Computability and classification problems

Lecture 1:

1. Part 1: Computability and non-computability
2. Part 2: Computability, provability, and definability

Lecture 2:

1. Part 1: Index sets
2. Part 2: Applications to classification problems

Computability can be used to measure the complexity of a classification problem.

Computability can be used to prove that there is no classification at all.

We need a formal notion of computability.

Computability can be used to measure the complexity of a classification problem.

Computability can be used to prove that there is no classification at all.

We need a formal notion of computability.

Computability can be used to measure the complexity of a classification problem.

Computability can be used to prove that there is no classification at all.

We need a formal notion of computability.

Part 1: Computability and non-computability

Which functions $f : \mathbb{N} \to \mathbb{N}$ are computable?

An early idea (number theory): Study functions which are obtained from simpler functions using some sort of simple rules.

For instance, if we believe that f is computable and g is computable, then $f \circ g$ and $f + g$ (etc.) should also be computable.

Also, we all believe that $f(x) = x + 1$ and $g(x) = 17$ are computable.

Which functions $f : \mathbb{N} \to \mathbb{N}$ are computable?

An early idea (number theory): Study functions which are obtained from simpler functions using some sort of simple rules.

For instance, if we believe that f is computable and g is computable, then $f \circ g$ and $f + g$ (etc.) should also be computable.

Also, we all believe that $f(x) = x + 1$ and $g(x) = 17$ are computable.

Definition (Herbrand, Gödel 1933)

A function $f : \mathbb{N}^k \to \mathbb{N}$ is recursive if it can be obtained from the basic functions:

- constant functions $C_n^k(x_1, \ldots, x_k) = n$,
- the successor function $S(x) = x + 1$,
- projection functions $P_i^k(x_1, \ldots, x_k) = x_i$,

using finitely many applications of the following operators:

- composition
  $f(x_1, \ldots, x_k) = h(g_1(x_1, \ldots, x_k), \ldots, g_m(x_1, \ldots, x_k))$
- primitive recursion $f(0, x_1, \ldots, x_k) = g(x_1, \ldots, x_k)$
  $f(S(y), x_1, \ldots, x_k) = h(y, f(y, x_1, \ldots, x_k), x_1, \ldots, x_k)$
- minimization

$$\mu(f)(x_1, \ldots, x_k) = z \iff z \text{ is least s.t. } f(z, x_1, \ldots, x_k) = 0.$$

If we omit minimisation we get primitive recursive functions.

Definition (Herbrand, Gödel 1933)

A function $f : \mathbb{N}^k \to \mathbb{N}$ is recursive if it can be obtained from the basic functions:

- constant functions $C_n^k(x_1, \ldots, x_k) = n$,
- the successor function $S(x) = x + 1$,
- projection functions $P_i^k(x_1, \ldots, x_k) = x_i$,

using finitely many applications of the following operators:

- composition $f(x_1, \ldots, x_k) = h(g_1(x_1, \ldots, x_k), \ldots, g_m(x_1, \ldots, x_k))$
- primitive recursion $f(0, x_1, \ldots, x_k) = g(x_1, \ldots, x_k)$ $f(S(y), x_1, \ldots, x_k) = h(y, f(y, x_1, \ldots, x_k), x_1, \ldots, x_k)$
- minimization

$$\mu(f)(x_1, \ldots, x_k) = z \iff z \text{ is least s.t. } f(z, x_1, \ldots, x_k) = 0.$$

If we omit minimisation we get primitive recursive functions.

Definition (Herbrand, Gödel 1933)

A function $f : \mathbb{N}^k \to \mathbb{N}$ is recursive if it can be obtained from the basic functions:

- constant functions $C_n^k(x_1, \ldots, x_k) = n$,
- the successor function $S(x) = x + 1$,
- projection functions $P_i^k(x_1, \ldots, x_k) = x_i$,

using finitely many applications of the following operators:

- composition
  $f(x_1, \ldots, x_k) = h(g_1(x_1, \ldots, x_k), \ldots, g_m(x_1, \ldots, x_k))$
- primitive recursion $f(0, x_1, \ldots, x_k) = g(x_1, \ldots, x_k)$
  $f(S(y), x_1, \ldots, x_k) = h(y, f(y, x_1, \ldots, x_k), x_1, \ldots, x_k)$
- minimization

$$\mu(f)(x_1, \ldots, x_k) = z \iff z \text{ is least s.t. } f(z, x_1, \ldots, x_k) = 0.$$

If we omit minimisation we get primitive recursive functions.

Definition (Herbrand, Gödel 1933)

A function $f : \mathbb{N}^k \to \mathbb{N}$ is recursive if it can be obtained from the basic functions:

- constant functions $C_n^k(x_1, \ldots, x_k) = n$,
- the successor function $S(x) = x + 1$,
- projection functions $P_i^k(x_1, \ldots, x_k) = x_i$,

using finitely many applications of the following operators:

- composition
  $f(x_1, \ldots, x_k) = h(g_1(x_1, \ldots, x_k), \ldots, g_m(x_1, \ldots, x_k))$
- primitive recursion $f(0, x_1, \ldots, x_k) = g(x_1, \ldots, x_k)$
  $f(S(y), x_1, \ldots, x_k) = h(y, f(y, x_1, \ldots, x_k), x_1, \ldots, x_k)$
- minimization

$$\mu(f)(x_1, \ldots, x_k) = z \iff z \text{ is least s.t. } f(z, x_1, \ldots, x_k) = 0.$$

If we omit minimisation we get primitive recursive functions.

Definition (Herbrand, Gödel 1933)

A function $f : \mathbb{N}^k \to \mathbb{N}$ is recursive if it can be obtained from the basic functions:

- constant functions $C_n^k(x_1, \ldots, x_k) = n$,
- the successor function $S(x) = x + 1$,
- projection functions $P_i^k(x_1, \ldots, x_k) = x_i$,

using finitely many applications of the following operators:

- composition
  $f(x_1, \ldots, x_k) = h(g_1(x_1, \ldots, x_k), \ldots, g_m(x_1, \ldots, x_k))$
- primitive recursion $f(0, x_1, \ldots, x_k) = g(x_1, \ldots, x_k)$
  $f(S(y), x_1, \ldots, x_k) = h(y, f(y, x_1, \ldots, x_k), x_1, \ldots, x_k)$
- minimization

$$\mu(f)(x_1, \ldots, x_k) = z \iff z \text{ is least s.t. } f(z, x_1, \ldots, x_k) = 0.$$

If we omit minimisation we get primitive recursive functions.

Definition (Herbrand, Gödel 1933)

A function $f : \mathbb{N}^k \to \mathbb{N}$ is recursive if it can be obtained from the basic functions:

- constant functions $C_n^k(x_1, \ldots, x_k) = n$,
- the successor function $S(x) = x + 1$,
- projection functions $P_i^k(x_1, \ldots, x_k) = x_i$,

using finitely many applications of the following operators:

- composition
  $f(x_1, \ldots, x_k) = h(g_1(x_1, \ldots, x_k), \ldots, g_m(x_1, \ldots, x_k))$
- primitive recursion $f(0, x_1, \ldots, x_k) = g(x_1, \ldots, x_k)$
  $f(S(y), x_1, \ldots, x_k) = h(y, f(y, x_1, \ldots, x_k), x_1, \ldots, x_k)$
- minimization

$$\mu(f)(x_1, \ldots, x_k) = z \iff z \text{ is least s.t. } f(z, x_1, \ldots, x_k) = 0.$$

If we omit minimisation we get primitive recursive functions.

Definition (Herbrand, Gödel 1933)

A function $f : \mathbb{N}^k \to \mathbb{N}$ is recursive if it can be obtained from the basic functions:

- constant functions $C_n^k(x_1, \ldots, x_k) = n$,
- the successor function $S(x) = x + 1$,
- projection functions $P_i^k(x_1, \ldots, x_k) = x_i$,

using finitely many applications of the following operators:

- composition
  $f(x_1, \ldots, x_k) = h(g_1(x_1, \ldots, x_k), \ldots, g_m(x_1, \ldots, x_k))$
- primitive recursion $f(0, x_1, \ldots, x_k) = g(x_1, \ldots, x_k)$
  $f(S(y), x_1, \ldots, x_k) = h(y, f(y, x_1, \ldots, x_k), x_1, \ldots, x_k)$
- minimization
  $$\mu(f)(x_1, \ldots, x_k) = z \iff z \text{ is least s.t. } f(z, x_1, \ldots, x_k) = 0.$$

If we omit minimisation we get primitive recursive functions.

Definition (Herbrand, Gödel 1933)

A function $f : \mathbb{N}^k \to \mathbb{N}$ is recursive if it can be obtained from the basic functions:

- constant functions $C^k_n(x_1, \ldots, x_k) = n$,
- the successor function $S(x) = x + 1$,
- projection functions $P^k_i(x_1, \ldots, x_k) = x_i$,

using finitely many applications of the following operators:

- composition
  $f(x_1, \ldots, x_k) = h(g_1(x_1, \ldots, x_k), \ldots, g_m(x_1, \ldots, x_k))$
- primitive recursion $f(0, x_1, \ldots, x_k) = g(x_1, \ldots, x_k)$
  $f(S(y), x_1, \ldots, x_k) = h(y, f(y, x_1, \ldots, x_k), x_1, \ldots, x_k)$
- minimization
  $$\mu(f)(x_1, \ldots, x_k) = z \iff z \text{ is least s.t. } f(z, x_1, \ldots, x_k) = 0.$$

If we omit minimisation we get primitive recursive functions.

Definition (Herbrand, Gödel 1933)

A function $f : \mathbb{N}^k \to \mathbb{N}$ is recursive if it can be obtained from the basic functions:

- constant functions $C^k_n(x_1, \ldots, x_k) = n$,
- the successor function $S(x) = x + 1$,
- projection functions $P^k_i(x_1, \ldots, x_k) = x_i$,

using finitely many applications of the following operators:

- composition
  $f(x_1, \ldots, x_k) = h(g_1(x_1, \ldots, x_k), \ldots, g_m(x_1, \ldots, x_k))$
- primitive recursion $f(0, x_1, \ldots, x_k) = g(x_1, \ldots, x_k)$
  $f(S(y), x_1, \ldots, x_k) = h(y, f(y, x_1, \ldots, x_k), x_1, \ldots, x_k)$
- minimization
  $$\mu(f)(x_1, \ldots, x_k) = z \iff z \text{ is least s.t. } f(z, x_1, \ldots, x_k) = 0.$$

If we omit minimisation we get primitive recursive functions.

Playing with this definition is similar to writing a basic computer program in some very simple language.

Example

Informally, addition can be recursively defined as follows:

$add(0, x) = x,$

$add(n + 1, x) = add(n, x) + 1.$

Formally,

$add(0, x) = P^1_1(x),$

$add(S(n), x) = S(add(n, x)).$

People (goes back to Kronecker, formally Skolem) in the late 19th -early 20th century checked that all standard number-theoretic functions have recursive definitions.

Playing with this definition is similar to writing a basic computer program in some very simple language.

Example

Informally, addition can be recursively defined as follows:

$add(0, x) = x,$

$add(n + 1, x) = add(n, x) + 1.$

Formally,

$add(0, x) = P_1^1(x),$

$add(S(n), x) = S(add(n, x)).$

People (goes back to Kronecker, formally Skolem) in the late 19th -early 20th century checked that all standard number-theoretic functions have recursive definitions.

Playing with this definition is similar to writing a basic computer program in some very simple language.

Example

Informally, addition can be recursively defined as follows:

$add(0, x) = x,$

$add(n+1, x) = add(n, x) + 1.$

Formally,

$add(0, x) = P^1_1(x),$

$add(S(n), x) = S(add(n, x)).$

People (goes back to Kronecker, formally Skolem) in the late 19th -early 20th century checked that all standard number-theoretic functions have recursive definitions.

Note that when we formally define

$$\mu(f)(x_1,\dots,x_k) = z \iff z \text{ is least s.t. } f(z,x_1,\dots,x_k) = 0,$$

for some arguments $x_1,\dots,x_k$ there could be no such z.

This allows to consider partial recursive functions.

Why is this important?

Theorem

There is no list of all recursive functions $f_0, f_1, \dots$ such that

$$U(i,x) = \text{“the i-th recursive function applied to x”}$$

is itself recursive.

Proof.

Consider $V(i) = U(i,i) + 1 = S(U(i,i))$. If $V = f_j$, then

$$f_j(j) = V(j) = U(j,j) + 1 = f_j(j) + 1.$$

□

Note that when we formally define

$$\mu(f)(x_1,\dots,x_k) = z \iff z \text{ is least s.t. } f(z,x_1,\dots,x_k) = 0,$$

for some arguments $x_1,\dots,x_k$ there could be no such z.

This allows to consider partial recursive functions.

Why is this important?

Theorem

There is no list of all recursive functions $f_0, f_1, \dots$ such that

$$U(i,x) = \text{"the i-th recursive function applied to x"}$$

is itself recursive.

Proof.

Consider $V(i) = U(i,i) + 1 = S(U(i,i))$. If $V = f_j$, then

$$f_j(j) = V(j) = U(j,j) + 1 = f_j(j) + 1.$$

□

Note that when we formally define

$$\mu(f)(x_1, \ldots, x_k) = z \iff z \text{ is least s.t. } f(z, x_1, \ldots, x_k) = 0,$$

for some arguments $x_1, \ldots, x_k$ there could be no such z.

This allows to consider partial recursive functions.

Why is this important?

Theorem

There is no list of all recursive functions $f_0, f_1, \ldots$ such that

$$U(i, x) = \text{“the i-th recursive function applied to x”}$$

is itself recursive.

Proof.

Consider $V(i) = U(i, i) + 1 = S(U(i, i))$. If $V = f_j$, then

$$f_j(j) = V(j) = U(j, j) + 1 = f_j(j) + 1.$$

□

Note that when we formally define

$$\mu(f)(x_1, \ldots, x_k) = z \iff z \text{ is least s.t. } f(z, x_1, \ldots, x_k) = 0,$$

for some arguments $x_1, \ldots, x_k$ there could be no such z.

This allows to consider partial recursive functions.

Why is this important?

Theorem

There is no list of all recursive functions $f_0, f_1, \ldots$ such that

$$U(i, x) = \text{"the i-th recursive function applied to x"}$$

is itself recursive.

Proof.

Consider $V(i) = U(i, i) + 1 = S(U(i, i))$. If $V = f_j$, then

$$f_j(j) = V(j) = U(j, j) + 1 = f_j(j) + 1.$$

□

However, if we allow partial recursive functions then there is a *universal enumeration of all such functions:*

Theorem (Kleene normal form)

There exist primitive recursive T and U such that a partial function f is recursive if and only if there is a number e such that for all n

$$f(n) = U(\mu_x T(e, n, x)).$$

You have the right to ask:

How is this even related to computability?

The answer basically is:

In a way, this universal enumeration is your laptop.

However, if we allow partial recursive functions then there is a *universal enumeration of all such functions:*

Theorem (Kleene normal form)

There exist primitive recursive T and U such that a partial function f is recursive if and only if there is a number e such that for all n

$$f(n) = U(\mu_x T(e, n, x)).$$

You have the right to ask:

How is this even related to computability?

The answer basically is:

In a way, this universal enumeration is your laptop.

However, if we allow partial recursive functions then there is a *universal enumeration of all such functions:*

Theorem (Kleene normal form)

There exist primitive recursive T and U such that a partial function f is recursive if and only if there is a number e such that for all n

$$f(n) = U(\mu_x T(e, n, x)).$$

You have the right to ask:

How is this even related to computability?

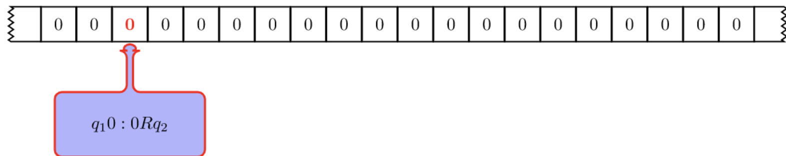
The answer basically is:

In a way, this universal enumeration is your laptop.

Definition (Turing 1936)

A Turing machine is a mathematical model of computation that defines an abstract machine composed of:

- a potentially infinite tape, and
- a working device with finitely many states



The tape has cells, each cell is either blank or has 0 or 1 in it.
The device can read the symbol in the cell it is currently observing, and based on its current state it can:

1. change the symbol in the cell,
2. move right or left,
3. change the state of the device.

Definition

A function $\mathbb{N} \to \mathbb{N}$ is Turing computable if there is a Turing machine T which, on input the binary representation of x on its tape, finishes its work with the binary representation of $f(x)$ written on its tape.

Theorem (Church, Turing)

A (partial) function $f : \mathbb{N} \to \mathbb{N}$ is recursive if, and only if, it is Turing computable.

Proof idea.

$\to$: By induction. Design a Turing machine for each elementary basic function and explain how to implement composition, primitive recursion, and minimisation.
$\leftarrow$: Design a primitive recursive predicate that says that number $p_0^{y_0} \dots p_n^{y_n}$ codes a valid computation $y_0, \dots, y_n$ of a Turing machine. Then use the minimisation operator to search for a halting computation, and then recover the output. □

Definition

A function $\mathbb{N} \to \mathbb{N}$ is Turing computable if there is a Turing machine T which, on input the binary representation of x on its tape, finishes its work with the binary representation of $f(x)$ written on its tape.

Theorem (Church, Turing)

A (partial) function $f : \mathbb{N} \to \mathbb{N}$ is recursive if, and only if, it is Turing computable.

Proof idea.

$\to$: By induction. Design a Turing machine for each elementary basic function and explain how to implement composition, primitive recursion, and minimisation.

$\leftarrow$: Design a primitive recursive predicate that says that number $p_0^{y_0} \dots p_n^{y_n}$ codes a valid computation $y_0, \dots, y_n$ of a Turing machine. Then use the minimisation operator to search for a halting computation, and then recover the output. $\square$

Definition

A function $\mathbb{N} \to \mathbb{N}$ is Turing computable if there is a Turing machine T which, on input the binary representation of x on its tape, finishes its work with the binary representation of $f(x)$ written on its tape.

Theorem (Church, Turing)

A (partial) function $f : \mathbb{N} \to \mathbb{N}$ is recursive if, and only if, it is Turing computable.

Proof idea.

$\to$: By induction. Design a Turing machine for each elementary basic function and explain how to implement composition, primitive recursion, and minimisation.

$\leftarrow$: Design a primitive recursive predicate that says that number $p_0^{y_0} \dots p_n^{y_n}$ codes a valid computation $y_0, \dots, y_n$ of a Turing machine. Then use the minimisation operator to search for a halting computation, and then recover the output. □

Here are several interesting consequences of this result and Kleene's normal form:

1. All Turing machines can be computably listed:

$$M_0, M_1, M_2, \ldots.$$

2. There exists *the Universal Turing Machine U*:

$$M_e(x) \equiv U(2^e 3^x).$$

The number e is called *the index* of M_e.

By the 1950-s Kleene, Markov, and many others have come up with many other models of computation.

All of these notions eventually were shown to be equivalent to (or even weaker than) Turing computability.

Church-Turing thesis: A function $f : \mathbb{N} \to \mathbb{N}$ can be algorithmically calculated if and only if it is Turing computable.

Now people could prove that some problems in mathematics are **not computable**.

By the 1950-s Kleene, Markov, and many others have come up with many other models of computation.

All of these notions eventually were shown to be equivalent to (or even weaker than) Turing computability.

Church-Turing thesis: A function $f : \mathbb{N} \to \mathbb{N}$ can be algorithmically calculated if and only if it is Turing computable.

Now people could prove that some problems in mathematics are **not computable**.

By the 1950-s Kleene, Markov, and many others have come up with many other models of computation.

All of these notions eventually were shown to be equivalent to (or even weaker than) Turing computability.

Church-Turing thesis: A function $f : \mathbb{N} \to \mathbb{N}$ can be algorithmically calculated if and only if it is Turing computable.

Now people could prove that some problems in mathematics are **not computable**.

By the 1950-s Kleene, Markov, and many others have come up with many other models of computation.

All of these notions eventually were shown to be equivalent to (or even weaker than) Turing computability.

Church-Turing thesis: A function $f : \mathbb{N} \to \mathbb{N}$ can be algorithmically calculated if and only if it is Turing computable.

Now people could prove that some problems in mathematics are **not computable**.

For example, let's look at the Halting Problem:

Given a description of a Turing machine, decide whether it halts on its own index.

We need to make it formal.

Definition

A set $X \subseteq \mathbb{N}$ is computable iff its characteristic function

$$\chi(n) = \begin{cases} 1 & \text{if } n \in X \\ 0 & \text{otherwise.} \end{cases}$$

is computable.

One possible formalisation is:

Question

Is the set $H = \{e : M_e(e) \text{ halts}\}$ computable?

For example, let's look at the Halting Problem:

Given a description of a Turing machine, decide whether it halts on its own index.

We need to make it formal.

Definition

A set $X \subseteq \mathbb{N}$ is computable iff its characteristic function

$$\chi(n) = \begin{cases} 1 & \text{if } n \in X \\ 0 & \text{otherwise.} \end{cases}$$

is computable.

One possible formalisation is:

Question

Is the set $H = \{e : M_e(e) \text{ halts}\}$ computable?

For example, let's look at the Halting Problem:

Given a description of a Turing machine, decide whether it halts on its own index.

We need to make it formal.

Definition

A set $X \subseteq \mathbb{N}$ is computable iff its characteristic function

$$\chi(n) = \begin{cases} 1 & \text{if } n \in X \\ 0 & \text{otherwise.} \end{cases}$$

is computable.

One possible formalisation is:

Question

Is the set $H = \{e : M_e(e) \text{ halts}\}$ computable?

Question

Is the set $H = \{e : M_e(e) \text{ halts}\}$ computable?

- Suppose the characteristic function h of H was computable.
- Define

$$g(e) = \begin{cases} 0 & \text{if } h(e) = 1 \text{ and } M_e(e) = U(2^e 3^e) = 1 \\ 1 & \text{otherwise.} \end{cases}$$

  The function is clearly computable (design a pseudo-code).
- Suppose j is such that $g \equiv M_j$.
- If $g(j) = M_j(j) = 0$ then this means that we are in the case when $h(e) = 1$ and $M_j(j) = U(2^j 3^j) = 1$, which is impossible.
- If $g(j) = M_j(j) = 1$ then we are again in the same case so $g(j) = 0$.

Question

Is the set $H = \{e : M_e(e) \text{ halts}\}$ computable?

- Suppose the characteristic function h of H was computable.
- Define

$$g(e) = \begin{cases} 0 & \text{if } h(e) = 1 \text{ and } M_e(e) = U(2^e 3^e) = 1 \\ 1 & \text{otherwise.} \end{cases}$$

  The function is clearly computable (design a pseudo-code).
- Suppose j is such that $g \equiv M_j$.
- If $g(j) = M_j(j) = 0$ then this means that we are in the case when $h(e) = 1$ and $M_j(j) = U(2^j 3^j) = 1$, which is impossible.
- If $g(j) = M_j(j) = 1$ then we are again in the same case so $g(j) = 0$.

Question

Is the set $H = \{e : M_e(e) \text{ halts}\}$ computable?

- Suppose the characteristic function h of H was computable.
- Define

$$g(e) = \begin{cases} 0 & \textit{if } h(e) = 1 \text{ and } M_e(e) = U(2^e 3^e) = 1 \\ 1 & \text{otherwise.} \end{cases}$$

  The function is clearly computable (design a pseudo-code).
- Suppose j is such that $g \equiv M_j$.
- If $g(j) = M_j(j) = 0$ then this means that we are in the case when $h(e) = 1$ and $M_j(j) = U(2^j 3^j) = 1$, which is impossible.
- If $g(j) = M_j(j) = 1$ then we are again in the same case so $g(j) = 0$.

Question

Is the set $H = \{e : M_e(e) \text{ halts}\}$ computable?

- Suppose the characteristic function h of H was computable.
- Define

$$g(e) = \begin{cases} 0 & \text{if } h(e) = 1 \text{ and } M_e(e) = U(2^e 3^e) = 1 \\ 1 & \text{otherwise.} \end{cases}$$

  The function is clearly computable (design a pseudo-code).
- Suppose j is such that $g \equiv M_j$.
- If $g(j) = M_j(j) = 0$ then this means that we are in the case when $h(e) = 1$ and $M_j(j) = U(2^j 3^j) = 1$, which is impossible.
- If $g(j) = M_j(j) = 1$ then we are again in the same case so $g(j) = 0$.

Question

Is the set $H = \{e : M_e(e) \text{ halts}\}$ computable?

- Suppose the characteristic function h of H was computable.
- Define

$$g(e) = \begin{cases} 0 & \text{if } h(e) = 1 \text{ and } M_e(e) = U(2^e 3^e) = 1 \\ 1 & \text{otherwise.} \end{cases}$$

  The function is clearly computable (design a pseudo-code).
- Suppose j is such that $g \equiv M_j$.
- If $g(j) = M_j(j) = 0$ then this means that we are in the case when $h(e) = 1$ and $M_j(j) = U(2^j 3^j) = 1$, which is impossible.
- If $g(j) = M_j(j) = 1$ then we are again in the same case so $g(j) = 0$.

Question

Is the set $H = \{e : M_e(e) \text{ halts}\}$ computable?

- Suppose the characteristic function h of H was computable.
- Define

$$g(e) = \begin{cases} 0 & \textit{if } h(e) = 1 \text{ and } M_e(e) = U(2^e 3^e) = 1 \\ 1 & \text{otherwise.} \end{cases}$$

  The function is clearly computable (design a pseudo-code).
- Suppose j is such that $g \equiv M_j$.
- If $g(j) = M_j(j) = 0$ then this means that we are in the case when $h(e) = 1$ and $M_j(j) = U(2^j 3^j) = 1$, which is impossible.
- If $g(j) = M_j(j) = 1$ then we are again in the same case so $g(j) = 0$.

We have proven:

Theorem

The Halting problem is undecidable.

As long as you can naturally *represent* your problem as a set of natural numbers, it makes sense to ask if the problem is decidable.

A few sample results:

Theorem (Novikov 1955, Boone 1958)

There is a finitely presented group in which the word problem is not computable.

Theorem (Markov 1958)

There is no algorithm which can decide for any pair of simplicial complexes whether they are homeomorphic.

We have proven:

Theorem

The Halting problem is undecidable.

As long as you can naturally *represent* your problem as a set of natural numbers, it makes sense to ask if the problem is decidable.

A few sample results:

Theorem (Novikov 1955, Boone 1958)

There is a finitely presented group in which the word problem is not computable.

Theorem (Markov 1958)

There is no algorithm which can decide for any pair of simplicial complexes whether they are homeomorphic.

Part 2: Computability, provability, and definability

§2.1 Hilbert's Tenth Problem. In logic we use **formulas.** There is a **recursive** formal definition of an arbitrary first-order formula which we omit.

Example

Let's restrict ourselves to the ordered semi-ring of natural numbers

$$(\mathbb{N}, +, \times, 0, 1).$$

The **signature** is $\{+, \times, 0, 1\}$. A typical quantifier-free formula $\phi(x, y)$ looks like:

$$((1+1)\times(x\times(x\times y)))+(1+1) = x \ \& \ (x+(1+1+1))\times(y\times x) = y.$$

Assuming the usual axioms it can be re-written as:

$$2x^2y - x + 2 = 0 \ \& \ (x+3)xy - y = 0$$

Evaluating such a formula essentially boils down to evaluating a bunch of polynomial equations.

§2.1 Hilbert's Tenth Problem.

In logic we use **formulas.** There is a **recursive** formal definition of an arbitrary first-order formula which we omit.

Example

Let's restrict ourselves to the ordered semi-ring of natural numbers

$$(\mathbb{N}, +, \times, 0, 1).$$

The **signature** is $\{+, \times, 0, 1\}$. A typical quantifier-free formula $\phi(x, y)$ looks like:

$$((1+1)\times(x\times(x\times y)))+(1+1) = x \;\&\; (x+(1+1+1))\times(y\times x) = y.$$

Assuming the usual axioms it can be re-written as:

$$2x^2y - x + 2 = 0 \;\&\; (x+3)xy - y = 0$$

Evaluating such a formula essentially boils down to evaluating a bunch of polynomial equations.

§2.1 Hilbert's Tenth Problem.

In logic we use **formulas**. There is a **recursive** formal definition of an arbitrary first-order formula which we omit.

Example

Let's restrict ourselves to the ordered semi-ring of natural numbers

$$(\mathbb{N}, +, \times, 0, 1).$$

The **signature** is $\{+, \times, 0, 1\}$. A typical quantifier-free formula $\phi(x, y)$ looks like:

$$((1+1)\times(x\times(x\times y)))+(1+1) = x \;\&\; (x+(1+1+1))\times(y\times x) = y.$$

Assuming the usual axioms it can be re-written as:

$$2x^2y - x + 2 = 0 \;\&\; (x+3)xy - y = 0$$

Evaluating such a formula essentially boils down to evaluating a bunch of polynomial equations.

Evaluating the corresponding existential projection of $\phi(x, y)$

$$\exists x \exists y \phi(x, y)$$

in $(\mathbb{N}, +, \times, 0, 1)$ is equivalent to evaluating

$$(\exists x)(\exists y)\ [(2x^2y - x + 2)^2 + ((x+3)xy - y)^2 = 0].$$

Thus, it is equivalent to **deciding** whether a polynomial with coefficients in $\mathbb{N}$ (a **Diophantine equation**) has a solution in $\mathbb{N}$.

Problem (Hilbert's Tenth Problem, 1900)

Is there an algorithm which, given a Diophantine equation, decides whether it has a solution (in $\mathbb{N}$)?

We have just argued that if the answer was "yes" then the **existential first-order theory** of $(\mathbb{N}, +, \times, 0, 1)$ would be computable.

Evaluating the corresponding existential projection of $\phi(x, y)$

$$\exists x \exists y \phi(x, y)$$

in $(\mathbb{N}, +, \times, 0, 1)$ is equivalent to evaluating

$$(\exists x)(\exists y)\ [(2x^2y - x + 2)^2 + ((x + 3)xy - y)^2 = 0].$$

Thus, it is equivalent to **deciding** whether a polynomial with coefficients in $\mathbb{N}$ (a **Diophantine equation**) has a solution in $\mathbb{N}$.

Problem (Hilbert's Tenth Problem, 1900)

Is there an algorithm which, given a Diophantine equation, decides whether it has a solution (in $\mathbb{N}$)?

We have just argued that if the answer was "yes" then the **existential first-order theory** of $(\mathbb{N}, +, \times, 0, 1)$ would be computable.

Evaluating the corresponding existential projection of $\phi(x, y)$

$$\exists x \exists y \phi(x, y)$$

in $(\mathbb{N}, +, \times, 0, 1)$ is equivalent to evaluating

$$(\exists x)(\exists y)\ [(2x^2y - x + 2)^2 + ((x+3)xy - y)^2 = 0].$$

Thus, it is equivalent to **deciding** whether a polynomial with coefficients in $\mathbb{N}$ (a **Diophantine equation**) has a solution in $\mathbb{N}$.

Problem (Hilbert's Tenth Problem, 1900)

Is there an algorithm which, given a Diophantine equation, decides whether it has a solution (in $\mathbb{N}$)?

We have just argued that if the answer was "yes" then the **existential first-order theory** of $(\mathbb{N}, +, \times, 0, 1)$ would be computable.

Some preliminary analysis:

Definition

A set is computably enumerable if it is equal to the domain of some (partial) computable function.

Example

Every Diophantine set $\{n : \exists x_1 \ldots \exists x_n p(n, x_0, \ldots, x_n) = 0\}$ is computably enumerable.

Proof.

Define a partial computable function f as follows. List all n-tuples of natural numbers and test whether $p(n, x_0, \ldots, x_n) = 0$ for a given tuple $\bar{x}$. If yes then output 1. □

Example

The Halting problem $\{e : M_e(e) \text{ halts}\}$ is computably enumerable.

Some preliminary analysis:

Definition

A set is computably enumerable if it is equal to the domain of some (partial) computable function.

Example

Every Diophantine set $\{n : \exists x_1 \dots \exists x_n p(n, x_0, \dots, x_n) = 0\}$ is computably enumerable.

Proof.

Define a partial computable function f as follows. List all n-tuples of natural numbers and test whether $p(n, x_0, \dots, x_n) = 0$ for a given tuple $\bar{x}$. If yes then output 1. □

Example

The Halting problem $\{e : M_e(e) \text{ halts}\}$ is computably enumerable.

Some preliminary analysis:

Definition

A set is computably enumerable if it is equal to the domain of some (partial) computable function.

Example

Every Diophantine set $\{n : \exists x_1 \ldots \exists x_n p(n, x_0, \ldots, x_n) = 0\}$ is computably enumerable.

Proof.

Define a partial computable function f as follows. List all n-tuples of natural numbers and test whether $p(n, x_0, \ldots, x_n) = 0$ for a given tuple $\bar{x}$. If yes then output 1. □

Example

The Halting problem $\{e : M_e(e) \text{ halts}\}$ is computably enumerable.

Some preliminary analysis:

Definition

A set is computably enumerable if it is equal to the domain of some (partial) computable function.

Example

Every Diophantine set $\{n : \exists x_1 \ldots \exists x_n p(n, x_0, \ldots, x_n) = 0\}$ is computably enumerable.

Proof.

Define a partial computable function f as follows. List all n-tuples of natural numbers and test whether $p(n, x_0, \ldots, x_n) = 0$ for a given tuple $\bar{x}$. If yes then output 1. □

Example

The Halting problem $\{e : M_e(e) \text{ halts}\}$ is computably enumerable.

Theorem (Matiyasevich–Robinson–Davis–Putnam, finished in 1970)

Every computably enumerable set is Diophantine.

Proof.

Hard. □

It follows that the computably enumerable sets are exactly the existentially definable subsets of $\mathbb{N}$!

Theorem (Matiyasevich–Robinson–Davis–Putnam, finished in 1970)

Every computably enumerable set is Diophantine.

Proof.

Hard. □

It follows that the computably enumerable sets are exactly the existentially definable subsets of $\mathbb{N}$!

Theorem (Matiyasevich–Robinson–Davis–Putnam, finished in 1970)

Every computably enumerable set is Diophantine.

Proof.

Hard. □

It follows that the computably enumerable sets are exactly the existentially definable subsets of $\mathbb{N}$!

Recall that there exists the Universal Turing Machine:

Corollary

There exists a polynomial $u(x, n, y_0, \ldots, y_k)$ with positive integer coefficients such that a set X is computably enumerable (Diophantine) if, and only if, for some $e \in \mathbb{N}$:

$$X = \{x : \exists \bar{y}\, u(x, e, \bar{y}) = 0\}.$$

Now we prove the unsolvability of Hilbert's Tenth Problem.

Recall that there exists the Universal Turing Machine:

Corollary

There exists a polynomial $u(x, n, y_0, \ldots, y_k)$ with positive integer coefficients such that a set X is computably enumerable (Diophantine) if, and only if, for some $e \in \mathbb{N}$:

$$X = \{x : \exists \bar{y}\, u(x, e, \bar{y}) = 0\}.$$

Now we prove the unsolvability of Hilbert's Tenth Problem.

Theorem (Strong unsolvability of Hilbert's Xth Problem)

There is no algorithm which, given e and x, decides whether

$$\exists \bar{y}\,[u(x,e,\bar{y})=0].$$

Proof.

- For a fixed e, define

$$W_e=\{x:\exists \bar{y}\,u(x,e,\bar{y})=0\}=dom\,M_e.$$

- We need to show: $Z=\{2^x3^e : x\in W_e\}$ is not computable.
- The Halting Problem H is computably enumerable but not computable.
- Fix j such that

$$H=W_j.$$

- If Z was computable then so would be its projection on W_j.

Theorem (Strong unsolvability of Hilbert's Xth Problem)

There is no algorithm which, given e and x, decides whether

$$\exists \bar{y}\,[u(x, e, \bar{y}) = 0].$$

Proof.

- For a fixed e, define

$$W_e = \{x : \exists \bar{y} u(x, e, \bar{y}) = 0\} = dom\, M_e.$$

- We need to show: $Z = \{2^x 3^e : x \in W_e\}$ is not computable.
- The Halting Problem H is computably enumerable but not computable.
- Fix j such that

$$H = W_j.$$

- If Z was computable then so would be its projection on W_j.

Theorem (Strong unsolvability of Hilbert's Xth Problem)

There is no algorithm which, given e and x, decides whether

$$\exists \bar{y}\,[u(x, e, \bar{y}) = 0].$$

Proof.

- For a fixed e, define

$$W_e = \{x : \exists \bar{y} u(x, e, \bar{y}) = 0\} = dom\, M_e.$$

- We need to show: $Z = \{2^x 3^e : x \in W_e\}$ is not computable.
- The Halting Problem H is computably enumerable but not computable.
- Fix j such that

$$H = W_j.$$

- If Z was computable then so would be its projection on W_j.

Theorem (Strong unsolvability of Hilbert's Xth Problem)

There is no algorithm which, given e and x, decides whether

$$\exists \bar{y}\,[u(x, e, \bar{y}) = 0].$$

Proof.

- For a fixed e, define

$$W_e = \{x : \exists \bar{y} u(x, e, \bar{y}) = 0\} = dom\, M_e.$$

- We need to show: $Z = \{2^x 3^e : x \in W_e\}$ is not computable.
- The Halting Problem H is computably enumerable but not computable.
- Fix j such that

$$H = W_j.$$

- If Z was computable then so would be its projection on W_j.

Theorem (Strong unsolvability of Hilbert's Xth Problem)

There is no algorithm which, given e and x, decides whether

$$\exists \bar{y}\, [u(x, e, \bar{y}) = 0].$$

Proof.

- For a fixed e, define

$$W_e = \{x : \exists \bar{y} u(x, e, \bar{y}) = 0\} = dom\, M_e.$$

- We need to show: $Z = \{2^x 3^e : x \in W_e\}$ is not computable.
- The Halting Problem H is computably enumerable but not computable.
- Fix j such that

$$H = W_j.$$

- If Z was computable then so would be its projection on W_j.

Theorem (Strong unsolvability of Hilbert's Xth Problem)

There is no algorithm which, given e and x, decides whether

$$\exists \bar{y}\,[u(x, e, \bar{y}) = 0].$$

Proof.

- For a fixed e, define

$$W_e = \{x : \exists \bar{y} u(x, e, \bar{y}) = 0\} = dom\, M_e.$$

- We need to show: $Z = \{2^x 3^e : x \in W_e\}$ is not computable.
- The Halting Problem H is computably enumerable but not computable.
- Fix j such that

$$H = W_j.$$

- If Z was computable then so would be its projection on W_j.

§2.2 Gödel's incompleteness.

Let $H = W_j$, as before.

Theorem (Gödel's incompleteness theorem)

There is a natural number n such that

$$\forall \bar{y} u(n, j, \bar{y}) \neq 0$$

holds but it is not provable in ZFC (the standard set of mathematical axioms).

Proof:

1. ZFC has a computable set of axioms.
2. Every proof is a finite sequence of formulae

$$\phi_0, \dots, \phi_k,$$

where each ϕ_i is either an axiom or is obtained from the previous ϕ_j, $j < i$, using an application of a logical rule (such as modus ponens).

§2.2 Gödel's incompleteness.

Let $H = W_j$, as before.

Theorem (Gödel's incompleteness theorem)

There is a natural number n such that

$$\forall \bar{y} u(n, j, \bar{y}) \neq 0$$

holds but it is not provable in ZFC (the standard set of mathematical axioms).

Proof:

1. ZFC has a computable set of axioms.
2. Every proof is a finite sequence of formulae

$$\phi_0, \dots, \phi_k,$$

where each ϕ_i is either an axiom or is obtained from the previous ϕ_j, $j < i$, using an application of a logical rule (such as modus ponens).

§2.2 Gödel's incompleteness.

Let $H = W_j$, as before.

Theorem (Gödel's incompleteness theorem)

There is a natural number n such that

$$\forall \bar{y} u(n, j, \bar{y}) \neq 0$$

holds but it is not provable in ZFC (the standard set of mathematical axioms).

Proof:

1. ZFC has a computable set of axioms.
2. Every proof is a finite sequence of formulae

$$\phi_0, \ldots, \phi_k,$$

where each ϕ_i is either an axiom or is obtained from the previous ϕ_j, $j < i$, using an application of a logical rule (such as modus ponens).

Proof (continued):

3. This makes the set of provable first-order statements in the arithmetic computably enumerable.
4. In particular, we can computably enumerate provable statements of the form $\forall \bar{y} u(n, j, \bar{y}) \neq 0$.
5. If for all n such that $\forall \bar{y} u(n, j, \bar{y}) \neq 0$ this fact was provable, then it would imply that

$$\bar{W}_j = \bar{H} = \mathbb{N} \setminus H$$

is computably enumerable.
6. To finish the theorem, it is sufficient to recall that H is not computable and also prove:

Lemma (Complementation Theorem)

If both X and $\bar{X}$ are computably enumerable, then X is computable.

Proof (continued):

3. This makes the set of provable first-order statements in the arithmetic computably enumerable.
4. In particular, we can computably enumerate provable statements of the form $\forall \bar{y} u(n, j, \bar{y}) \neq 0$.
5. If for all n such that $\forall \bar{y} u(n, j, \bar{y}) \neq 0$ this fact was provable, then it would imply that

$$\bar{W}_j = \bar{H} = \mathbb{N} \setminus H$$

is computably enumerable.
6. To finish the theorem, it is sufficient to recall that H is not computable and also prove:

Lemma (Complementation Theorem)

If both X and $\bar{X}$ are computably enumerable, then X is computable.

Proof (continued):

3. This makes the set of provable first-order statements in the arithmetic computably enumerable.
4. In particular, we can computably enumerate provable statements of the form $\forall \bar{y} u(n, j, \bar{y}) \neq 0$.
5. If for all n such that $\forall \bar{y} u(n, j, \bar{y}) \neq 0$ this fact was provable, then it would imply that

$$\bar{W}_j = \bar{H} = \mathbb{N} \setminus H$$

is computably enumerable.

6. To finish the theorem, it is sufficient to recall that H is not computable and also prove:

Lemma (Complementation Theorem)

If both X and $\bar{X}$ are computably enumerable, then X is computable.

Proof (continued):

3. This makes the set of provable first-order statements in the arithmetic computably enumerable.
4. In particular, we can computably enumerate provable statements of the form $\forall \bar{y} u(n, j, \bar{y}) \neq 0$.
5. If for all n such that $\forall \bar{y} u(n, j, \bar{y}) \neq 0$ this fact was provable, then it would imply that

$$\bar{W}_j = \bar{H} = \mathbb{N} \setminus H$$

is computably enumerable.
6. To finish the theorem, it is sufficient to recall that H is not computable and also prove:

Lemma (Complementation Theorem)

If both X and $\bar{X}$ are computably enumerable, then X is computable.

Proof of the lemma.

Initiate the enumeration of both X and $\bar{X}$ and see which one contains a given number x. □

This proves Gödel's incompleteness theorem. □

So there is a first-order fact about some fixed polynomial which is **true** but is **not provable**!

Proof of the lemma.

Initiate the enumeration of both X and $\bar{X}$ and see which one contains a given number x. □

This proves Gödel's incompleteness theorem. □

So there is a first-order fact about some fixed polynomial which is **true** but is **not provable**!

§2.3 The Arithmetical Hierarchy.

Identify sets with their characteristic functions.

Definition

We say that a set Y is computable relative to a set Z, written $Y \leq_T Z$, if there is an oracle Turing machine M_e such that

$$Y \equiv M_e^Z.$$

Think of a hard drive containing Z. Also, think of the keyboard (you are the oracle!).

Here is a stronger version of this:

Definition

$Y \leq_1 Z$ if there is a (total) 1-1 computable f such that

$$x \in Y \iff f(x) \in Z.$$

§2.3 The Arithmetical Hierarchy.

Identify sets with their characteristic functions.

Definition

We say that a set Y is computable relative to a set Z, written $Y \leq_T Z$, if there is an oracle Turing machine M_e such that

$$Y \equiv M_e^Z.$$

Think of a hard drive containing Z. Also, think of the keyboard (you are the oracle!).

Here is a stronger version of this:

Definition

$Y \leq_1 Z$ if there is a (total) 1-1 computable f such that

$$x \in Y \iff f(x) \in Z.$$

§2.3 The Arithmetical Hierarchy.

Identify sets with their characteristic functions.

Definition

We say that a set Y is computable relative to a set Z, written $Y \leq_T Z$, if there is an oracle Turing machine M_e such that

$$Y \equiv M_e^Z.$$

Think of a hard drive containing Z. Also, think of the keyboard (you are the oracle!).
Here is a stronger version of this:

Definition

$Y \leq_1 Z$ if there is a (total) 1-1 computable f such that

$$x \in Y \iff f(x) \in Z.$$

Definition

The Turing jump of $X \subseteq \mathbb{N}$ is the set

$$X' = \{e : M_e^X(e) \text{ halts}\}.$$

In particular, $\emptyset' = H = \{e : M_e(e) \text{ halts}\}$, and

$$0^{(n+1)} = (0^{(n)})'.$$

Definition

Define the **arithmetical classes** Σ_n^0 and Π_n^0:

1. $X \in \Sigma_n^0 \iff X \leq_1 0^{(n)}$,
2. $X \in \Pi_n^0 \iff X \leq_1 \overline{0^{(n)}} = \mathbb{N} \setminus 0^{(n)}$,
3. $X \in \Delta_n^0 \iff X \in (\Sigma_n^0 \cap \Pi_n^0)$.

Definition

The Turing jump of $X \subseteq \mathbb{N}$ is the set

$$X' = \{e : M_e^X(e) \text{ halts}\}.$$

In particular, $\emptyset' = H = \{e : M_e(e) \text{ halts}\}$, and

$$0^{(n+1)} = (0^{(n)})'.$$

Definition

Define the **arithmetical classes** Σ_n^0 and Π_n^0:

1. $X \in \Sigma_n^0 \iff X \leq_1 0^{(n)}$,
2. $X \in \Pi_n^0 \iff X \leq_1 \overline{0^{(n)}} = \mathbb{N} \setminus 0^{(n)}$,
3. $X \in \Delta_n^0 \iff X \in (\Sigma_n^0 \cap \Pi_n^0)$.

Definition

The Turing jump of $X \subseteq \mathbb{N}$ is the set

$$X' = \{e : M_e^X(e) \text{ halts}\}.$$

In particular, $\emptyset' = H = \{e : M_e(e) \text{ halts}\}$, and

$$0^{(n+1)} = (0^{(n)})'.$$

Definition

Define the **arithmetical classes** Σ_n^0 and Π_n^0:

1. $X \in \Sigma_n^0 \iff X \leq_1 0^{(n)}$,
2. $X \in \Pi_n^0 \iff X \leq_1 \overline{0^{(n)}} = \mathbb{N} \setminus 0^{(n)}$,
3. $X \in \Delta_n^0 \iff X \in (\Sigma_n^0 \cap \Pi_n^0)$.

Theorem

For a set $X \subseteq \mathbb{N}$ and $n > 0$, the following are equivalent:

1. $X \in \Sigma^0_n$;
2. X is computably enumerable relative to $0^{(n-1)}$:

$$\exists e \;\; X = dom\, M_e^{0^{(n-1)}}.$$

3. There is a computable function f of $n+1$ arguments such that

$$X = \{z : \exists x_1 \forall x_2 \ldots Q x_n (f(z, x_1, \ldots, x_n) = 0)\}$$

In fact, f can be replaced with a polynomial over $\mathbb{N}$.

Theorem

For a set $X \subseteq \mathbb{N}$ and $n > 0$, the following are equivalent:

1. $X \in \Sigma^0_n$;
2. X is computably enumerable relative to $0^{(n-1)}$:

$$\exists e \;\; X = dom\, M_e^{0^{(n-1)}}.$$

3. There is a computable function f of $n+1$ arguments such that

$$X = \{z : \exists x_1 \forall x_2 \dots Q x_n (f(z, x_1, \dots, x_n) = 0)\}$$

In fact, f can be replaced with a polynomial over $\mathbb{N}$.

Theorem

For a set $X \subseteq \mathbb{N}$ and $n > 0$, the following are equivalent:

1. $X \in \Sigma^0_n$;
2. X is computably enumerable relative to $0^{(n-1)}$:

$$\exists e \;\; X = dom\, M_e^{0^{(n-1)}}.$$

3. There is a computable function f of $n+1$ arguments such that

$$X = \{z : \exists x_1 \forall x_2 \ldots Q x_n (f(z, x_1, \ldots, x_n) = 0)\}$$

In fact, f can be replaced with a polynomial over $\mathbb{N}$.

Theorem

For a set $X \subseteq \mathbb{N}$ and $n > 0$, the following are equivalent:

1. $X \in \Sigma^0_n$;
2. X is computably enumerable relative to $0^{(n-1)}$:

$$\exists e \;\; X = dom\, M_e^{0^{(n-1)}}.$$

3. There is a computable function f of $n+1$ arguments such that

$$X = \{z : \exists x_1 \forall x_2 \ldots Q x_n (f(z, x_1, \ldots, x_n) = 0)\}$$

In fact, f can be replaced with a polynomial over $\mathbb{N}$.

Theorem

For a set $X \subseteq \mathbb{N}$ and $n > 0$, the following are equivalent:

1. $X \in \Sigma^0_n$;
2. X is computably enumerable relative to $0^{(n-1)}$:

$$\exists e \;\; X = dom\, M_e^{0^{(n-1)}}.$$

3. There is a computable function f of $n+1$ arguments such that

$$X = \{z : \exists x_1 \forall x_2 \ldots Q x_n (f(z, x_1, \ldots, x_n) = 0)\}$$

In fact, f can be replaced with a polynomial over $\mathbb{N}$.

Theorem

For a set $X \subseteq \mathbb{N}$ and $n > 0$, the following are equivalent:

1. $X \in \Delta^0_n$;
2. X is computable relative to $0^{(n-1)}$:

$$\exists e \;\; X = M_e^{0^{(n-1)}}.$$

3. There exist computable functions f and g of $n+1$ arguments such that

$$X = \{z : \exists x_1 \forall x_2 \ldots Qx_n(f(z, x_1, \ldots, x_n) = 0)\}$$

and

$$X = \{z : \forall x_1 \exists x_2 \ldots Rx_n(g(z, x_1, \ldots, x_n) = 0)\}.$$

Theorem

For a set $X \subseteq \mathbb{N}$ and $n > 0$, the following are equivalent:

1. $X \in \Delta^0_n$;
2. X is computable relative to $0^{(n-1)}$:
$$\exists e \;\; X = M_e^{0^{(n-1)}}.$$
3. There exist computable functions f and g of $n+1$ arguments such that
$$X = \{z : \exists x_1 \forall x_2 \ldots Qx_n (f(z, x_1, \ldots, x_n) = 0)\}$$
and
$$X = \{z : \forall x_1 \exists x_2 \ldots Rx_n (g(z, x_1, \ldots, x_n) = 0)\}.$$

Theorem

For a set $X \subseteq \mathbb{N}$ and $n > 0$, the following are equivalent:

1. $X \in \Delta^0_n$;
2. X is computable relative to $0^{(n-1)}$:

$$\exists e \;\; X = M_e^{0^{(n-1)}}.$$

3. There exist computable functions f and g of $n+1$ arguments such that

$$X = \{z : \exists x_1 \forall x_2 \ldots Qx_n (f(z, x_1, \ldots, x_n) = 0)\}$$

and

$$X = \{z : \forall x_1 \exists x_2 \ldots Rx_n (g(z, x_1, \ldots, x_n) = 0)\}.$$

Theorem

For a set $X \subseteq \mathbb{N}$ and $n > 0$, the following are equivalent:

1. $X \in \Delta_n^0$;
2. X is computable relative to $0^{(n-1)}$:

$$\exists e \ X = M_e^{0^{(n-1)}}.$$

3. There exist computable functions f and g of $n+1$ arguments such that

$$X = \{z : \exists x_1 \forall x_2 \ldots Q x_n (f(z, x_1, \ldots, x_n) = 0)\}$$

and

$$X = \{z : \forall x_1 \exists x_2 \ldots R x_n (g(z, x_1, \ldots, x_n) = 0)\}.$$

The famous diagram (in the Soare's book)