ and } add(x, y + 1) = S(add(x, y)).$$

In standard language this will look as follows:

$$x + 0 = x \text{ and } x + (y + 1) = (x + y) + 1.$$

Examples

The addition function is computable

$$add(x, 0) = x \text{ and } add(x, y+1) = S(add(x, y)).$$

In standard language this will look as follows:

$$x + 0 = x \text{ and } x + (y+1) = (x+y) + 1.$$

The multiplication function is computable

Examples

The addition function is computable

$$add(x, 0) = x \text{ and } add(x, y+1) = S(add(x, y)).$$

In standard language this will look as follows:

$$x + 0 = x \text{ and } x + (y+1) = (x+y) + 1.$$

The multiplication function is computable

$$mult(x, 0) = 0$$

Examples

The addition function is computable

$$add(x, 0) = x \text{ and } add(x, y + 1) = S(add(x, y)).$$

In standard language this will look as follows:

$$x + 0 = x \text{ and } x + (y + 1) = (x + y) + 1.$$

The multiplication function is computable

$$mult(x, 0) = 0 \text{ and } mult(x, y + 1) = add(mult(x, y), x).$$

Primitive recursive functions

Primitive recursive functions

Definition

Primitive recursive functions

Definition

Functions that can be obtained from basic functions $0(x)$ and $S(x)$ by using composition and primitive recursion are called primitive recursive.

Primitive recursive functions

Definition

Functions that can be obtained from basic functions $0(x)$ and $S(x)$ by using composition and primitive recursion are called primitive recursive.

Many functions are primitive recursive

Primitive recursive functions

Definition

Functions that can be obtained from basic functions $0(x)$ and $S(x)$ by using composition and primitive recursion are called primitive recursive.

Many functions are primitive recursive

Addition, multiplication, subtraction, and even $prime(x)$ are primitive recursive.

Primitive recursive functions

Definition

Functions that can be obtained from basic functions $0(x)$ and $S(x)$ by using composition and primitive recursion are called primitive recursive.

Many functions are primitive recursive

Addition, multiplication, subtraction, and even $\mathit{prime}(x)$ are primitive recursive.

Are all computable functions primitive recursive?

Primitive recursive functions

Definition

Functions that can be obtained from basic functions $0(x)$ and $S(x)$ by using composition and primitive recursion are called primitive recursive.

Many functions are primitive recursive

Addition, multiplication, subtraction, and even $prime(x)$ are primitive recursive.

Are all computable functions primitive recursive?

One may wonder whether any computable function is primitive recursive?

Primitive recursive functions

Definition

Functions that can be obtained from basic functions $0(x)$ and $S(x)$ by using composition and primitive recursion are called primitive recursive.

Many functions are primitive recursive

Addition, multiplication, subtraction, and even $\mathit{prime}(x)$ are primitive recursive.

Are all computable functions primitive recursive?

One may wonder whether any computable function is primitive recursive? The answer is NO!

Primitive recursive functions

Definition

Functions that can be obtained from basic functions $0(x)$ and $S(x)$ by using composition and primitive recursion are called primitive recursive.

Many functions are primitive recursive

Addition, multiplication, subtraction, and even $prime(x)$ are primitive recursive.

Are all computable functions primitive recursive?

One may wonder whether any computable function is primitive recursive? The answer is NO!

Basic functions are defined everywhere.

Primitive recursive functions

Definition

Functions that can be obtained from basic functions $0(x)$ and $S(x)$ by using composition and primitive recursion are called primitive recursive.

Many functions are primitive recursive

Addition, multiplication, subtraction, and even $\mathit{prime}(x)$ are primitive recursive.

Are all computable functions primitive recursive?

One may wonder whether any computable function is primitive recursive? The answer is NO!

Basic functions are defined everywhere.

The schemes for construction of new functions do not change this property.

Primitive recursive functions

Definition

Functions that can be obtained from basic functions $0(x)$ and $S(x)$ by using composition and primitive recursion are called primitive recursive.

Many functions are primitive recursive

Addition, multiplication, subtraction, and even $prime(x)$ are primitive recursive.

Are all computable functions primitive recursive?

One may wonder whether any computable function is primitive recursive? The answer is NO!

Basic functions are defined everywhere.

The schemes for construction of new functions do not change this property.

The universal function, which is computable, is not defined everywhere.

Primitive recursive functions

Definition

Functions that can be obtained from basic functions $0(x)$ and $S(x)$ by using composition and primitive recursion are called primitive recursive.

Many functions are primitive recursive

Addition, multiplication, subtraction, and even $\mathit{prime}(x)$ are primitive recursive.

Are all computable functions primitive recursive?

One may wonder whether any computable function is primitive recursive? The answer is NO!

Basic functions are defined everywhere.

The schemes for construction of new functions do not change this property.

The universal function, which is computable, is not defined everywhere.

Hence it cannot be primitive recursive.

Explicit example

Explicit example

Ackerman's function

Explicit example

Ackerman's function

$$A_1(x, y) = x + y,$$

Explicit example

Ackerman's function

$$A_1(x, y) = x + y, A_n(x, 0) = 1$$

Explicit example

Ackerman's function

$A_1(x, y) = x + y, A_n(x, 0) = 1$ and $A_n(x, y) = A_{n-1}(x, A_n(x, y-1))$ if $y > 0$.

Explicit example

Ackerman's function

$A_1(x, y) = x + y$, $A_n(x, 0) = 1$ and $A_n(x, y) = A_{n-1}(x, A_n(x, y - 1))$ if $y > 0$.

Small values of n

Explicit example

Ackerman's function

$$A_1(x, y) = x + y, A_n(x, 0) = 1 \text{ and } A_n(x, y) = A_{n-1}(x, A_n(x, y - 1)) \text{ if } y > 0.$$

Small values of n

It can be shown that:

$$A_2(x, y) = x \cdot y,$$

Explicit example

Ackerman's function

$$A_1(x, y) = x + y, A_n(x, 0) = 1 \text{ and } A_n(x, y) = A_{n-1}(x, A_n(x, y-1)) \text{ if } y > 0.$$

Small values of n

It can be shown that:

$$A_2(x, y) = x \cdot y, A_3(x, y) = x^y.$$

Explicit example

Ackerman's function

$$A_1(x,y) = x + y, A_n(x,0) = 1 \text{ and } A_n(x,y) = A_{n-1}(x, A_n(x, y-1)) \text{ if } y > 0.$$

Small values of n

It can be shown that:

$$A_2(x,y) = x \cdot y, A_3(x,y) = x^y.$$

Theorem

Explicit example

Ackerman's function

$$A_1(x, y) = x + y, A_n(x, 0) = 1 \text{ and } A_n(x, y) = A_{n-1}(x, A_n(x, y - 1)) \text{ if } y > 0.$$

Small values of n

It can be shown that:

$$A_2(x, y) = x \cdot y, A_3(x, y) = x^y.$$

Theorem

For any primitive recursive function $f(y)$,

Explicit example

Ackerman's function

$$A_1(x, y) = x + y, A_n(x, 0) = 1 \text{ and } A_n(x, y) = A_{n-1}(x, A_n(x, y-1)) \text{ if } y > 0.$$

Small values of n

It can be shown that:

$$A_2(x, y) = x \cdot y, A_3(x, y) = x^y.$$

Theorem

For any primitive recursive function $f(y)$, there is an n, such that

Explicit example

Ackerman's function

$$A_1(x, y) = x + y, A_n(x, 0) = 1 \text{ and } A_n(x, y) = A_{n-1}(x, A_n(x, y-1)) \text{ if } y > 0.$$

Small values of n

It can be shown that:

$$A_2(x, y) = x \cdot y, A_3(x, y) = x^y.$$

Theorem

For any primitive recursive function $f(y)$, there is an n, such that

$$f(y) < A_n(2, y) \textit{ for all } y \geq 3.$$

What is missing?

What is missing?

Definition

What is missing?

Definition

Given a function $f(x_1, ..., x_n, y)$, which is already defined,

What is missing?

Definition

Given a function $f(x_1, ..., x_n, y)$, which is already defined, μ-recursion lets us define a new function $h(x_1, ..., x_n)$ as follows:

What is missing?

Definition

Given a function $f(x_1, ..., x_n, y)$, which is already defined, μ-recursion lets us define a new function $h(x_1, ..., x_n)$ as follows:

$$h(x_1, ..., x_n) = \mu_y f(x_1, ..., x_n, y)$$

What is missing?

Definition

Given a function $f(x_1, ..., x_n, y)$, which is already defined, μ-recursion lets us define a new function $h(x_1, ..., x_n)$ as follows:

$$h(x_1, ..., x_n) = \mu_y f(x_1, ..., x_n, y) = \min\{y : f(x_1, ..., x_n, y) = 0\}.$$

What is missing?

Definition

Given a function $f(x_1, ..., x_n, y)$, which is already defined, μ-recursion lets us define a new function $h(x_1, ..., x_n)$ as follows:

$$h(x_1, ..., x_n) = \mu_y f(x_1, ..., x_n, y) = \min\{y : f(x_1, ..., x_n, y) = 0\}.$$

Remark

What is missing?

Definition

Given a function $f(x_1, ..., x_n, y)$, which is already defined, μ-recursion lets us define a new function $h(x_1, ..., x_n)$ as follows:

$$h(x_1, ..., x_n) = \mu_y f(x_1, ..., x_n, y) = \min\{y : f(x_1, ..., x_n, y) = 0\}.$$

Remark

$h(x_1, ..., x_n)$ returns the smallest solution y to the equation $f(x_1, ..., x_n, y) = 0$.

What is missing?

Definition

Given a function $f(x_1, ..., x_n, y)$, which is already defined, μ-recursion lets us define a new function $h(x_1, ..., x_n)$ as follows:

$$h(x_1, ..., x_n) = \mu_y f(x_1, ..., x_n, y) = \min\{y : f(x_1, ..., x_n, y) = 0\}.$$

Remark

$h(x_1, ..., x_n)$ returns the smallest solution y to the equation $f(x_1, ..., x_n, y) = 0$.

Remark

What is missing?

Definition

Given a function $f(x_1, ..., x_n, y)$, which is already defined, μ-recursion lets us define a new function $h(x_1, ..., x_n)$ as follows:

$$h(x_1, ..., x_n) = \mu_y f(x_1, ..., x_n, y) = \min\{y : f(x_1, ..., x_n, y) = 0\}.$$

Remark

$h(x_1, ..., x_n)$ returns the smallest solution y to the equation $f(x_1, ..., x_n, y) = 0$.

Remark

In programming, this corresponds to the **while** *loop.*

Partial Recursive Functions

Partial Recursive Functions

Definition

Partial Recursive Functions

Definition

Functions that can be obtained from basic functions $0(x)$ and $S(x)$ by using composition, primitive recursion and μ-recursion are called partial recursive.

Partial Recursive Functions

Definition

Functions that can be obtained from basic functions $0(x)$ and $S(x)$ by using composition, primitive recursion and μ-recursion are called partial recursive.

Definition

Partial Recursive Functions

Definition

Functions that can be obtained from basic functions $0(x)$ and $S(x)$ by using composition, primitive recursion and μ-recursion are called partial recursive.

Definition

Partial recursive functions that are defined everywhere are called total recursive.

Partial Recursive Functions

Definition

Functions that can be obtained from basic functions $0(x)$ and $S(x)$ by using composition, primitive recursion and μ-recursion are called partial recursive.

Definition

Partial recursive functions that are defined everywhere are called total recursive.

Three types of recursion

$$\text{primitive recursive} \subset \text{total recursive} \subset \text{partial recursive}.$$

Kleene's Normal Form Theorem

Kleene's Normal Form Theorem

Number of μ-recursions

Kleene's Normal Form Theorem

Number of μ-recursions

What is the maximum number of μ-recursions that one needs to get an arbitrary partial recursive function?

Kleene's Normal Form Theorem

Number of μ-recursions

What is the maximum number of μ-recursions that one needs to get an arbitrary partial recursive function?

Theorem

Kleene's Normal Form Theorem

Number of μ-recursions

What is the maximum number of μ-recursions that one needs to get an arbitrary partial recursive function?

Theorem

There exist primitive recursive functions f and g,

Kleene's Normal Form Theorem

Number of μ-recursions

What is the maximum number of μ-recursions that one needs to get an arbitrary partial recursive function?

Theorem

There exist primitive recursive functions f and g, such that for every partial recursive function h

Kleene's Normal Form Theorem

Number of μ-recursions

What is the maximum number of μ-recursions that one needs to get an arbitrary partial recursive function?

Theorem

There exist primitive recursive functions f and g, such that for every partial recursive function h there exists p, such that

Kleene's Normal Form Theorem

Number of μ-recursions

What is the maximum number of μ-recursions that one needs to get an arbitrary partial recursive function?

Theorem

There exist primitive recursive functions f and g, such that for every partial recursive function h there exists p, such that

$$h(x_1, ..., x_n) = g(x_1, ..., x_n, \mu_y f(p, x_1, ..., x_n, y)).$$

Kleene's Normal Form Theorem

Number of μ-recursions

What is the maximum number of μ-recursions that one needs to get an arbitrary partial recursive function?

Theorem

There exist primitive recursive functions f and g, such that for every partial recursive function h there exists p, such that

$$h(x_1, ..., x_n) = g(x_1, ..., x_n, \mu_y f(p, x_1, ..., x_n, y)).$$

Remark

Any partial recursive function can be written with a single use of μ-recursion.

References

References

Books

References

Books

Ch. Moore, S. Mertens, The Nature of Computation, Oxford University Press (2011).