The Foundations of Computability Theory

Borut Robič

University of Ljubljana
Slovenia, 2015

Forward

- About Computability Theory
- Rules of the game
- Exams

Contents…

…Contents…

...Contents

Part I. THE ROOTS OF COMPUTABILITY THEORY

- Ch 1 Introduction
- Ch 2 The foundational crisis of mathematics
- Ch 3 Formalism
- Ch 4 Hilbert's attempt at recovery

Ch 1. Introduction

- **Definition.** (algorithm intuitively) An algorithm for solving a problem is a finite set of instructions that lead the processor, in a finite number of steps, from the input data of the problem to the corresponding solution.

Here:

- algorithm … a *finite* sequence of instructions
  instruction … *simple* enough, *unambiguous*, precisely *defines the next* instruction
  processor … a device capable of *mechanically* following, *interpreting*, and *executing* instructions while using *no self-reflection or external help*
  computation … a sequence of instruction executions (steps)

Algorithms and Computations Before the 20th Century

- Euclid 325–265 (Eudoxus, 408–355) B.C.E. : Euclid's algorithm
- India 1st–4th century: positional decimal number system
- Al-Khwarizmi, 9th century: algorithms for solving equations
- Europe 17th century: Schickard, Pascal: the machine for addition/subtraction of natural numbers; Leibniz: the concept of arithmetization (the universal language and computing machine)
- Europe 19th century: Babbage's concept of different programs for different problems (the analytical machine)

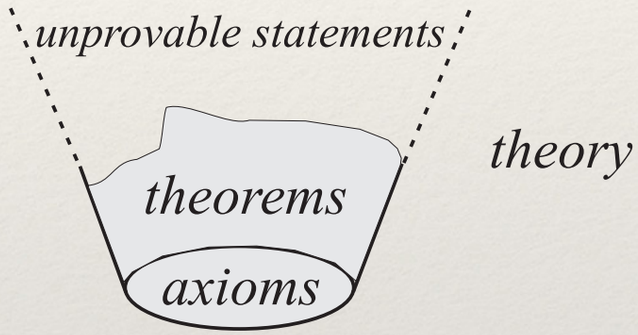
But:

- The concept of the algorithm remained at the intuitive level. The need for a formal definition arose in the 20th century.

Ch 2. The Foundational Crisis of Mathematics

Axiomatic Method

- To treat a field of interest, we first select
  - basic notions … notions fundamental to the field
  - axioms … assert properties of basic notions
  - initial theory = basic notions + axioms
- Then the initial theory is further developed
  - development of the theory = the process of defining new notions + proving new theorems
  - definition… introduces new notions using basic and/or defined notions
  - proof … finite sequence of mental steps that shows that a given statement follows from the axioms and/or already proven statements (theorems)
  - theory = basic + defined notions + axioms + theorems + unprovable statements



Evident Axiomatic Systems

- from Euclid to mid-19th century: axioms are evident, i.e., in perfect agreement with human experience in the field of interest
- **but**, in the 19th century: experience and intuition can be misleading
  - e.g., Euclid's fifth axiom (Parallel Postulate), although self-evident, *has* alternatives; these give rise to non-Euclidean geometries, parts of reality!

Hypothetical Axiomatic Systems

- basic notions … their real nature is no longer important
- axioms = hypotheses; needn't be self-evident/in agreement with reality
- there is *no obvious link* between the theory and reality
- the reasonableness and usefulness of the theory depends on the successful interpretations (applications of theory on parts of reality)

Cantor's Naive Set Theory

- **Basic notions**
  - (Cantor's set) A set is any collection of definite, distinguishable objects of our intuition or of our intellect to be conceived as a whole.
  - (Membership relation $\in$) A member of a set is any object that is in the set. An object either is or it is not a member of the set. (*Law of Excluded Middle*)
- **Axioms**
  - (Extensionality) A set is completely determined by its members.
    - e.g., $S = \{a, b, c\}$ or $S = \{x \mid x \text{ has property } P\} = \{x \mid P(x)\}$
  - (Abstraction) Every property determines a set.
  - (Choice) Given any set $\mathcal{F}$ of nonempty pairwise disjoint sets, there is a set that contains exactly one member of each set in $\mathcal{F}$.

- **New, defined notions**
  - *relations between sets*: = (equality of sets), ⊆ (subset of a set)
  - *operations on sets*: complement, ∪ (union), ∩ (intersection), - (difference), and 2* (power set of a set)
  - and using the above:
    - ordered pair, Cartesian product, function (injective, surjective, bijective)
    - natural number
    - cardinal number (describing the size of a set)
    - ordinal number (describing the order in a set)
- … **but also surprising theorems**
  - e.g., *There are larger and larger infinite sets whose sizes are larger and larger infinite cardinal numbers—and this never ends!*
- Cantor's set theory was useful but curious, wild world of infinities!

Paradoxes in Cantor's Set Theory

- **Logical paradoxes**
  - In Cantor's set theory, *paradoxical statements* were proved, e.g.,
    - there is an ordered set Ω whose ordinal number is *larger than itself* ! (Burali-Forti)
    - there is a set $\mathcal{U}$ whose cardinal number is *larger than itself* ! (Cantor himself)
    - there is a set $\mathcal{R}$ that *both is and is not* a member of itself! (Russell)
- **Why do we fear paradoxes?**
  - If, in a theory, a statement and its negation can be proved, then *any* statement of the theory can be proved. Such a theory has no cognitive value!

Schools of Recovery

- **Intuitionism**
  - Brouwer, Heyting, …
  - Demanded for mathematical rigour, e.g.,
    - infinite sets are potentialities, not actualities
    - to exist is the same as to be constructed
    - the Law of Excluded Middle is not fully accepted
- **Logicism**
  - Boole, Frege, Peano, Peirce, Russell, Whitehead
  - Developed symbolic language for concise/precise mathematical expression, e.g.,
    - algebraic logic, *Propositional Calculus* **P**
    - rules of construction, quantified variables, *First-order Logic* **L**
    - Principia Mathematica, a theory containing fundamental theorems of mathematics in symbolic language

… cont'd (Schools of Recovery)

- **Formalism**
  - Hilbert, Ackermann, Bernays, …
  - Focussed on the syntax of math. expressions instead of their semantics
    - defined *rules* to build well-formed objects from symbols
    - invented formal axiomatic systems to implement these ideas
  - A *formal axiomatic system* consists of
    - a symbolic language
    - + rules of construction (to build *formulas,* well-formed expressions)
    - + rules of inference (to build *formal proofs,* w.f. sequences of formulas)
    - Some formulas are *axioms*. A formal proof is a finite sequence of formulas each of which is inferred from axioms and/or previous formulas only. A *theorem* is a formula that can be formally proved.
- A (formal) **theory** = axioms + theorems + other formulas

… cont'd (Schools of Recovery)

… cont'd (Formalism)

- **Interpretation**
  - a theory must be given some meaning (*interpreted* in a field of interest)
  - this tells us whether the theory is *applicable*
- **Metatheory**
  - answers the questions *about* the formal theory
  - e.g., Is the formal theory *consistent*? Is it *complete*? Is it *decidable*?

- **Goals of formalism**
  - short term:
    - Is *Principia Mathematica* a consistent theory? Is it complete?
  - long term:
    - develop all mathematics in one formal axiomatic system and prove that such mathematics is free of all (un)known paradoxes

Ch 3. Formalism

Formal Axiomatic System

- = symbolic language + axioms + rules of inference
- **Symbolic language**
  - = alphabet + rules of construction
  - alphabet = individual-constant symbols $a,b,c,\ldots$
    + individual-variable symbols $x,y,x,\ldots$
    + function symbols $f,g,h,\ldots$
    + predicate symbols $P,Q,R,\ldots$
    + logical connectives $\wedge,\vee,\Rightarrow,\Leftrightarrow,\neg$
    + quantification symbols $\forall,\exists$
    + punctuation marks

    (+ function-variable symbols + predicate-variable symbols)
  - rules of construction: define *terms* and *formulas,* i.e., well-formed sequences of symbols

… cont'd (Formal Axiomatic System)

- **Axioms**
  - = selected formulas = logical axioms + proper axioms
  - logical axioms: epitomize the principles of pure logic reflection
  - proper axioms: condense other special basic notions and facts
- **Rules of inference**
  - specify conditions in which, given a set of *premises*, one may derive a *conclusion*
  - e.g., Modus Ponens: G, $G \Longrightarrow F \vdash F$ and Generalization: $F(x) \vdash \forall F(x)$
- **Development of the theory**
  - starts when the symbolic language, axioms, and rules of inference are fixed
  - each new notion must be defined by the basic and/or already defined notions
  - each new proposition must be derived (formally proved) to become a theorem
  - $\vdash_{\mathbf{F}} F$ … denotes that F is *derivable* (*formally provable*) in the theory (f.a.s) **F**
- **Benefits**
  - it is in principle easier to maintain and check the validity of formal proofs

Interpretations and Models

- **Interpretation of a Theory**
  - assigns a particular meaning to a (formal) theory in a particular *domain,* i.e., describes, for every closed formula of the theory, how the formula is to be understood as a statement about members, functions, and relations of the domain
  - an open formula (one with *free* individual-varible symbols) is
    - satisfiable if these symbols can be assigned values so that the resulting statement is true
    - valid under the (current) interpretation if any assignment of values to these symbols results in a true statement
- a theory may have several interpretations
- a formula is logically valid if it is valid under every interpretation
  - logical axioms are logically valid; **what about proper axioms?**

… cont'd (Interpretations and Models)

- **Model of a Theory**
  - a model of a theory = an interpretation of the theory under which also *all proper axioms are valid* (hence, *all* axioms are valid)
  - intuitively: a model of the theory is a field of our interest that the theory sensibly formalizes
  - a theory may have several models
  - a formula is valid in the theory if it is valid in every model of the theory
    - intuitively: such a formula represents a (mathematical) *Truth* expressible in the formal axiomatic system (theory)
    - $\models_{\mathbf{F}} F$ … denotes that F is valid in the theory (f.a.s.) **F**

Formalization of Logic, Arithmetic, and Set Theory

- **Formalization of Logic**
  - First-order logic **L** (= *First-order Predicate Calculus*)
  - **L** is a formal axiomatic system that
    - formalizes all the logical principles/tools needed to develop any formal theory in a logically unassailable way
    - has
      - symbolic language
        - individual-variable symbols $x,y,z,\ldots$
          + logical connectives $\wedge,\vee,\Rightarrow,\Leftrightarrow,\neg, \forall, \exists$
          + equality symbol $=$
          + punctuation marks
        - rules of construction of formulas
      - five axiom schemas (patterns for construction of logical axioms)
      - rules of inference Modus Ponens, Generalization

- **First-Order Formal Axiomatic Systems and Theories**
  - are extensions of **L** (i.e., contain **L** as a sub-theory)
  - in addition to **L** they have
    - *proper symbols* (specific to the domain of interest)
    - *proper axioms* (condense specific basic facts of the domain of interest)
  - important examples
    - *Formal Arithmetic* **A**
    - *Axiomatic set theories* **ZF(C)** and **NBG**

- **Formalization of Arithmetic**
  - Formal Arithmetic **A** (= *Peano Arithmetic*)
  - **A** has
    - proper symbols
      - individual-constant symbol: 0
      - function symbols: $'$, $\oplus$, $\odot$
    - nine proper axioms, e.g.,
      - $\forall x(0 \neq x')$
      - $\forall x(x \oplus 0 = x)$
      - $F(0) \wedge \forall x\big(F(x) \Rightarrow F(x')\big) \Rightarrow \forall x F(x)$, for any formula F with free x *(Axiom of Mathematical Induction)*

- **Formalization of Set Theory**
  - Axiomatic set theory **ZFC** (*Zermelo-Fraenkel axiomatic set theory*)
  - Axiomatic set theory **NBG** (*von Neumann-Bernays-Gödel axiomatic set theory*)
  - **ZFC** has
    - proper symbols
      - individual-constant symbol: Ø
      - predicate symbol: ∈ (binary relation)
    - nine proper axioms (with *Axiom of Choice*)
- **NBG**
  - introduces the basic notion of *class*
  - **NBG** is a conservative extension of **ZF** (whatever can be proved in **ZF** can also be proved in **NBG**; the opposite holds for formulas that are also formulas in **ZF**.)

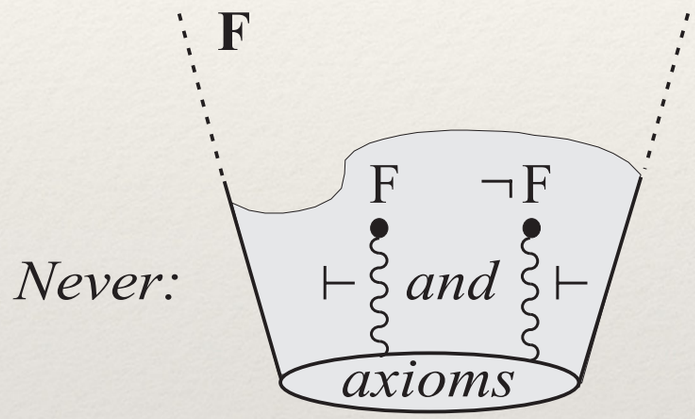
Ch 4. Hilbert's Attempt at Recovery

Hilbert's Program

- = a promising formalistic attempt to recover mathematics
- David Hilbert
- **Main ideas**
  - use *formal axiomatic systems* to put mathematics on a sound footing
  - to achieve that
    - define certain *fundamental problems* about f.a.s. and their theories
    - construct a f.a.s. **M** that will *formalize all mathematics*
    - solve (positively) the fundamental problems for the case of **M**

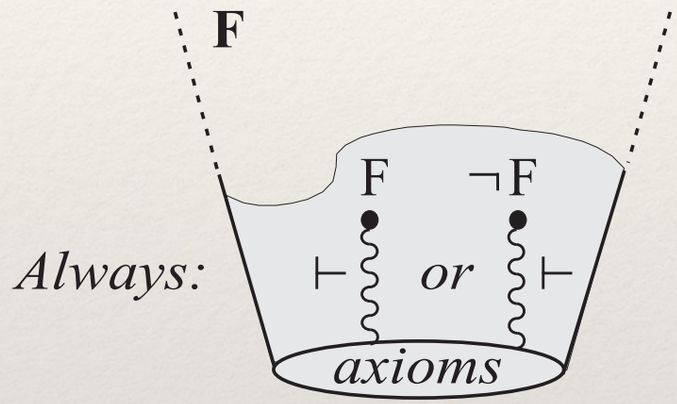
Fundamental Problems of the Foundations of Math

- **Consistency Problem**
  - **Definition.** A theory **F** is consistent if for no closed formula $F \in \mathbf{F}$ both F and ¬F are derivable in **F**.



- in an *in*consistent theory *any* formula of the theory is derivable (such a theory has no cognitive value)
- **Definition.** (consistency problem) Is a theory **F** consistent?

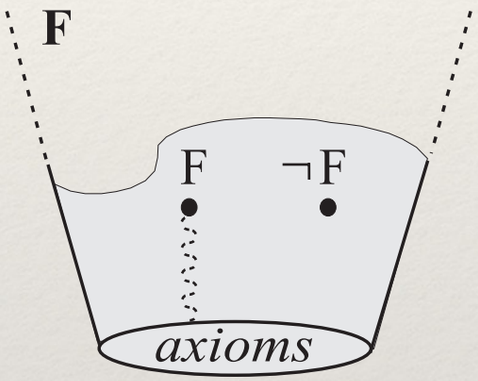
- **Syntactic Completeness Problem**
  - **Definition.** A consistent theory **F** is syntactically complete if, for every closed formula F ∈ **F**, either F or ¬F is derivable in **F**.



  - in a syntactically complete **F**, every closed formula is either provable or refutable (no formula is independent of **F**)
  - **Definition.** (synt.compl.prob.) Is a theory **F** syntactically complete?

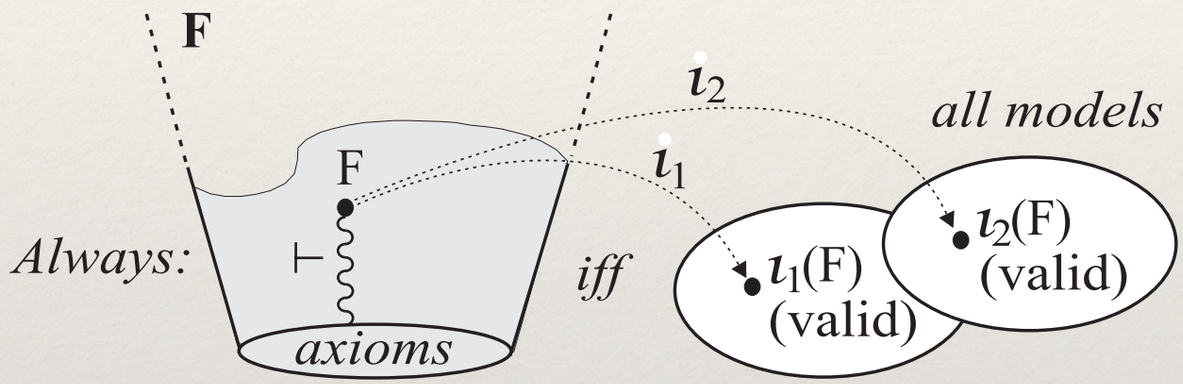
- **Decidability Problem**
  - **Definition.** A consistent and syntactically complete theory **F** is decidable if there is a decision procedure (algorithm) capable of answering, for any formula F ∈ **F**, the question "Is F derivable in **F**?"

Decision procedure for
"Is F *derivable in* **F***?"*



- a decidable **F** allows for a systematic and effective search for formal proofs (without investing our ingenuity and creativity)
- **Definition.** (decidability problem) Is a theory **F** decidable?

- **Semantic Completeness Problem**
  - **Definition.** A consistent theory **F** is semantically complete if, for every formula F ∈ **F**, F is derivable in **F** *iff* F is valid in **F**.



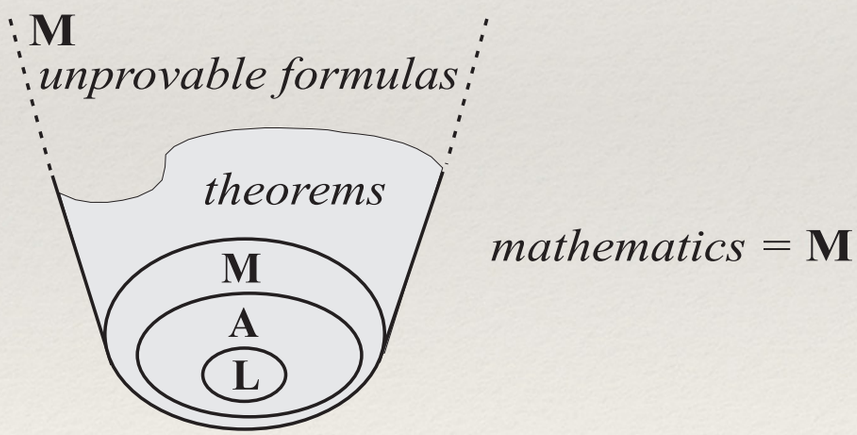
  - in a semantically complete **F**, a formula is derivable *iff* the formula is valid in every model of **F** (F represents a *Truth*)
  - **Definition.** (sem.compl.prob.) Is a theory **F** semantically complete?

Hilbert's Program

- **Program**
  - A. Find a f.a.s. **M** capable of deriving all theorems of mathematics.
  - B. Prove that the theory **M** is semantically complete.
  - C. Prove that the theory **M** is consistent.
  - D. Construct an algorithm that is a decision procedure for the theory **M.**
- Having attained A,B,C,D, every mathematical statement would be mechanically verifiable. Why? Let be given an arbitrary mathematical statement. Then:
  - 1. write the statement as a formula F ∈ **M**
  - 2. since **M** is semantically complete (B.), F is valid in **M** *iff* F is derivable in **M**
  - 3. since **M** is consistent (C.), F and ¬F are not both derivable
  - 4. apply the decision procedure (D.) to decide which of F and ¬F is derivable
  - *Conclusion: if F is derivable, the statement is a Truth in math; otherwise, it is not*
  - **Note**. Hilbert expected that **M** would be syntactically complete!

The Fate of Hilbert's Program

- **Formalization of Mathemtics: f.a.s. M**
  - **M** should *inevitably* contain:
    - *First-order Logic* **L** (for the logically unassailable development of **M**)
    - *Formal Arithmetic* **A** (to bring natural numbers to **M**)
  - **M** would *probably* also contain one of the axiomatic systems **ZFC** or **NBG**
  - **M** would *perhaps* contain other f.a.s. that would formalize other fields of math



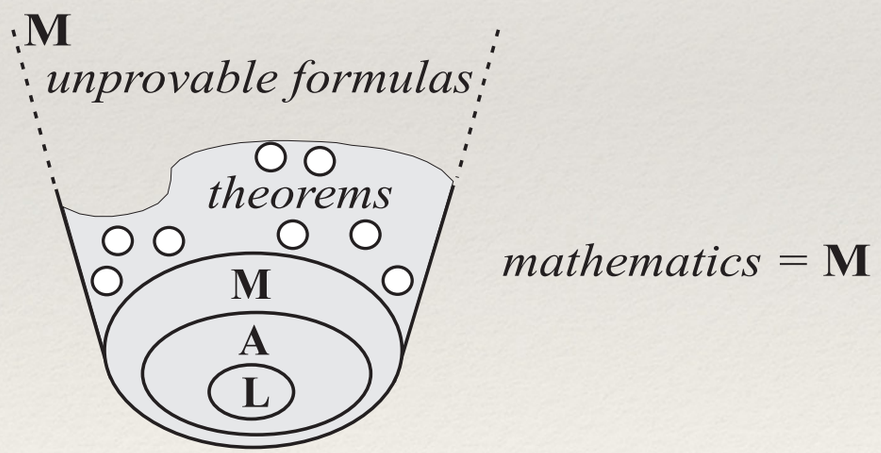
- **Decidability of M: Entscheidungsproblem**
  - The goal D of Hilbert's program is called Entscheidungsproblem. It asks: Construct a decision procedure (algorithm) that will, for any F ∈ **M**, decide whether or not F is derivable in **M** ($\vdash_{\mathbf{M}}$F).
  - intuitively, the decision procedure would be:

    systematically generate finite sequences of symbols of **M**, and
    for each newly generated sequence
      check whether the sequence is a proof of F in **M**;
      if so, **then** answer YES and halt
      else check whether the sequence is a proof of ¬F in **M**;
        if so, **then** answer NO and halt.

  - Note: assuming that either F or ¬F is provable in **M,** the algorithm always halts
    Hence: *if* **M** is consistent *and* **M** is syntactically complete
    then there is a decision procedure for **M** (**M** is decidable)

… cont'd (The Fate of Hilbert's Program)

- **Completeness of M: Gödels First Incompleteness Theorem**
  - **Theorem**(Gödel). *If the Formal Arithmetic* **A** *is consistent, then* **A** *is semantically incomplete.*
  - Consequences*: If* **M** is consistent, then **M** is semantically incomplete.
    - That is: there are formulas in **M** that represent *Truths* yet are not derivable in **M**
    - That is: Mathematics developed in **M** is like a "Swiss cheese full of holes" with some *Truths* dwelling in the holes, inaccessible to usual mathematical reasoning (= logical deduction in **M**)



- **Consistency of M: Gödels Second Incompleteness Theorem**
  - **Theorem**(Gödel). *If the Formal Arithmetic* **A** *is consistent, then this cannot be proved in* **A**.
  - Consequences: Proving the consistency of **A** would require means that are more complex (and less transparent) than those available in **A.**
    - E.g., Gentzen (1936) proved that **A** is consistent by using *transfinite induction*
  - So, we believe that **A** is consistent**.**
    - But this does *not* imply that **M** would be consistent!
    - Why? There is a generalization of Gödel's theorem: *If a consistent theory* **F** *contains* **A**, *then the consistency of* **F** *cannot be proved within* **F**. (Take **F** := **M**.)

- **Legacy of Hilbert's Program**
  - The mechanical, syntax-directed development of mathematics within the framework of formal axiomatic systems may be safe from paradoxes, *but* such mathematics suffers from semantic incompleteness and the lack of a possibility of proving its consistency.
  - Thus, Hilbert's program *failed*.

However:

- The problem of finding an algorithm that is a decision procedure for a given theory remained topical.
- Since there was a possibility of non-existence of such an algorithm, a formalization of the concept of the algorithm became necessary.

Part II. CLASSICAL COMPUTABILITY THEORY

- Ch 5 The quest for a formalization
- Ch 6 The Turing machine
- Ch 7 The first basic results
- Ch 8 Incomputable problems
- Ch 9 Methods of proving the in computability

Ch 5. The Quest for a Formalization

What is an Algorithm? What Do We Mean by Computation?

- Is there some *other algorithmic way* of recognizing every mathematical *Truth*?
- But, what is algorithm, anyway?
  - **Definition.** An algorithm (intuitively) for solving a problem is a finite set of instructions that lead the processor, in a finite number of steps, from the input data of the problem to the corresponding solution.
  - **Questions.** What instructions should be *basic* (i.e., allowed)? Would they suffice to compose *any* algorithm? Would they execute in a discrete or continuous way? Would their results be predictable (deterministic) or not? Could the processor execute any basic instruction? Where would be kept the algorithm, input data, intermediate and final results? …

Models of Computation

- **Definition.** A definition that formally describes and characterises the basic notions of algorithmic computation (i.e., the algorithm and its environment) is called the model of computation.
- What could a model of computation take as an example?
  - Modelling after functions
    - *Recursive functions*
    - *General recursive functions*
    - *Lambda-calculus*
  - Modelling after humans
    - *Turing machine*
  - Modelling after languages
    - *Post machine*
    - *Markov algorithms*

- **Recursive Functions** (Gödel (1931), Kleene (1936))
  - Given are the following three initial functions:
    - zero function: $\zeta(n) = 0$, for every $n \in \mathrm{N}$;
    - successor function: $\sigma(n) = n+1$, for every $n \in \mathrm{N}$;
    - projection function: $\pi_i^k(\boldsymbol{n}) = n_i$, where $\boldsymbol{n}$ denotes the sequence $n_1,\ldots,n_k$ and $1 \le i \le k$.
  - Given are the following three *rules of construction*:
    - a function $f : \mathrm{N}^k \longrightarrow \mathrm{N}$ is said to be constructed by composition (from functions g and h_is) if $f(\boldsymbol{n}) = g(h_1(\boldsymbol{n}),\ldots,h_m(\boldsymbol{n}))$, where $g : \mathrm{N}^m \longrightarrow \mathrm{N}$ and $h_i : \mathrm{N}^k \longrightarrow \mathrm{N}$ for $i = 1,\ldots,m$;
    - a function $f : \mathrm{N}^{k+1} \longrightarrow \mathrm{N}$ is said to be constructed by primitive recursion (from functions g and h) if $f(\boldsymbol{n},0) = g(\boldsymbol{n})$ and $f(\boldsymbol{n}, m+1) = h(\boldsymbol{n}, m, f(\boldsymbol{n}, m))$, for $m \ge 0$, where $g : \mathrm{N}^k \longrightarrow \mathrm{N}$ and $h : \mathrm{N}^{k+2} \longrightarrow \mathrm{N}$;
    - a function $f : \mathrm{N}^k \longrightarrow \mathrm{N}$ is said to be constructed by μ-operation (from the function g) if $f(\boldsymbol{n}) = \mu x\, g(\boldsymbol{n}, x)$, where $\mu x\, g(\boldsymbol{n}, x)$ is the least $x \in \mathrm{N}$ such that $g(\boldsymbol{n}, x) = 0$ and $g(\boldsymbol{n}, z)\!\downarrow$ for $z = 0,\ldots,x\text{-}1$.
  - The construction of a function f is a finite sequence $f_1,\ldots, f_k$, where $f_k = f$ and each f_i is either one of the initial functions or is constructed by one of the rules of construction from its predecessors in the sequence.
  - **Definition.** A function is recursive if it can be constructed as described above.

- **Model of computation (Gödel-Kleene)**
  - an "algorithm" is a construction of a recursive function
  - a "computation" is a calculation of a value of a recursive function that proceeds according to the construction of the function
  - a "computable" function is a recursive function

… cont'd (Models of Computation)

- **General Recursive Functions** (Herbrand(1931), Gödel (1934))
  - Let f denote an *unknown* function and let g_1 ,…, g_k be known numerical functions.
  - Let $E(f)$ denote a *system of equations* (with f and gs) which
    - is in *standard* form, i.e.,
      - f is only allowed on the left-hand side of the equations, and
      - f must appear as $f(g_i(\ldots), \ldots, g_j(\ldots)) = \ldots$
    - guarantees that f is a *well-defined* function.
  - There are two *rules for manipulating* $E(f)$ to calculate the value of f :
    - in an equation, all occurrences of a variable can be *substituted* by the same number
    - in an equation, an occurrence of a function can be *replaced* by its value
  - The system $E(f)$ *defines* the function f.
  - **Definition.** A function is *general recursive* if there is a system that defines it.

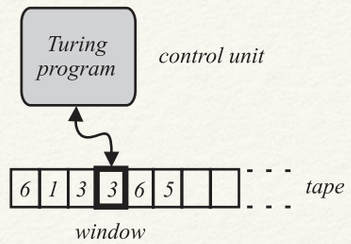
- **Model of computation (Herbrand-Gödel-Kleene)**
  - an "algorithm" is a system of equations E(*f*) for some *f*
  - a "computation" is a calculation of a value of a general recursive function *f* that proceeds according to E(*f*) and the two rules
  - a "computable" function is a general recursive function

- **Lambda-Calculus** (Church (1931-34))
  - Let $f, g, x, y, z, \ldots$ denote variables.
  - A λ-term is a well-formed expression defined inductively as follows:
    - a variable is a λ-term (called atom)
    - if M is a λ-term and x a variable, then $(\lambda x.M)$ is a λ-term (built from M by abstraction)
    - if M and N are λ-terms, then (MN) is a λ-term (called the application of M on N)
  - λ-terms can be transformed into other λ-terms. A transformation is a series of one-step transformations called β-reductions. There are two rules to do a β-reduction:
    - α-conversion renames a bound variable in a λ-term i
    - β-contraction transforms a λ-term $(\lambda x.M)N$ —called β-redex— into a λ-term obtained from M by substituting N for every bound occurrence of x in M. (We say that M is *applied* on N.)
- When a λ-term contains no β-redexes, it cannot further be β-reduced; such a λ-term is said to be in β-normal *form*. (Intuitively, a λ-term is in β-normal form if it contains no functions to apply.)
- **Definition.** A function is λ-definable if it can be represented by a λ-term.

… cont'd (Models of Computation)

- **Model of computation (Church)**
  - an "algorithm" is a λ-term
  - a "computation" is a transformation of an initial λ-term into the final one
  - a "computable" function is a λ-definable function

Turing Machine (Turing (1936))



- The Turing machine (TM) consists of several components:
  - a control unit (≈ human brain);
  - a potentially infinite tape divided into equal sized cells (≈ paper used during human computation);
  - a window that can move over any cell and makes it accessible to the control unit (≈ human eye & hand with a pen)
- The control unit is always in some state. Two of the states are the initial and the final state. There is a *program* (called the Turing program, TP) in the control unit. Different TMs have different TPs. Before the TM is started, the following is done:
  - an input word is written on the tape (the into word is written in an alphabet Σ and contains input data);
  - the window is shifted to the beginning of the input word;
  - the control unit is set to the initial state.
- From now on, the TM operates independently, step by step, as directed by its TP. At each step, TM reads the symbol from the cell under the window into its control unit and, based on this symbol and the current state of the control unit:
  - writes a symbol into the cell under the window (while deleting the old symbol);
  - moves the window to one of the neighboring cells or leaves the window as it is;
  - changes the state of the control unit.
- The TM halts, if its control unit enters the final state or if its TP has no instruction for the next step.
- **Definition.** A function $f : \mathbb{N}^k \longrightarrow \mathbb{N}$ is Turing-computable if there is a TM T such that if the input word to T represents numbers $a_1, \ldots, a_k$, then, after halting, the tape contents represent the number $f(a_1, \ldots, a_k)$.

- **Model of computation (Turing)**
  - an "algorithm" is a Turing program
  - a "computation" is an execution of a Turing program on a Turing machine
  - a "computable" function is a Turing-computable function

… cont'd (Models of Computation)

- **Post machine** (Post (1920s))
  - The Post machine (PM) consists of several components:
    - a control unit;
    - a potentially infinite *read-only tape* divided into equal sized cells;
    - a window that can move over any cell and makes it accessible to the control unit;
    - a queue for symbols.
  - The control unit is always in some state. Some of the states are the initial, accept and reject state. There is a *program* (called the Post program, PP) in the control unit. Different PMs have different PPs. Before the PM is started, the following is done:
    - an input word is written on the tape (the into word is written in an alphabet Σ and contains input data);
    - the window is shifted to the beginning of the input word;
    - the control unit is set to the initial state.
  - From now on, the PM operates independently as directed by PP. At each step, PM reads the symbol from the tape and consumes the symbol from the head of the queue; then, based on the two symbols and the current state:
    - adds a symbol to the end of the queue;
    - moves the window to a neighboring cell;
    - changes the state of the control unit.
  - The PM halts if the word in the queue is *accepted* or *rejected* or if its PP has no instruction for the next step.
  - **Definition.** A function $f : \mathrm{N}^k \longrightarrow \mathrm{N}$ is Post-computable if there is a PM P such that if the input word to P represents numbers $a_1, \ldots, a_k$, then, after halting, the queue contents represent the number $f(a_1, \ldots, a_k)$.

… cont'd (Models of Computation)

- **Model of computation (Post)**
  - an "algorithm" is a Post program
  - a "computation" is an execution of a Post program on a Post machine
  - a "computable" function is a Post-computable function

- **Markov Algorithms** (Markov (1951))
  - A Markov algorithm (MA) is a finite sequence M of productions

$$\alpha_1 \to \beta_1$$
$$\alpha_1 \to \beta_1$$
$$\dots$$
$$\alpha_n \to \beta_n$$

where α_i ,β_i are words over an alphabet Σ. The sequence M is also called the *grammar*.

  - A production $\alpha_i \to \beta_i$ is applicable to a word w if α_i is a subword of w. If $\alpha_i \to \beta_i$ is applied to w, it replaces the leftmost occurrence of α_i in w with β_i.
  - An execution of a Markov algorithm M is a sequence of steps that transform a given input word via a sequence of intermediate words into some output word. At each step, the last intermediate word is transformed by the first applicable production. Some productions are said to be final.
  - The execution halts if the last applied production was final or there was no production to apply. Then, the last intermediate word is the output word.
  - **Definition.** A function $f : \mathbb{N}^k \to \mathbb{N}$ is Markov-computable if there is a MA M such that if the input word represents numbers $a_1, \dots, a_k$, then, after halting, the output word represents $f(a_1, \dots, a_k)$.

… cont'd (Models of Computation)

- **Model of computation (Markov)**
  - an "algorithm" is a Markov algorithm (grammar)
  - a "computation" is an execution of a Markov algorithm
  - a "computable" function is a Markov-computable function

Computability (Church-Turing) Thesis

- Which model (if any) is the right one, i.e., which appropriately formalises (↔) the intuitive concepts of the "algorithm," "computation," and "computable" function?

- Speculations:

  - Church thesis (1936): λ-calculus is the right one.

  - Turing thesis (1936) : Turing machine is the right one

- But we cannot <u>prove</u> that (a *vague* concept A) ↔ (a *rigorous* concept B) !!

- Luckily, the (rigorously defined) models of computation were proved to be *equivalent* in the sense that *what can be computed by one of them can also be computed by any other.*

- This strengthened the belief in the following Computability Thesis:

  Basic intuitive concepts of computing are appropriately formalised as follows:

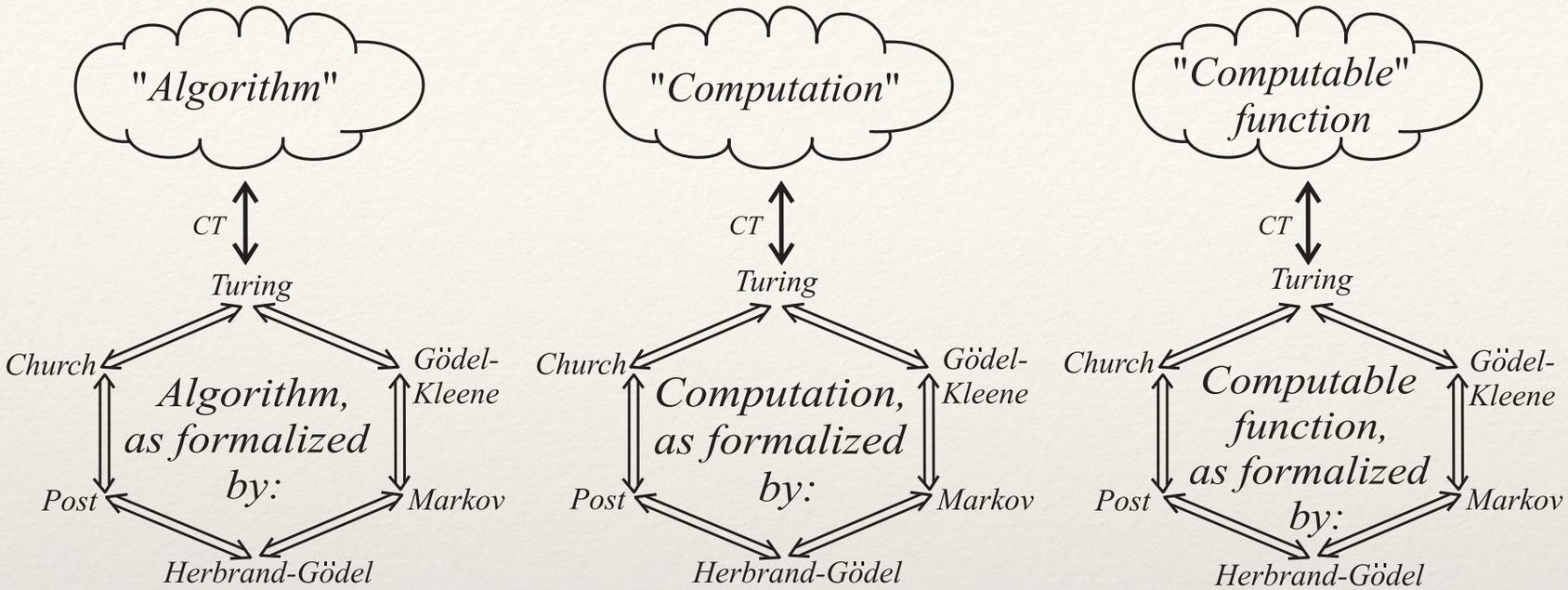
  "algorithm" ↔ *Turing program*

  "computation" ↔ *execution of a Turing program on a Turing machine*

  "computable" function ↔ *Turing-computable function*

  Instead of the TM we can use any other equivalent model.

… cont'd (Computability Thesis)



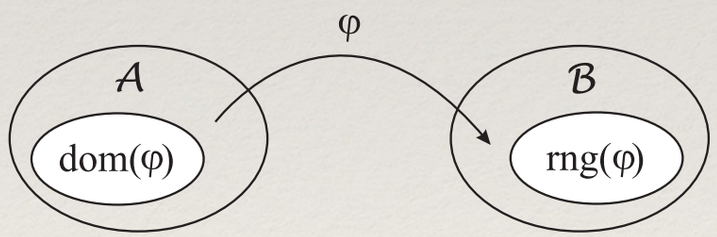
- The Computability thesis established a bridge between the *intuitive concepts* of "algorithm," "computation", and "computable" function on the one hand, and their *formal counterparts* defined by models of computation on the other.
- In this way, it opened the door to a mathematical treatment of these intuitive concepts.
- Until now, the thesis was not refuted; most researchers believe that the thesis holds.

… cont'd (Computability Thesis)

- The concepts "algorithm" and "computation" are now formalized. We no longer use quotation marks to distinguish between their intuitive and formal meanings.
- But, with the concept of "computable" function we we must first clarify *which functions we must talk about*.
- Why?
  - Recursive functions are computable (by Computability thesis).
  - There are *countably* many recursive functions.
  - There are *uncountably* many numerical functions.
  - So there must be *many numerical* functions that are *not recursive*.
  - Can we find a numerical function that is not recursive but still "computable" ?
  - Yes! We do this by a method called diagonalization.
- So, there are "computable" functions which are not recursive !!!
- Does this refute the Computability thesis?
  - No, if we do *not* consider *only total* functions. (The value of a total function is always defined.)
- So, *we must also talk about partial functions*. (The value of a partial function can be undefined.)
  - Actually, the μ-operation already allows for the construction of partial recursive functions.

… cont'd (Computability Thesis)

- **Definition.** We say that $\varphi:\mathcal{A}\to\mathcal{B}$ is a partial function if φ may be undefined for some elements of $\mathcal{A}$.
- We write $\varphi(a)\downarrow$ if φ is defined for a; otherwise we write $\varphi(a)\uparrow$.
- The domain of φ is the set $\mathrm{dom}(\varphi) = \{a\in\mathcal{A};\ \varphi(a)\downarrow\}$.
- We have $\mathrm{dom}(\varphi)\subseteq\mathcal{A}$. When $\mathrm{dom}(\varphi)=\mathcal{A}$, we say that φ is a total function (or just a function).
- We write $\varphi(a)\downarrow = b$ if φ is defined for a and its value is b.
- *The* range of φ is the set $\mathrm{rng}(\varphi) = \{b\in\mathcal{B};\ \exists a\in\mathcal{A} : \varphi(a)\downarrow = b\}$.
- The function is surjective if $\mathrm{rng}(\varphi)=\mathcal{B}$, and it is injective, if different elements of $\mathrm{dom}(\varphi)$ are mapped into different elements of $\mathrm{rng}(\varphi)$.
- Partial functions $\varphi : \mathcal{A}\to\mathcal{B}$ and $\psi : \mathcal{A}\to\mathcal{B}$ are equal, denoted by $\varphi\simeq\psi$, if they have the same domains and the same values (for every $x\in\mathcal{A}$ it holds that $\varphi(x)\downarrow \Leftrightarrow \psi(x)\downarrow$ and $\varphi(x)\downarrow \Rightarrow \varphi(x)=\psi(x)$).



… cont'd (Computability Thesis)

- We can now give the formalization of the concept of "computable" function.

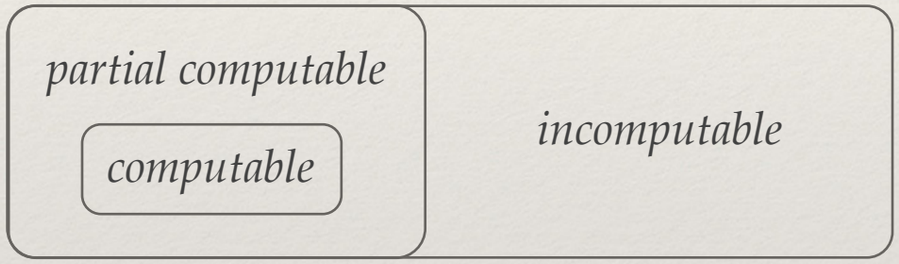
- In essence, it says that *a partial function is "computable" if there is an algorithm which can compute its value whenever the function is defined*.

The intuitive concept of "computable" partial function $\varphi : \mathcal{A} \to \mathcal{B}$ *is formalized as follows:*

φ *is* "computable" $\leftrightarrow$ *there exists a TM that can compute the value* $\varphi(x)$ *for any* $x \in \text{dom}(\varphi)$ and $\text{dom}(\varphi)=\mathcal{A}$

φ *is* partial "computable" $\leftrightarrow$ *there exists a TM that can compute the value* $\varphi(x)$ *for any* $x \in \text{dom}(\varphi)$

φ *is* "incomputable" $\leftrightarrow$ *there is no TM that can compute the value* $\varphi(x)$ *for any* $x \in \text{dom}(\varphi)$



Informally:

If $\varphi : \mathcal{A} \to \mathcal{B}$ is *partial computable*, the computation of $\varphi(x)$ *halts* for $x \in dom(\varphi)$ and *does not halt* for $x \in \mathcal{A} - dom(\varphi)$.

In particular, if $\varphi : \mathcal{A} \to \mathcal{B}$ is *computable*, the computation of $\varphi(x)$ *halts* for $x \in \mathcal{A}$.

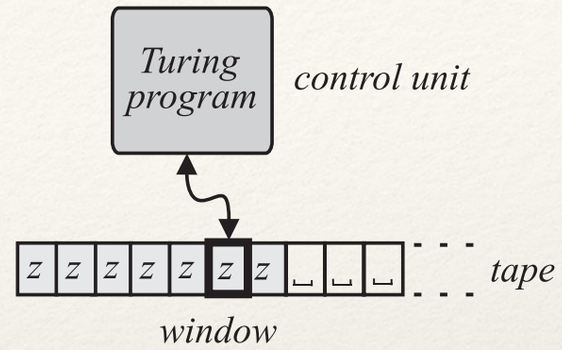
If $\varphi : \mathcal{A} \to \mathcal{B}$ is *incomputable*, the computation of $\varphi(x)$ does not halt for $x \in \mathcal{A} - dom(\varphi)$ and for some $x \in dom(\varphi)$.

Ch 6. The Turing Machine

- The Turing machine (TM) is a model of computation that convincingly formalized intuitive concepts of algorithm, computation, and computable function. Most researchers accepted it as the most appropriate model of computation. We will build on the TM.
- There is a *basic variant* of the TM and *generalized variants*.

Basic Model

- **Definition.** The *basic* variant of the Turing machine has:
  - a control unit containing a Turing program;
  - a tape consisting of cells;
  - a movable window which is connected to the control unit.



- The tape:
  - for writing and reading the input data, intermediate data, and output data (results);
  - potentially infinite in one direction;
  - in each cell there is a tape symbol belonging to a finite tape alphabet Γ. The symbol ␣ (empty space, blank) indicates that the cell is empty. There are at least two more symbols in Γ: 0 and 1.
  - the input data is contained in the input word, which is a word over some finite input alphabet Σ (such that $\{0,1\} \subseteq \Sigma \subseteq \Gamma - \{␣\}$). The input word is written in the leftmost cells, all the other cells are empty.

- The control unit:
  - always in some state belonging to a finite set of states Q. The initial state *is* q_1. Some states are *final;* they are in the set of final states $F \subseteq Q$.
  - contains a program called the Turing program TP.
- The Turing program
  - directs the whole TM;
  - characteristic of a particular TM;
  - a partial function $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{\text{Left, Right, Stay}\}$ called the transition function.
- The window:
  - can only move to the neighbouring cell (Left or Right) or stays where it is (Stay);
  - the control unit can read a symbol from the current cell and write a symbol to the cell.

… cont'd (Basic Model)

- Before the TM is started:
  - an input word is written to the beginning of the tape;
  - the window is shifted to the beginning of the tape;
  - the control unit is set to the initial state.
- Then the TM operates independently, in a mechanical stepwise fashion as instructed by δ. Specifically, if the TM is in a state q_i and it reads a symbol z_r , then:
  - *if* q_i is a final state, then the TM *halts*;
  - *else, if* $\delta(q_i, z_r)\uparrow$, then TM *halts*;
  - *else, if* $\delta(q_i, z_r)\downarrow = (q_j , z_w , D)$, then the TM does the following:
    - changes the state to q_j ;
    - writes z_w through the window;
    - moves the window to the next cell in direction $D \in$ {Left, Right} or leaves the window where it is (D = Stay).

… cont'd (Basic Model)

- Formally, a TM is a seven-tuple $T = (Q, \Sigma, \Gamma, \delta, q_1, \sqcup, F)$. To fix a particular TM, we fix Q,Σ,Γ,δ,F.
- The computation:
  - **Definition.** Let us start a TM T on an input word w. The internal configuration of T after a finite number of computational steps is the word uq_iv, where
    - q_i is the current state of T;
    - uv are the current contents of the tape (up to (a) the rightmost non-blank symbol or (b) the symbol to the left of the window, whichever of a,b is rightmost);
    - T is scanning leftmost symbol of v (in case a) and $\sqcup$ (in case b).
  - The initial configuration is q_1w.
  - Given an internal configuration, the next internal configuration can easily be constructed using the transition function δ.
  - The computation of T on w is represented by a sequence of internal configurations starting with the initial configuration.
  - Just as the computation may not halt, the sequence may also be infinite.

Generalized Models

- There are several generalizations of the basic model. Each extends the basic model in some respect:
  - Finite storage TM: The control unit can memorize several tape symbols and use them during computation.
  - Multi-track TM: The tape is divided into several tracks, each containing its own contents.
  - Two-way unbounded TM: The tape is potentially infinite in both directions.
  - Multi-tape TM: There are several tapes each having its own window that is independent of other windows.
  - Multidimensional TM: The tape is multi-dimensional.
  - Nondeterministic TM: The transition function offers alternative transitions and the machine always chooses the "right" one.
- Although each of the generalizations seem to be more powerful than the basic model, it is not so.

Each of the generalisations is equivalent to the basic model.
This is because each of them can be simulated by the basic model.

Reduced Models

- There are also simplifications of the basic model. Each fixes the basic model in some respect. By fixing everything to the simplest possibility we obtain:
  - Reduced model: The parameters Σ, Γ, F in the formal definition of the Turing machine $T = (Q, \Sigma, \Gamma, \delta, q_1, \sqcup, F)$ are fixed as follows: $\Sigma := \{0,1\}$; $\Gamma := \{0,1,\sqcup\}$; $F := \{q_2\}$. So, the reduced TMs are $T = (Q, \{0,1\}, \{0,1,\sqcup\}, \delta, q_1, \sqcup, \{q_2\})$. Since Q can be determined from δ, the reduced TMs can be specified by their δs only.
- Although the reduced model seems to be less powerful than the basic one, it is not so.

The reduced model is equivalent to the basic model.
This is because the basic model can be be simulated by the reduced model.

Universal Turing Machine

- If each TM were described by a characteristic natural number (index), then each TM could compute with other TMs by including their indexes into its input word. Such coding would also enable self-reference of TMs.
- **Coding and enumeration of TMs**:
  - Let $T = (Q, \Sigma, \Gamma, \delta, q_1, \sqcup, F)$ be an arbitrary TM and $\delta(q_i, z_j) = (q_k, z_\ell, D_m)$ an instruction of its TP. We encode the instruction by the word $K = 0^i10^j10^k10^\ell10^m$, where D_1 = Left, D_2 = Right, and D_3 = Stay.
  - In this way, we encode each instruction of the program δ.
  - From the codes $K_1, K_2, \ldots, K_r$ we construct the code of T: $\langle T \rangle = 111K_111K_211\ldots11K_r111$.
  - We interpret $\langle T \rangle$ to be the binary code of some natural number. We call this number the index of T.
  - **Convention**: Any natural number whose binary code is not of the above form is an index of the empty TM (TP of the empty TM is everywhere undefined.).
- *Every natural number is the index of exactly one Turing machine; we can speak of i-th TM, T_i.*

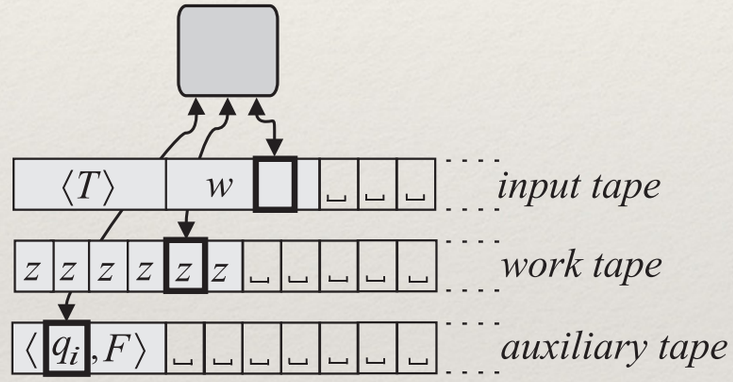
- **The Existence of a Universal Turing Machine**

  There is a TM that can compute whatever is computable by any other TM.

- **How?**

  - *The idea*: contract a TM U that is capable of simulating *any* other TM T.

  - *The concept of U:* Let $T = (Q, \Sigma, \Gamma, \delta, q_1, \sqcup, F)$ be an arbitrary TM and w input to T. Then:



The input tape contains $\langle T \rangle$ and w. The work tape is used by U in exactly the same way as T would use its own tape when given the input w. The auxiliary tape is used by U to record the current state in which the simulated T would be at that time. Instructions of T are extracted from $\langle T \rangle$.

… cont'd (Universal Turing Machine)

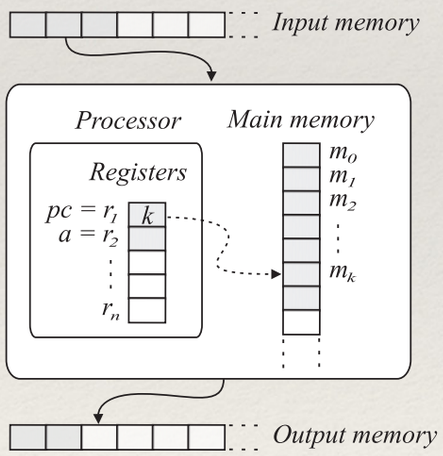
- **Practical Consequences**
  - Data vs. instructions

    There is no a priory difference between data and instructions;
    the distinction between the two is established by their interpretation.

  - General-purpose computer

    It is possible to construct a physical computing machine that can
    compute whatever is computable by any other physical computing machine.

- Von Neumann's architecture and the RAM model of computation



The RAM and the TM are equivalent;
what can be computed on one of them
can be computed on the other.

Use of a Turing Machine

- There are three elementary tasks for which we can use Turing machine:
  - **Function computation**: given a function φ and $u_1,\ldots,u_k$, compute $\varphi(u_1,\ldots,u_k)$
  - **Set generation**: given a set S, list all of its elements
  - **Set recognition**: given a set S and an x, answer the question $x \in ?\ S$

Function Computation

A TM is implicitly associated, for each $k \geq 1$, with a k-ary function, called a *proper function*.

- Let $T = (Q, \Sigma, \Gamma, \delta, q_1, \sqcup, F)$ be an arbitrary TM and $u_1,\ldots,u_k$ words written in Σ. Write $u_1,\ldots,u_k$ to the tape, start T, and wait until T halts and leaves a single word in Σ on the tape. *If this happens*, and the resulting word is denoted by v, then we say that T has computed the value v of its k-ary proper function, $\psi_T^{(k)}$, for the arguments $u_1,\ldots,u_k$.

- If e is an index of T, we also denote the k-ary proper function of T by $\psi_e^{(k)}$. When k is known from the context, we write just ψ_T or ψ_e.

- The interpretation of the words $u_1,\ldots,u_k$ and v is left to us. For example, they can be (encodings of) natural numbers.

… cont'd (Use of a Turing Machine)

… cont'd (Function Computation)

Instead of constructing $\psi_T^{(k)}$ we often face the **opposite question**:

- Given a function $\varphi : (\Sigma^*)^k \to \Sigma^*$, find a TM $T = (Q, \Sigma, \Gamma, \delta, q_1, \sqcup, F)$ capable of computing φ's values, i.e., a T such that $\psi_T^{(k)} = \varphi$.

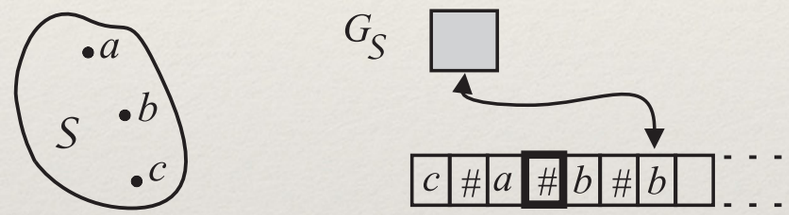
- Depending on how powerful, if at all, such a T can be, we distinguish between three kinds of φs.

- **Definition**. Let $\varphi : (\Sigma^*)^k \to \Sigma^*$ be a function. We say that
  - φ is computable *if* there is a TM that can compute φ anywhere on $\mathrm{dom}(\varphi)$ and $\mathrm{dom}(\varphi) = (\Sigma^*)^k$;
  - φ is partial computable *if* there is a TM that can compute φ anywhere on $\mathrm{dom}(\varphi)$;
  - φ is incomputable *if* there is *no* TM that can compute φ anywhere on $\mathrm{dom}(\varphi)$.

Set Generation

When can elements of a set S be "generated", i.e., listed in a sequence such that every element of S sooner or later appears in the sequence? When can the sequence be generated by an algorithm?

- A TM $T = (Q, \Sigma, \Gamma, \delta, q_1, \sqcup, F)$ that generates a set S writes to its tape, in succession, the elements of S and nothing else. The elements are delimited by the appropriate tape symbol in Γ-Σ , say #. Such a TM T is also denoted by G_S.



- **Post Thesis**.

  The intuitive concept of set generation is appropriately formalised as follows:

  a set S can be "generated" ⟷ S can be generated by a Turing machine

- **Definition.** A set S is computably enumerable (c.e.) if S can be generated by a TM.

- **Theorem**. A set S is c.e. ⇔ S = ∅ *or* S is the range of a computable function on N.

Set Recognition

A TM is implicitly associated with a set, called its *proper set*.

- Let $T = (Q, \Sigma, \Gamma, \delta, q_1, \sqcup, F)$ be an arbitrary TM and $w \in \Sigma^*$. Write w to the tape, start T, and wait until T halts. If T *halts* in a *final* state, we say that T accepts w.
  - If T halts on w in a *non*-final state, we say that it rejects w; if T never halts, we say that it does not recognize w.
- The proper set of T is the set of all the words that T accept; it is denoted by $L(T)$.

… cont'd (Use of a Turing Machine)

… cont'd (Set Recognition)

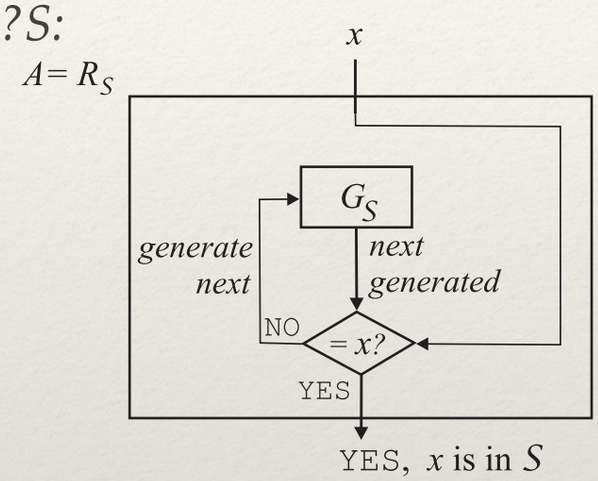
Instead of constructing L(T), we often face the **opposite question**:

- Given a set S, find a TM T such that $L(T) = S$.
- The existence of such a T is connected with S's amenability to *set recognition*. Informally, to *completely recognise* S in an environment (universe) U, is to determine which elements of U are members of S and which are not.
- We involve the notion of the characteristic function. The characteristic function of a set S, where $S \subseteq U$, is a function $\chi_S : U \to \{0,1\}$ defined by $\chi_S(x)=1$, if $x \in S$, and $\chi_S(x)=0$, if $x \notin S$. Note that $\chi_S : U \to \{0,1\}$ is total.
- We distinguish between three kinds of sets S, based on the extent to which the values of χ_S can possibly be computed on U.
- **Definition**. Let U be the universe and $S \subseteq U$ be an arbitrary set. We say that
  - S is decidable in U *if* χ_S is computable function on U;
  - S is semi-decidable in U *if* χ_S is computable function on S;
  - S is undecidable in U *if* χ_S is incomputable function on U.

Generation vs. Recognition

Theorem. *If a set S is c.e., then S is semi-decidable (in the universe U).*

- *Proof*. Let S be c.e. We use the generator G_S to construct an algorithm A for answering the question $x \in ?S$:



- Intuitively, G_S generates elements of S until x is generated (if at all).

… cont'd (Use of a Turing Machine)

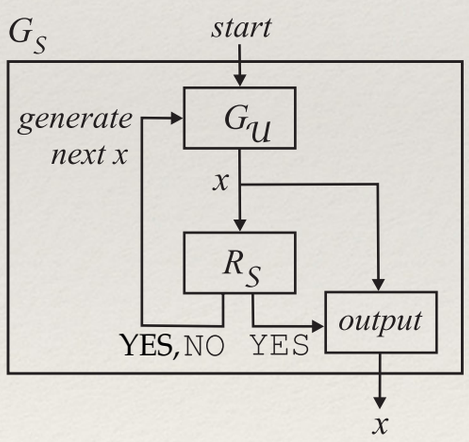
… cont'd (Generation vs. Recognition)

Theorem. *Let the universe U be c.e. If a set is semi-decidable, the S is c.e.*

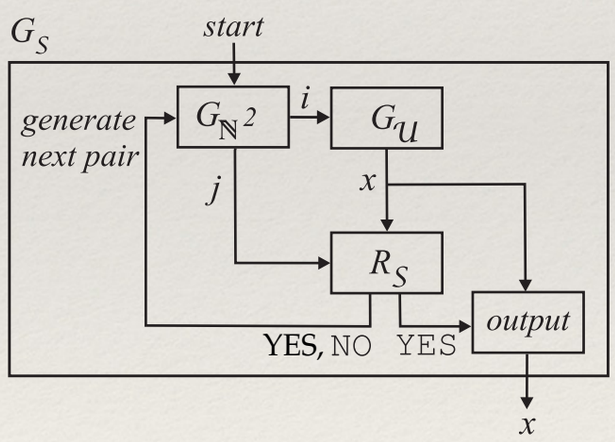
- *Proof (naive).* Let S be semi-decidable. G_S is on Fig. *a)*. (1) G_S asks G_U to generate the next element $x \in U$. (2) G_S asks R_S to answer $x \in$? S. (3) If the answer is YES, G_S outputs (generates) x. (4) G_S continues with (1). **BUT**: if $x \notin$ S, R_S *may run forever and never* return NO!

- *Proof (correct).* G_S is on Fig. *b)*. The trap is avoided by dovetailing. (1) G_S asks the *pair generator* G_{N^2} to generate the next pair $(i,j) \in N \times N$. (2) G_S asks G_U to generate i-th element of U, say x. (3) G_S asks R_S the question $x \in$? S. (4) If R_S answers YES *in exactly j-th step*, G_S generates (outputs) x. (4) G_S continues with (1).

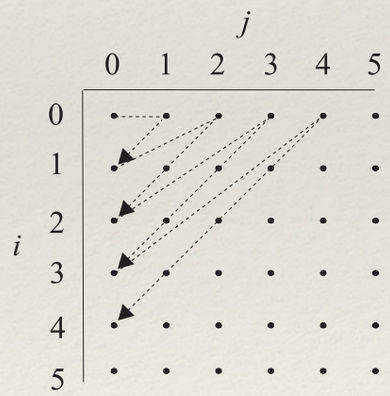
- The order of generated pairs (i,j) is on Fig. *c)*. Note that each pair is generated exactly once.



a)



b)



… cont'd (Use of a Turing Machine)

… cont'd (Generation vs. Recognition)

Recall: Σ^* is the set of all the words over the alphabet Σ, and N is the set of all natural numbers.

Theorem. *Σ^* and N are c.e. sets.*

Corollary. *Let $U = \Sigma^*$ or $U = N$. Then: A set is semi-decidable iff S is c.e.*

In what follows, we will have either $U = \Sigma^*$ *or* $U = N$. *Why? Is this ok?*

Theorem. *There is a bijection $f : \Sigma^* \to N$.*

So, *w*hen a property of sets is independent of the nature of their elements, we are allowed to choose whether to study the property using $U = \Sigma^*$ or $U = N$. The results will apply to the alternative, too. Three properties of this kind are especially interesting: the *decidability*, *semi-decidability*, and *undecidability* of sets. We will use the two alternatives according to the context and ease of using.

Ch 7. The First Basic Results

- Now, the basic notions and concepts are defined so we can start developing our theory. We will present:
  - several theorems about c.e. sets
    - the *Padding Lemma*
    - the *Parametrization (s-m-n) Theorem*
    - the *Recursion (Fixed Point) Theorem*
- Then we will present some practical consequences of the above theorems.

Some Basic Properties of Semi-Decidable (c.e.) Sets

- **Theorem.** *S is decidable $\Rightarrow$ S is semi-decidable.*
- **Theorem.** *S is decidable $\Rightarrow$ $\overline{S}$ is decidable.*
- **Theorem.** *S and $\overline{S}$ are semi-decidable $\Leftrightarrow$ S is decidable.*
- **Theorem.** *S is semi-decidable $\Leftrightarrow$ S is the domain of a computable function.*
- **Theorem.** *$\mathcal{A}$ and $\mathcal{B}$ are semi-decidable $\Rightarrow$ $\mathcal{A} \cup \mathcal{B}$ and $\mathcal{A} \cap \mathcal{B}$ are semi-decidable.*

  $\mathcal{A}$ and $\mathcal{B}$ are decidable $\Rightarrow$ $\mathcal{A} \cup \mathcal{B}$ and $\mathcal{A} \cap \mathcal{B}$ are decidable.

Padding Lemma

- *We already know:* Each natural number is the index of exactly one TM. What about the other way round? Is each TM represented by exactly one index? No!
- A TM has many indexes. Let T be a TM and $\langle T \rangle = 111K_1 11K_2 11 \ldots 11K_r 111$. We can
  - *permute* the subwords K_1 , K_2 ,…, K_r or
  - *insert* new subwords K_{r+1} , K_{r+2} ,… , where each of them represents a *redundant* instruction (that will never be executed).

  By such permuting and padding we can construct unlimited number of new codes. Each of them describes a different yet equivalent program (i.e., it executes in the same way as T's program). Hence, also a partial computable function has several indexes.

- **Lemma.** A *partial computable function has countably infinitely many indexes. Given one of them, the others can be generated.*
- **Definition.** The index set of a p.c. function φ is the set $\mathrm{ind}(\varphi) = \{x \in N \mid \psi_x \simeq \varphi\}$.

Parametrization (s-m-n) Theorem

- Let $\varphi_x(y,z)$ be a p.c. function. Fix the variable $y := p \in N$. (We call p the parameter.) Hence, we obtain a new p.c. function of *one* variable, $\psi(z) = \varphi_x(p,z)$. What is the index of ψ? The parametrization theorem states that ψ*'s* index *only* depends on x and p and it can be computed by a *computable* function.

- **Theorem.**(Parametrization) *There is injective computable function* $s : N^2 \to N$ *such that, for every* $x,p \in N$, *we have* $\varphi_x(p,z) = \psi_{s(x,p)}(z)$.

- The generalization to more variables and parameters is called the s-m-n theorem.

- **Theorem.**(*s-m-n*) For any $m,n \geqq 1$ *there is injective computable function* $s^m{}_n : N^{m+1} \to N$ *such that, for every* $x, p_1, \ldots, p_m \in N$, $\varphi_x(p_1, \ldots, p_m, z_1, \ldots, z_n) = \psi_{s(x, p1, \ldots, pm)}(z_1, \ldots, z_n)$.

- *Informally*: input parameters can be eliminated and, instead, integrated into the program.

Recursion (Fixed-Point) Theorem

- Let $f : N \to N$ be an arbitrary computable function. Recall: f is total.
- We can view f as a transformation that modifies *every* TM T_i into $T_{f(i)}$ by transforming T_i's program (encoded by i) into another Turing program (encoded by $f(i)$).
- In general, $f(i) \neq i$, so the two programs differ. What about their proper functions ψ_i and $\psi_{f(i)}$? This is where the recursion theorem (also called the fixed point theorem) enters.
- **Theorem**. (Recursion) *For every computable function f there is an $n \in N$ such that $\psi_n \simeq \psi_{f(n)}$. The number n can be computed from the index of the function f.*
- *Informally*: if f transforms every TM, then some TM (encoded by) n is transformed into an equivalent TM (encoded by) $f(n)$. In other words, if f modifies every TM, there is always some TM T_n for which the modified TM $T_{f(n)}$ computes the same function as T_n.
- Such an n is called the fixed point of the function f.
- **Theorem**. *A computable function has countably infinitely many fixed points.*

Practical Consequences: Recursive Program Definition and Execution

- The recursion theorem and parametrization theorem allow a Turing-computable function to be defined recursively, i.e., with its own index: $\varphi_n = [\ldots n \ldots x \ldots]$. We anticipated that because Turing machine and recursive functions are equivalent models, but only the latter model explicitly exhibits recursion.
- During its computation, a recursively defined function φ_n may call itself with different actual parameters. Such a function φ_n *can* be computed on a Turing machine. How?
  - Its Turing program δ must be able to activate itself with new actual parameters.
  - For each activation of δ, TM allocates a new activation record on its tape. The activation record contains the new actual parameters and empty field for the result of this activation (i.e. call of φ_n).
  - When the result is computed, it is written into the empty field of the callee's activation record. Next, some previously designated state, called the return state, is entered. This enables the awaiting caller to resume its execution.
  - The caller then reads the result, deletes the callee's activation record on the tape and continues its execution right after the call.
  - Obviously, the machine uses its tape as a stack of activation records: when a new call of φ_n is made (completed), the corresponding activation record is pushed on (popped from) the stack.
- *This mechanism is used in general-purpose computer to handle procedure calls during program execution.*

Ch 8. Incomputable Problems

- Diagonalization, combined with self-reference, made it possible to discover the first incomputable problem, i.e., a decision problem called the *Halting Problem,* for which there is no single algorithm capable of solving every instance of the problem.
- After that many other incomputable problems were discovered in various fields of science.
- Incompatibility is a constituent part of reality.

Decision Problems and Other Kinds of Problems

We define the following four kinds of computational problems:

- **Decision problems.** The solution of a decision problem is the *answer* YES *or* NO.
- **Search problems.** The solution of a search problem is *an element* of a given set such that the element has a given property.
- **Counting problems.** The solution of a counting problem is *the number of elements* of a given set that have a given property.
- **Generating problems.** The solution of a generating problem is *a list of elements* of a given set that have a given property.

In the following, *w*e will focus on decision problems.

Language of a Decision Problem

Let $\mathcal{D}$ be a decision problem. We define the following notions:

- The instance d of $\mathcal{D}$ is obtained by replacing the variables in the definition of $\mathcal{D}$ with actual data.
- An instance $d \in \mathcal{D}$ is positive or negative if the answer to d is YES or NO, respectively.
- Let Σ be the input alphabet of a TM. The coding function is a computable and injective function $code : \mathcal{D} \to \Sigma^*$ that transforms every instance $d \in \mathcal{D}$ into a word $code(d)$ over Σ. We usually write $\langle d \rangle$ instead of $code(d)$.
- The language of a decision problem $\mathcal{D}$ is the set $L(\mathcal{D}) = \{\langle d \rangle \in \Sigma^* \mid d \text{ is a positive instance of } \mathcal{D}\}$.

Obviously: An instance d of $\mathcal{D}$ is positive $\Leftrightarrow \langle d \rangle \in L(\mathcal{D})$

Hence: *Solving a decision problem $\mathcal{D}$ can be reduced to recognizing the set $L(\mathcal{D})$ in Σ^*.*

Decidability of Decision Problems

We can now extend our terminology about sets to decision problems.

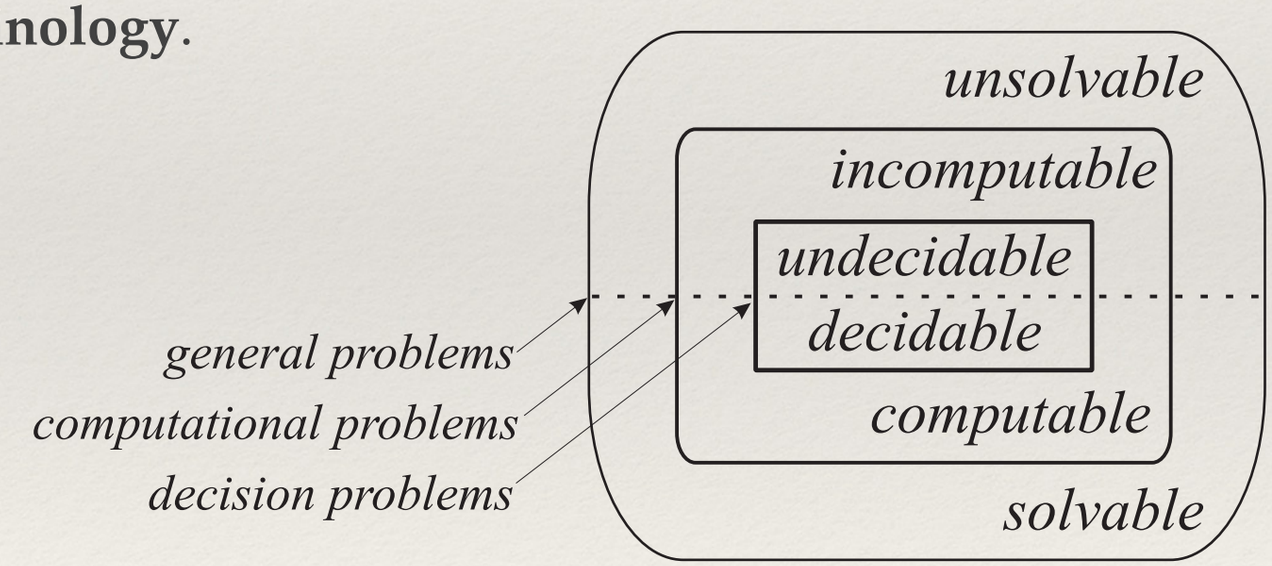
Definition. Let $\mathcal{D}$ be a decision problem. We say that the problem

$\mathcal{D}$ is decidable (or computable) if $L(\mathcal{D})$ is decidable set;

$\mathcal{D}$ is semi-decidable if $L(\mathcal{D})$ is semi-decidable set;

$\mathcal{D}$ is undecidable (or incomputable) if $L(\mathcal{D})$ is undecidable set.

- **Terminology**.



Subproblems of a Decision pProblem

Often we encounter a decision problem that is a *special version* of another decision problem. Is there any connection between the decidabilities of the two problems?

Definition. A decision problem $\mathcal{D}_{Sub}$ is a subproblem of a decision problem $\mathcal{D}_{Prob}$ if $\mathcal{D}_{Sub}$ is obtained from $\mathcal{D}_{Prob}$ by imposing additional restrictions on (some of) the variables of $\mathcal{D}_{Prob}$.

Theorem. *Let* $\mathcal{D}_{Sub}$ *be a subproblem of a decision problem* $\mathcal{D}_{Prob}$ *. Then*:

$\mathcal{D}_{Sub}$ *is undecidable* $\Rightarrow$ $\mathcal{D}_{Prob}$ *is undecidable.*

There Is an Incomputable Problem - Halting Problem

Definition. The Halting Problem $\mathcal{D}_{\mathcal{H}alt}$ is defined by $\mathcal{D}_{\mathcal{H}alt} \equiv$ "Given a Turing machine T and word $w \in \Sigma^*$, does T halt on w?"

Theorem. *The Halting Problem $\mathcal{D}_{\mathcal{H}alt}$ is undecidable.*

Before we go to the proof, we introduce two important sets (languages).

Definition. The universal language, denoted by $\mathcal{K}_o$, is the language of the Halting Problem, that is $\mathcal{K}_o = L(\mathcal{D}_{\mathcal{H}alt}) = \{\langle T,w\rangle \mid T \text{ halts on } w\}$.

Definition. The diagonal language, denoted by $\mathcal{K}$, is defined by $\mathcal{K} = \{\langle T,T\rangle \mid T \text{ halts on } \langle T\rangle\}$.

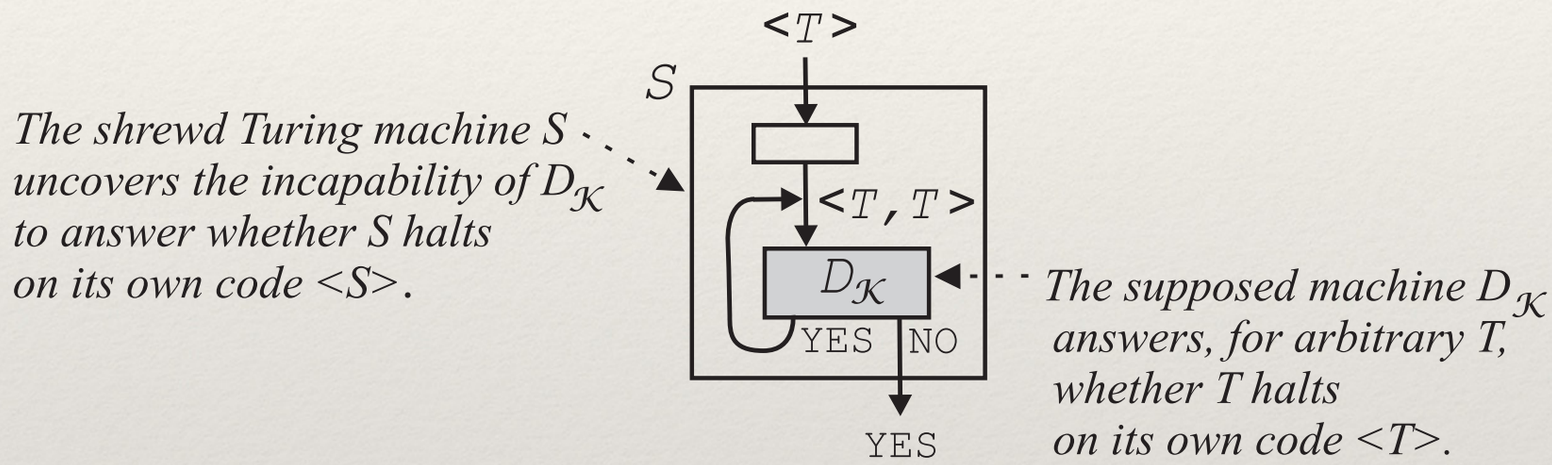
Observe that $\mathcal{K}$ is the language $L(\mathcal{D}_{\mathcal{H}})$ of the decision problem "Given a Turing machine T, does T halt on its own code $\langle T\rangle$?"

Proof of the theorem.

Lemma. *The set $\mathcal{K}$ is undecidable.*

Proof of the lemma. Suppose that $\mathcal{K}$ is decidable. Then there is a TM $D_{\mathcal{K}}$ that decides $\mathcal{K}$.

Using $D_{\mathcal{K}}$ we construct a new, shrewd TM S as follows.



If S is given as input $\langle S \rangle$, it puts the $D_{\mathcal{K}}$ in trouble: $D_{\mathcal{K}}$ is unable to decide $\langle S,S \rangle \in ? \mathcal{K}$.

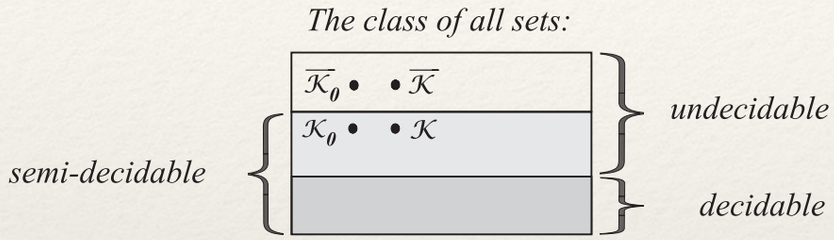
So $D_{\mathcal{K}}$ does not exist; $\mathcal{K}$ is undecidable and $\mathcal{D}_{\mathcal{H}}$ is incomputable (undecidable). □

Since $\mathcal{D}_{\mathcal{H}}$ is a subproblem of $\mathcal{D}_{\mathcal{H}alt}$, also $\mathcal{D}_{\mathcal{H}alt}$ is incomputable (undecidable). □

Consequences

Theorem. *The sets $\mathcal{K}_0$ and $\mathcal{K}$ are semi-decidable.*

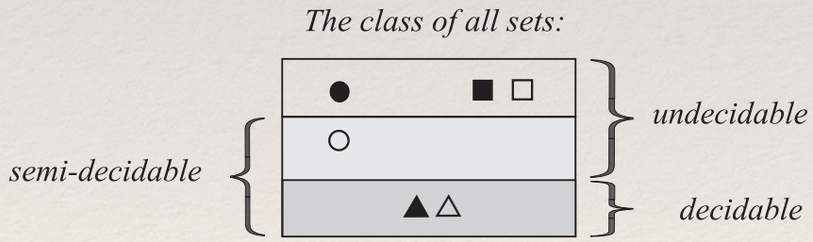
Theorem. *The sets $\overline{\mathcal{K}_0}$ and $\overline{\mathcal{K}}$ are not semi-decidable.*



Similarly holds for the corresponding complementary decision problems.

There are three possibilities for the decidability of a set S and its complement:

1. both are decidable;
2. both are undecidable; one is semi-decidable and the other is not;
3. both are undecidable and neither is semi-decidable.



Similarly holds for the corresponding complementary decision problems.

Corollary. Incomputable functions *exist* (*for example, the characteristic function* $\chi_{\mathcal{K}0}$).

Some Other Incomputable Problems

There are many other incomputable problems. For example, incomputable are:

- some problems about Turing machines
- Post's correspondence problem
- some problems about algorithms and computer programs
- some problems about programming languages and grammars
- some problems about computable functions
- some problems from number theory
- some problems from algebra
- some problems from analysis
- some problems from topology
- some problems from mathematical *logic*
- some problems about games

Ch 9. Methods of Proving the Incomputability

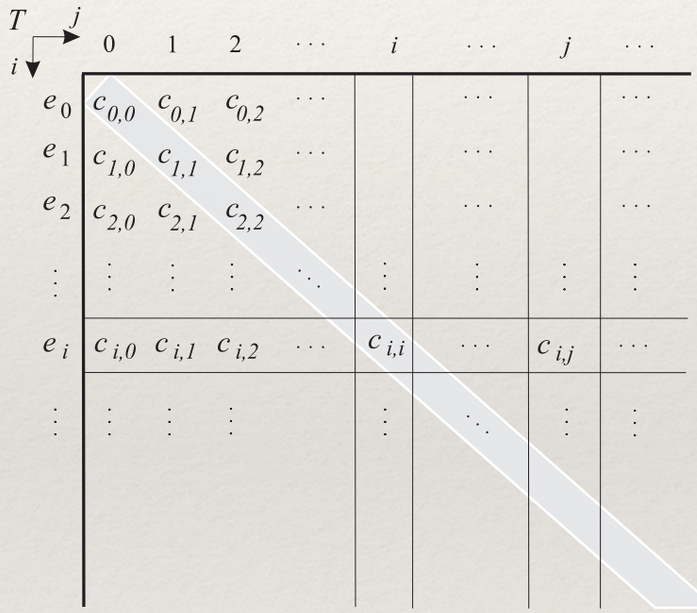
- Today we have at our disposal several methods of proving the undecidability of decision problems. These are:
  - proving by *diagonalization*
  - proving by *reduction*
  - proving by *Recursion Theorem*
  - proving by *Rice's Theorem*

Proving by Diagonalization

Direct Diagonalization

Let P be a property and $S = \{x \mid P(x)\}$. Let $\mathcal{T} \subseteq S$ such that $\mathcal{T} = \{e_0, e_1, e_2, \ldots\}$ and each e_i is uniquely represented as $e_i = (c_{i,0}, c_{i,1}, c_{i,2}, \ldots)$, where $c_{i,j} \in C$ for some set C. Suppose we believe that $\mathcal{T} \subsetneq S$, i.e. S *cannot* be fully exhibited by listing the elements of $\mathcal{T}$. Can we prove that?

Imagine this table:



The diagonal elements define the diagonal $d = (c_{0,0}, c_{1,1}, c_{2,2}, \ldots)$.

Suppose we find a function $sw : C \to C$ where $sw(c) \neq c, \forall c \in C$. Call sw the switching function.

Define $sw(d) = (sw(c_{0,0}), sw(c_{1,1}), sw(c_{2,2}), \ldots)$ and note that $sw(d) \neq e_i$ for $\forall i$. So, $sw(d) \notin \mathcal{T}$.

Suppose that $sw(d)$ has the property P. Then $sw(d) \in S$. Hence $sw(d) \in S\text{-}\mathcal{T}$ and $\mathcal{T} \subsetneq S$.

Indirect Diagonalization

Let P be a property of algorithms. Question: *Is there an algorithm D_P capable of deciding, for an arbitrary algorithm A, whether or not A has property P*? Suppose that we doubt that D_P exists. How can we proved that? First, recall that algorithms (=TMs) can be enumerated. Imagine this table:

$A_i(j)$ is the result of applying the algorithm A_i on input j.

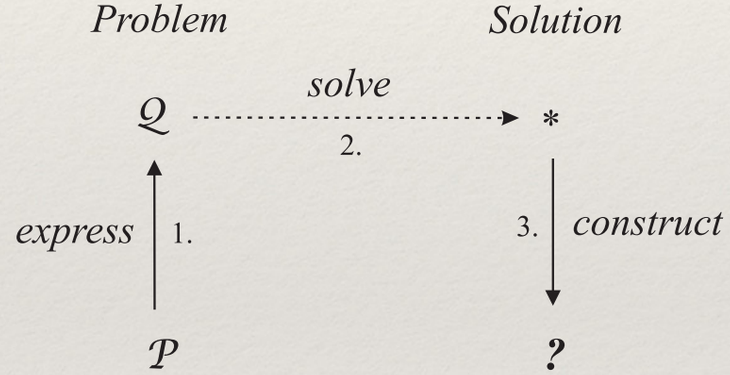
Suppose that D_P exists. Try to construct an algorithm S, such that 1) S uses D_P and 2) if S is applied on $\langle S \rangle$, it *uncovers the inability* of D_P to decide whether or not S has the property P. *If* such a shrewd algorithm S *is* constructed, then D_P doesn't exist, and P is undecidable.

Proving by Reduction

Reductions in General

Given a problem $\mathcal{P}$, instead of solving it directly, we may try to solve it *indirectly* by executing the following scenario:

1. *express* $\mathcal{P}$ in terms of some other problem $\mathcal{Q}$;
2. *solve* $\mathcal{Q}$;
3. *construct* the solution to $\mathcal{P}$ by using the solution to $\mathcal{Q}$ only.



Solving the problem $\mathcal{P}$ is reduced to (substituted by) solving the problem $\mathcal{Q}$.

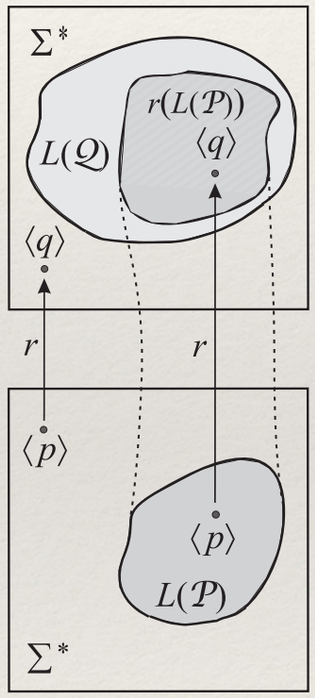
To express $\mathcal{P}$ in terms $\mathcal{Q}$ we need a computable function $r : \Sigma^* \to \Sigma^*$ such that

1. for every instance $p \in \mathcal{P}$, r maps the code $\langle p \rangle$ into a code $\langle q \rangle$, where $q \in \mathcal{Q}$;
2. the solution to $p \in \mathcal{P}$ can be computed from the solution to $q \in \mathcal{Q}$, where $\langle q \rangle = r(\langle p \rangle)$.

If such an r is found, it is called the reduction of the problem $\mathcal{P}$ to the problem $\mathcal{Q}$, and we say that $\mathcal{P}$ is reducible to the problem $\mathcal{Q}$, and denote this by $\mathcal{P} \leq \mathcal{Q}$.

The m-Reduction

Definition. Let $\mathcal{P}$ and $\mathcal{Q}$ be decision problems. A reduction $r : \Sigma^* \to \Sigma^*$ is said to be the m-reduction of $\mathcal{P}$ to $\mathcal{Q}$ if the following additional condition is met: $\langle p \rangle \in L(\mathcal{P}) \iff r(\langle p \rangle) \in L(\mathcal{Q})$. In this case we say that $\mathcal{P}$ is m-reducible to $\mathcal{Q}$ and denote this by $\mathcal{P} \leq_m \mathcal{Q}$, .



Obviously $r(L(\mathcal{P})) \subseteq L(\mathcal{Q})$.

If $r(L(\mathcal{P})) \subset L(\mathcal{Q})$, then r reduces $\mathcal{P}$ to a proper subproblem of $\mathcal{Q}$.

If $r(L(\mathcal{P})) = L(\mathcal{Q})$, then r reduces $\mathcal{P}$ to $\mathcal{Q}$.

Definition. When the above $r : \Sigma^* \to \Sigma^*$ is *injective*, we say r is the 1-reduction of $\mathcal{P}$ to $\mathcal{Q}$. We also say that $\mathcal{P}$ is 1-reducible to $\mathcal{Q}$ and denote this by $\mathcal{P} \leq_1 \mathcal{Q}$, .

Theorem. *Let $\mathcal{P}$ and $\mathcal{Q}$ be decision problems. Then*:

a) $\mathcal{P} \leq_m \mathcal{Q}, \ \wedge \ \mathcal{Q}, \quad$ *is decidable* $\Rightarrow \ \mathcal{P} \quad$ *is decidable*

b) $\mathcal{P} \leq_m \mathcal{Q}, \ \wedge \ \mathcal{Q},$ *is semi-decidable* $\Rightarrow \ \mathcal{P}$ *is semi-decidable*

Corollary. *Let $\mathcal{U}$ and $\mathcal{Q}$ be decision problems. Then*:

$\mathcal{U}$ *is undecidable* $\wedge \ \mathcal{U} \leq_m \mathcal{Q}, \ \Rightarrow \ \mathcal{Q}$ *is undecidable*

This is the backbone of the following method.

Method. The undecidability of a decision problem $\mathcal{Q}$ can be proved as follows:

1. *Select* an undecidable problem $\mathcal{Q}$
2. *Prove* that $\mathcal{U} \leq_m \mathcal{Q}$
3. Conclude: $\mathcal{Q}$ is undecidable

Proving by the Recursion Theorem

Recall the Fixed-Point Theorem: *Every computable function has a fixed point*.

This reveals the following method for proving the incompatibility of functions:

Method. Let g be a function with no fixed point. Then, g is not computable, i.e., it is not total, or it is incomputable, or both. If we prove that g *is* total, then g must be incomputable.

We can develop this into a method for proving the undecidability of decision problems.

Method. Undecidability of a decision problem $\mathcal{D}$ can be proved as follows:

1. *Suppose* that $\mathcal{D}$ is a decidable problem
2. *Construct* a *computable* function g using the characteristic function $\chi_{L(\mathcal{D})}$
3. *Prove* that g has no fixed point
4. This is a contradiction with the Fixed-Point Theorem
5. Conclude that $\mathcal{D}$ is undecidable

Proving by the Rice's Theorem

Rice's Theorem for Functions

Definitions.

- Let P be a property sensible for functions. We say that P is intrinsic property of functions *if* functions are viewed only as mappings from one set to another; that is, P is insensitive to the machine, algorithm, and program, that are used to compute function values.
- Let P be a property intrinsic to functions, and φ an arbitrary p.c. function. Define the following decision problem: D_P = "Does a p.c. function φ have the property P?" We say that P is a decidable property *if* D_P is a decidable problem.
- We say that an intrinsic property of functions is trivial *if either* every p.c. function has the property P *or* no p.c. function has the property P.

Theorem. *Let P be an arbitrary intrinsic property of p.c. functions. Then:*

$$P \text{ is a decidable property} \iff P \text{ is trivial}$$

Based on this, we obtain the following method.

Method. Given a property P, the undecidability of the decision problem D_P = "Does a p.c. function φ have the property P?" can be proved as follows:

1. *Show* that P meets the following conditions
   - *a.* P is a property sensible for functions
   - *b.* P is insensitive to the machine, algorithm, or program used to compute φ
2. If P fulfills the above conditons, then *show* that P is non-trivial. To do this,
   - *c. find* a p.c. function that has the property P
   - *d. find* a p.c. function that does not have the property P

If all the steps are successful, then the problem D_P is undecidable.

Rice's Theorem for Index Sets

Definitions.

- Let P be an intrinsic property of p.c. functions. Define $\mathcal{F}$ to be the class of all the p.c. functions having the property P; that is, $\mathcal{F} = \{\psi \mid \psi \text{ has the property } P\}$.
- The decision problem D_P can be now rewritten as $D_P = \text{“}\varphi \in ?\ \mathcal{F}\text{”}$.
- Define $\text{ind}(\mathcal{F}) = \bigcup_{\psi \in \mathcal{F}} \text{ind}(\psi)$. In other words, $\text{ind}(\mathcal{F})$ is the set of *all* of the indexes of *all* Turing Machines that compute *any* of the functions $\mathcal{F}$.
- The decision problem D_P can now be rewritten as $D_P = \text{“}x \in ?\ \text{ind}(\mathcal{F})\text{”}$.

So, D_P is a decidable problem *iff* $\text{ind}(\mathcal{F})$ is a decidable set.

But, when is $\text{ind}(\mathcal{F})$ decidable?

Theorem. *Let $\mathcal{F}$ be an arbitrary set of p.c. functions. Then:*

$$\text{ind}(\mathcal{F}) \textit{ is a decidable set} \iff \text{ind}(\mathcal{F}) \textit{ is either } \emptyset \textit{ or } \mathbb{N}$$

Rice's Theorem for Sets

Definitions.

- Let R be a property sensible sets. We say that R is intrinsic property of sets *if* it is independent of the way of recognizing the sets; that is, R is insensitive to the machine, algorithm, and program, that are used to recognize the sets.
- Let R be a property intrinsic to sets, and $\mathcal{X}$ an arbitrary c.e. set. Define the following decision problem: D_R = "Does a c.e. set $\mathcal{X}$ have the property R?" We say that R is a decidable property *if* D_R is a decidable problem.
- We say that an intrinsic property of c.e. sets is trivial *if* it holds for all c.e. sets or for none.

Theorem. *Let R be an arbitrary intrinsic property of c.e. sets. Then:*

$$R \text{ is a decidable property} \iff R \text{ is trivial}$$

… cont'd (Proving by the Rice's Theorem)

… cont'd (Rice's Theorem for Sets)

Based on this, we obtain the following method.

Method. Given a property R, the undecidability of the decision problem D_R = "Does a c.e. set $\mathcal{X}$ have the property R?" can be proved as follows:

1. *Show* that R meets the following conditions
   - *a.* R is a property sensible for sets
   - *b.* R is insensitive to the machine, algorithm, or program used to recognise $\mathcal{X}$
2. If R fulfills the above conditons, then *show* that R is non-trivial. To do this,
   - *c. find* a c.e. set that has the property R
   - *d. find* a c.e. set that does not have the property R

If all the steps are successful, then the problem D_R is undecidable.

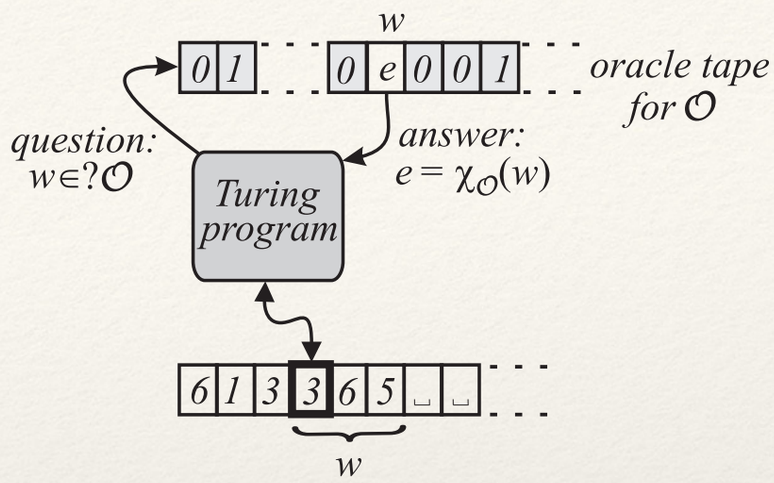
Part III. RELATIVE COMPUTABILITY

- Ch 10 Computation with external help
- Ch 11 Degrees of unsolvability
- Ch 12 The Turing hierarchy of unsolvability
- Ch 13 The class $\mathcal{D}$ of degrees of unsolvability
- Ch 14 C.E. degrees and the priority method
- Ch 15 The arithmetical hierarchy

Ch 10. Computation With External Help

- What if an *unsolvable* decision problem, for example the Halting Problem, were *somehow made solvable* with some hypothetical procedure?
- We must assume that the hypothetical procedure would *not* be the *ordinary* Turing machine; otherwise, our theory would become inconsistent.
- The question can be explored by introducing the *oracle Turing machine*.
- Such a machine was conceived by Alan Turing and further developed by Emil Post.

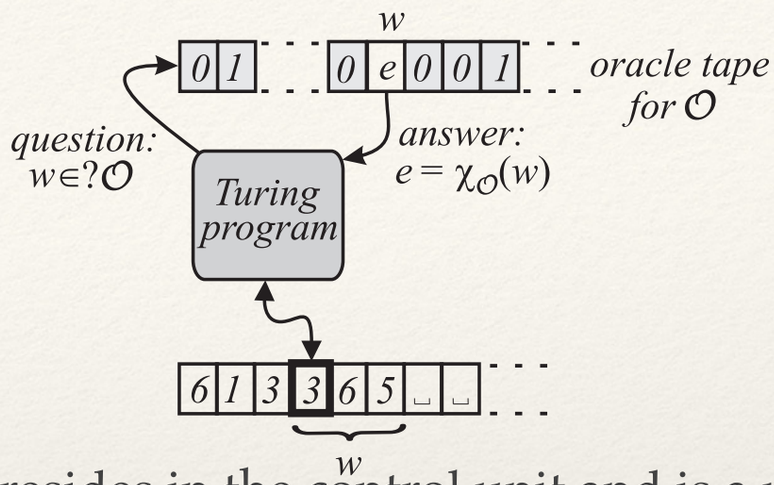
The Oracle Turing Machine



Definitions.

- The oracle Turing machine (o-TM) with oracle set $\mathcal{O}$ (in short $\mathcal{O}$-TM) consists of a control unit, an input tape, an oracle Turing program, an *oracle tape*, and a *set* $\mathcal{O}$. Formally, $\mathcal{O}$-TM is a eight-tuple $T^{\mathcal{O}} = (Q, \Sigma, \Gamma, \delta\tilde{}, q_1, \sqcup, F, \mathcal{O})$.
- *As usual*, the control unit is in a state from $Q = \{q_1,\ldots,q_s\}$, $s \geqq 1$; the initial state is q_1 and $F \subseteq Q$ is the set of final states. The input tape is one-way unbounded tape with a window; the tape alphabet is $\Gamma = \{z_1,\ldots,z_t\}$, $t \geqq 3$ with $z_1 = 0$, $z_2 = 1$, $z_t = \sqcup$. The input alphabet is a set Σ, where $\{0,1\} \subseteq \Sigma \subseteq \Gamma$-$\{\sqcup\}$.
- The oracle set $\mathcal{O}$ is an arbitrary subset of Σ^*. The oracle tape is a *read-only* tape that contains, for *each* $w \in \Sigma^*$, the value $\chi_{\mathcal{O}}(w)$ of the characteristic function $\chi_{\mathcal{O}} : \mathcal{O} \to \{0,1\}$. There is no window; yet, it can *immediately* find and return $\chi_{\mathcal{O}}(w)$, for any $w \in \Sigma^*$!

… cont'd (The Oracle Turing Machine)



… cont'd (Definitions)

- The oracle Turing program (in short o-TP) resides in the control unit and is a partial function $\delta^{\sim} : Q \times \Sigma \times \{0,1\} \rightharpoonup Q \times \Gamma \times \{\text{Left, Right, Stay}\}$. Thus, any instruction is of the form $\delta^{\sim}(q,z,e)=(q',z',D)$, and is interpreted as follows: *If the control unit is in the state q, and reads z and e from the input and oracle tape, resp., then it changes to the state q', writes z' to the input tape, and moves the window in the direction D. Here, e denotes $\chi_{\mathcal{O}}(w)$, where w is the word starting under the window and ending in the rightmost nonempty cell of the input tape.*
- Before the $\mathcal{O}$-TM is started, the following takes place: 1) an input word from Σ^* is written to the beginning of the input tape; 2) the window is shifted to the beginning of the input tape; 3) control unit is set to q_1; and 4) an oracle set $\mathcal{O} \subseteq \Sigma^*$ is fixed.
- From now on $\mathcal{O}$-TM operates in a mechanical stepwise fashion, as instructed by $\delta^{\sim}$.
- $\mathcal{O}$-TM halts when either enters a final state, or reads z and e in q such that $\delta^{\sim}(q,z,e)\uparrow$.

Some Basic Properties of o-TMs

Properties:

- Oracle Turing programs $\delta^{\sim}$ are not affected by changes in $\mathcal{O}$.
- During the execution, oracle Turing programs $\delta^{\sim}$ can ignore $\mathcal{O}$.

Theorem. *Oracle computation is a generalization of ordinary computation.*

Coding and Enumeration of o-TMs

- **Coding**
  - Let $T^{\mathcal{O}} = (Q, \Sigma, \Gamma, \delta, q_1, \sqcup, F, \mathcal{O})$ be an arbitrary $\mathcal{O}$-TM and $\delta\tilde{}(q_i, z_j, e) = (q_k, z_\ell, D_m)$ an instruction of its o-TP. We encode the instruction by the word $K = 0^i10^j10^e10^k10^\ell10^m$, where D_1 = Left, D_2 = Right, and D_3 = Stay. In this way, we encode each instruction of the oracle program $\delta\tilde{}$.
  - From the codes $K_1, \ldots, K_r$ we construct the code of $\delta\tilde{}$ as follows: $\langle\delta\tilde{}\rangle = 111K_111K_211\ldots11K_r111$. This is also the code of $T^{\mathcal{O}}$, i.e. $\langle T^{\mathcal{O}}\rangle = \langle\delta\tilde{}\rangle$. (Note that $\langle T^{\mathcal{O}}\rangle$ is independent of the particular $\mathcal{O}$.)
- **Enumeration**
  - We interpret $\langle T^{\mathcal{O}}\rangle$ to be the binary code of some natural number, called the index of $T^{\mathcal{O}}$.
  - **Convention**: Any natural number whose binary code is not of the above form is an index of the empty $\mathcal{O}$-TM (its o-TP is everywhere undefined).
- *Every natural number is the index of exactly one o-TM (and $\mathcal{O}$-TM); we can speak of i-th $\mathcal{O}$-TM, $T^{\mathcal{O}}_i$.*

Computation with Oracles

Generalization of Classical Definitions

Function Computation

- **Definition**. Let $\mathcal{O} \subseteq \Sigma^*$, $k \geqslant 1$, and $\varphi_{\geqslant}: (\Sigma^*)^k \to \Sigma^*$ a function. We say that
  - φ is $\mathcal{O}$-computable *if* there is an $\mathcal{O}$-TM that can compute φ anywhere on $\mathrm{dom}(\varphi)$ and $\mathrm{dom}(\varphi) = (\Sigma^*)^k$;
  - φ is partial $\mathcal{O}$-computable (or $\mathcal{O}$-p.c.) *if* there is an $\mathcal{O}$-TM that can compute φ anywhere on $\mathrm{dom}(\varphi)$;
  - φ is $\mathcal{O}$-incomputable *if* there is *no* $\mathcal{O}$-TM that can compute φ anywhere on $\mathrm{dom}(\varphi)$.
- **Definition**. Given any $\mathcal{O} \subseteq \Sigma^*$, $n \geqslant 0$, and $k \geqslant 1$, the oracle Turing machine $T^{\mathcal{O}}_n$ computes a function $\Psi^{\mathcal{O},(k)}_x : (\Sigma^*)^k \to \Sigma^*$ called the k-ary proper functional of $T^{\mathcal{O}}_n$. When k is understood, we omit it.

… cont'd (Computation with Oracles)

… cont'd (Generalization of Classical Definitions

Set Recognition

- **Definition**. Let $\mathcal{O} \subseteq \Sigma^*$ be an oracle set. For an arbitrary $S \subseteq \Sigma^*$ we say that
  - S is $\mathcal{O}$-decidable (or $\mathcal{O}$-computable) in Σ^* *if* χ_S is $\mathcal{O}$-computable function on Σ^*;
  - S is $\mathcal{O}$-semi-decidable (or $\mathcal{O}$-c.e.) in Σ^* *if* χ_S is $\mathcal{O}$-computable function on S;
  - S is $\mathcal{O}$-undecidable (or $\mathcal{O}$-incomputable) in Σ^* *if* χ_S is $\mathcal{O}$-incomputable function on Σ^*.

… cont'd (Computation with Oracles)

… cont'd (Generalization of Classical Definitions

Index Set

- **Lemma**.(Generalized Padding Lemma) *An $\mathcal{O}$-p.c. function has countably infinitely many indexes. Given one of them, the others can be generated.*

- **Definition** (Index set of $\mathcal{O}$-p.c. function). The index set of an $\mathcal{O}$-p.c. function φ is the set $\mathrm{ind}^{\mathcal{O}}(\varphi) = \{x \in \mathbb{N} \mid \Psi_x^{\mathcal{O}} \simeq \varphi\}$.

Convention.

From now on $\mathbb{N}$ *will be the universe.*

From now on we will focus on single-argument functions.

Other Ways to Make External Help Available

- **Kleene** introduced external help to partial recursive (p.r.) functions. Given $\mathcal{O} \subseteq \mathbb{N}$, he added the characteristic function $\chi_{\mathcal{O}}$ to the set $\{\zeta, \pi_i^k, \sigma\}$. Any function that can be constructed by these initial functions by finite number of applications of the rules of construction (i.e. composition, primitive recursion, μ-operation) is called p.r. relative to $\mathcal{O}$.

- **Post** introduced external help into his canonical systems by hypothetically adding primitive assertions expressing the (non)membership in $\mathcal{O}$.

- Davis proved the equivalence of Kleene's and Post's approaches.

- Based on this and following the Computability Thesis the following thesis was proposed:

Relative Computability Thesis

Basic intuitive concepts of computing with external help are appropriately formalized as follows:

"algorithm with external help" $\leftrightarrow$ *oracle Turing program*

"computation with external help" $\leftrightarrow$ *execution of an oracle TP on o-TM*

function "computable with external help" $\leftrightarrow$ *$\mathcal{O}$-p.c. function*

Instead of the o-TM we can use any other equivalent model.

Ch 11. Degrees of Unsolvability

- We saw that there exist solvable and unsolvable computational problems.
- So, it makes sense to talk about the "*degree of unsolvability*."
- But, our understanding of the notion "degree of unsolvability" is just *intuitive*.
- Now, we want to *formalize* it.

Turing Reduction

Definition: (Turing Reduction) Let $\mathcal{A}, \mathcal{B} \subseteq \mathbb{N}$ be arbitrary sets. We say that $\mathcal{A}$ is Turing reducible (in short, *T-reducible*) to $\mathcal{B}$, if $\mathcal{A}$ is $\mathcal{B}$-decidable. We denote this by $\mathcal{A} \leq_T \mathcal{B}$, which reads "*If $\mathcal{B}$ is decidable, then also $\mathcal{A}$ is decidable.*" The relation $\leq_T$ is called the Turing reduction (in short, *T-reduction*).

Let $\mathcal{P}$ and $\mathcal{Q}$ be decision problems and $\mathrm{L}(\mathcal{P})$ and $\mathrm{L}(\mathcal{Q})$ their languages. We say that the *problem $\mathcal{P}$* is *reducible* to the *problem $\mathcal{Q}$*, (and denote this by $\mathcal{P} \leq_T \mathcal{Q}$) if $\mathrm{L}(\mathcal{P}) \leq_T \mathrm{L}(\mathcal{Q})$.

Basic Properties of the Turing Reduction

Theorem. *Let $\mathcal{A}$ be a decidable set. Then $\mathcal{A} \leq_T \mathcal{B}$ for arbitrary set $\mathcal{B}$.*

Theorem. *For every set S it holds that $\overline{S} \leq_T S$.*

Theorem. *If two sets are related by $\leq_m$, then they are also related by $\leq_T$.*

Theorem. *If two sets are related by $\leq_T$, they may not be related by $\leq_m$.*

Theorem. *For arbitrary sets $\mathcal{A}, \mathcal{B}$ it holds: $\mathcal{A} \leq_T \mathcal{B} \quad \wedge \quad \mathcal{B}$ is decidable $\Rightarrow$ $\mathcal{A}$ is decidable*

Corollary. *For arbitrary sets $\mathcal{A}, \mathcal{B}$ it holds: $\mathcal{A}$ is undecidable $\wedge\, \mathcal{A} \leq_T \mathcal{B} \Rightarrow \mathcal{B}$ is undecidable*

Theorem. *Turing reduction $\leq_T$ is a reflexive and transitive relation.*

Turing Degrees

Definition. Let $\mathcal{A}, \mathcal{B} \subseteq \mathbb{N}$ be arbitrary sets. We say that $\mathcal{A}$ is Turing-equivalent (in short, *T-equivalent*) to $\mathcal{B}$, if $\mathcal{A} \leq_T \mathcal{B} \wedge \mathcal{B} \leq_T \mathcal{A}$. We denote this by $\mathcal{A} \equiv_T \mathcal{B}$, and read "*If one of* $\mathcal{A}, \mathcal{B}$ *were decidable, also the other would be decidable.*" The relation $\equiv_T$ is called the Turing equivalence (in short, *T-equivalence*).

Theorem. *For every set* S *it holds that* $S \equiv_T S$.

Definition. A Turing degree (in short, *T-degree*) of a set S, denoted by $\deg(S)$, is the equivalence class $\{\mathcal{X} \in 2^{\mathbb{N}} \mid \mathcal{X} \equiv_T S\}$.

Formalization. *The intuitive notion of the degree of unsolvability is formalized by*

"degree of unsolvability" $\leftrightarrow$ *Turing degree*

Theorem. *There exist at least two T-degrees:*

$$\deg(\emptyset) = \{\mathcal{X} \in 2^{\mathbb{N}} \mid \mathcal{X} \equiv_T \emptyset\} \quad \textit{and}$$

$$\deg(\mathcal{K}) = \{\mathcal{X} \in 2^{\mathbb{N}} \mid \mathcal{X} \equiv_T \mathcal{K}\}$$

The Relation <

Definition. Let $\deg(\mathcal{A})$ and $\deg(\mathcal{B})$ be arbitrary T-degrees. Then $\deg(\mathcal{A})$ is lower than $\deg(\mathcal{B})$, denoted by $\deg(\mathcal{A}) < \deg(\mathcal{B})$, if $\mathcal{A} <_T \mathcal{B}$ (i.e. $\mathcal{A} \leq_T \mathcal{B} \wedge \mathcal{A} \not\equiv_T \mathcal{B}$).

Theorem. *The relation $<_T$ is irreflexive.*

Since $\emptyset <_T \mathcal{K}$, we find that $\deg(\emptyset) < \deg(\mathcal{K})$.

So, the degree of undecidability of decidable decision problems is lower than the degree of unsolvability of undecidable decision problems T-equivalent to the Halting Problem.

But we have intuitively anticipated this!

Nevertheless, this formalization will enable us to discover in the next chapter a surprising fact that there are many more *other* degrees of unsolvability!

Ch 12. The Turing hierarchy of unsolvability

- At this point we only know of *two* degrees of unsolvability: deg($\emptyset$) and deg($\mathcal{K}$).
- We will now prove that there are *infinitely many other* degrees of unsolvability.
- To prove this we will introduce the *Turing jump operator*.

The Turing Jump

Recall: The Halting Problem is the question

"Does T halt on input $\langle T \rangle$?"

and its language is

$$\begin{aligned}\mathcal{K} &= \{\langle T \rangle \mid T \text{ halts on input } \langle T \rangle\} \\ &= \{x \mid T_x \text{ halts on input } x\} \\ &= \{x \mid \Psi_x(x)\downarrow\}.\end{aligned}$$

Let S be an arbitrary set. Let us adapt the Halting Problem for oracle Turing machines T^S:

"Does T^S halt on input $\langle T^S \rangle$?"

and define, in the similar fashion, its language as

$$\begin{aligned}\mathcal{K}^S &= \{\langle T^S \rangle \mid T^S \text{ halts on input } \langle T^S \rangle\} \\ &= \{x \mid T_x{}^S \text{ halts on input } x\} \\ &= \{x \mid \Psi_x{}^S(x)\downarrow\}.\end{aligned}$$

Definition: The **Turing jump of a set** S is the set $\mathcal{K}^S$ defined by $\mathcal{K}^S = \{x \mid \Psi_x{}^S(x)\downarrow\}$. The set $\mathcal{K}^S$ we also denote by S'.

Properties of the Turing Jump of a Set

Lemma. *$\mathcal{A}$ is S-c.e. $\Longrightarrow$ $\mathcal{A}$ is S′-decidable*

Theorem. *Let S be and arbitrary set. Then $S \leq_T S'$.*

Theorem. *Let S be and arbitrary set. Then:*

a) S′ is S-undecidable (i.e., $S' \not\leq_T S'$)

b) S′ is S-c.e.

Corollary. *Let S be and arbitrary set. Then* $\deg(S) < \deg(S')$.

By taking S := $\mathcal{K}$ in the corollary, we obtain $\deg(\mathcal{K}) < \deg(\mathcal{K}')$.

We have discovered that there is a T-degree that is higher than $\deg(\mathcal{K})$.

Hence, there exist decision problems that are more difficult than the Halting Problem.

Hierarchies of T-Degrees

Definition: The ***n*th Turing jump of the set** S is the set $S^{(n)}$, which is inductively defined as follows: $S^{(n)} = S$, if $n = 0$;

$S^{(n)} = (S^{(n-1)})'$, if $n \geqslant 1$.

Theorem. *Let S be and arbitrary set. Then:*

a) $S^{(n)} <_T S^{(n+1)}$

b) $S^{(n+1)}$ *is* $S^{(n)}$*-c.e.*

c) $\deg(S^{(n)}) < \deg(S^{(n+1)})$.

For every degree of unsolvability there is a higher degree of unsolvability. For every decision problem, even undecidable one, there is a more difficult decision problem. There is no most difficult decision problem.

The Jump Hierarchy

Now let us take $S := \emptyset$.

By applying the previous several times we obtain the following jump hierarchy of sets:

$$\emptyset^{(0)} <_T \emptyset^{(1)} <_T \emptyset^{(2)} <_T \ \ldots <_T \emptyset^{(i)} <_T \emptyset^{(i+1)} <_T \ \ldots$$

and the associated jump hierarchy of T-degrees

$$\deg(\emptyset^{(0)}) < \deg(\emptyset^{(1)}) < \deg(\emptyset^{(2)}) < \ \ldots < \deg(\emptyset^{(i)}) < \deg(\emptyset^{(i+1)}) < \ \ldots$$

Ch 13. The Class $\mathcal{D}$ of Degrees of Unsolvability

- We will now view the collection of T-degrees as a mathematical structure $\mathcal{D}$ endowed with the relation $<$ and the function $'$.
- This view will simplify our expression and the investigation of the properties of T-degrees.

The Structure $(\mathcal{D}, \leq, ')$

Definition. The class $\mathcal{D}$ of all T-degrees is defined as $\mathcal{D} = 2^{\mathbb{N}} / \equiv_T$.

Convention. The members of $\mathcal{D}$ are written by boldface characters, e.g. $\boldsymbol{a}$, $\boldsymbol{b}$, $\boldsymbol{c}$, $\boldsymbol{d}$, $\boldsymbol{0}$.

Theorem. $\mathcal{A} \equiv_T \mathcal{B} \;\Rightarrow\; \mathcal{A}' \equiv_T \mathcal{B}'$

We can define on $\mathcal{D}$ the relation $\leq$ (i.e., the reflexive closure of the relation $<$).

We can extend the function $'$ to be a function that maps a T-degree into a T-degree.

Definition. The Turing jump of a T-degree $\boldsymbol{d}$ is the T-degree $\boldsymbol{d'} = \deg(S')$, where S is an arbitrary member of $\boldsymbol{d}$.

Let us denote $\deg(\emptyset^{(i)})$ by $\boldsymbol{0}^{(i)}$.

The jump hierarchy is now $\boldsymbol{0}^{(0)} \leq \boldsymbol{0}^{(1)} \leq \boldsymbol{0}^{(2)} \leq \ \ldots \leq \boldsymbol{0}^{(i)} \leq \boldsymbol{0}^{(i+1)} \leq \ldots$

Some Basic Properties of $(\mathcal{D}, \leq, ')$

Cardinality of Degrees and of the Class $\mathcal{D}$

Given a T-degree, how many sets are there in it?
How many T-degrees are there in the class $\mathcal{D}$?

Theorem. *Every T-degree is countable.*

Theorem. *Every T-degree is countably infinite.*

Theorem. *The class $\mathcal{D}$ is uncountable. Its cardinality is $2^{\aleph_0}$*

The Class $\mathcal{D}$ as a Mathematical Structure

Theorem. $(\mathcal{D}, \leq)$ *is partially ordered.*

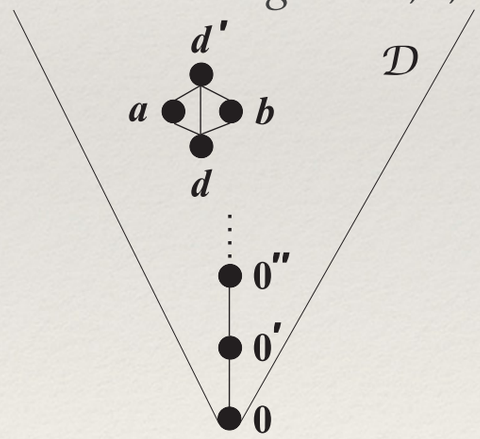
Definition. T-degrees $\boldsymbol{a},\boldsymbol{b}$ are ≤-incomparable (denoted by $\boldsymbol{a} \mid \boldsymbol{b}$) if neither $\boldsymbol{a} \leq \boldsymbol{b}$ nor $\boldsymbol{b} \leq \boldsymbol{a}$.

Theorem. *There exist T-degrees* $\boldsymbol{a},\boldsymbol{b}$, *such that* $\boldsymbol{0} \leq \boldsymbol{a},\boldsymbol{b} \leq \boldsymbol{0}'$ *and* $\boldsymbol{a} \mid \boldsymbol{b}$.

(The theorem was proved by Post and Kleene with their *Method of Finite Extensions*.)

The last theorem can be generalized.

Theorem. *For any T-degree* $\boldsymbol{d}$ *there exist T-degrees* $\boldsymbol{a},\boldsymbol{b}$, *such that* $\boldsymbol{d} \leq \boldsymbol{a},\boldsymbol{b} \leq \boldsymbol{d}'$ *and* $\boldsymbol{a} \mid \boldsymbol{b}$.



Theorem. *There are* $2^{\aleph_0}$ *mutually* ≤*-incomparable T-degrees in* $\mathcal{D}$.

The Class $\mathcal{D}$ as a Mathematical Structure

Distinguished T-Degrees

Since there are $\leq$-incomparable T-degrees, $\leq$ does *not* linearly order $\mathcal{D}$.
This gives rise to a series of questions about the existence of certain distinguished elements, such as minimal, least, greatest, maximal elements, upper bounds, lower bounds, least upper bounds, greatest lower bounds.

Theorem. *There is a* $\leq$-least *T-degree in* $(\mathcal{D}, \leq)$. *This is the T-degree* $\mathbf{0}$.

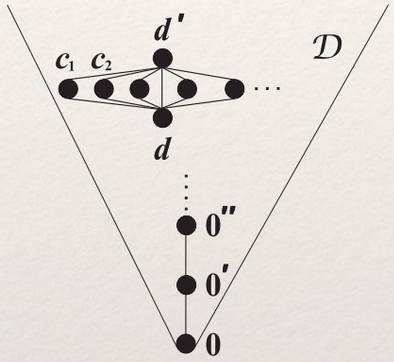
Theorem. *Any two T-degrees have a* $\leq$-least upper bound.

Theorem. *There is a pair of T-degrees that have no* $\leq$*-greatest lower bound.*

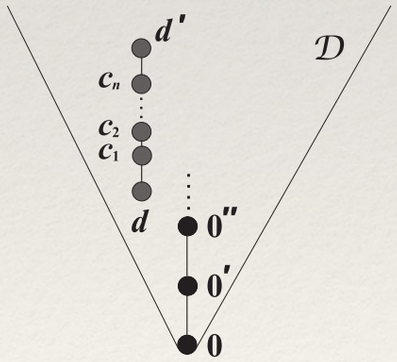
Hence, $(\mathcal{D}, \leq)$ is not a lattice but an upper semi-lattice.

Intermediate *T*-Degrees

Theorem. *For any T-degree* $\boldsymbol{d}$ *and* $n \geqq 1$, *there are pairwise* $\leq$*-incomparable T-degrees* $\boldsymbol{c}_1, \ldots, \boldsymbol{c}_n$ *such that* $\boldsymbol{d} < \boldsymbol{c}_k < \boldsymbol{d}'$, *for* $k = 1, \ldots, n$.

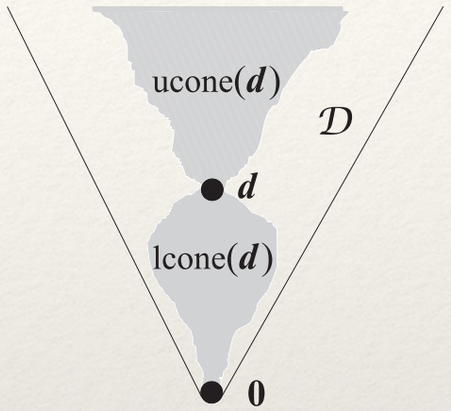


Theorem. *For any T-degree* $\boldsymbol{d}$ *and* $n \geqq 1$, *there are T-degrees* $\boldsymbol{c}_1, \ldots, \boldsymbol{c}_n$ *such that* $\boldsymbol{d} < \boldsymbol{c}_1 < \ldots < \boldsymbol{c}_n < \boldsymbol{d}'$.



Cones

Definition. The upper cone of a T-degree $\boldsymbol{d}$ is the set $\text{ucone}(\boldsymbol{d}) = \{x \in \mathcal{D} \mid \boldsymbol{d} \leq x\}$, and the lower cone of $\boldsymbol{d}$ is $\text{lcone}(\boldsymbol{d}) = \{x \in \mathcal{D} \mid x \leq \boldsymbol{d}\}$.

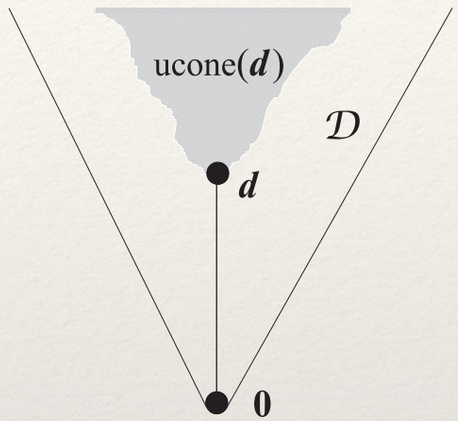


Theorem. *The upper cone* $\text{ucone}(\boldsymbol{d})$ *is uncountable, for any T-degree* $\boldsymbol{d}$.

Theorem. *The lower cone* $\text{lcone}(\boldsymbol{d})$ *is countable, for any T-degree* $\boldsymbol{d}$.

Minimal T-Degrees

Definition. A T-degree $\boldsymbol{d}$ is minimal if $\boldsymbol{d} \neq \boldsymbol{0}$ and there is no T-degree $\boldsymbol{c}$ such that $\boldsymbol{0} < \boldsymbol{c} < \boldsymbol{d}$.



Theorem. *There exists a minimal T-degree; in addition,* $\boldsymbol{d} \leq \boldsymbol{0''}$.

Ch 14. C.E. Degrees and the Priority Method

- C.e. sets are important for they are undecidable and yet, in away, ``close'' to the decidable sets.
- So are important T-degrees that stem from c.e. sets.
- Such T-degrees are called c.e. degrees.

C.E. Turing Degrees

Completeness

Definition. Let $\leq_C$ be an arbitrary reducibility. A c.e. set S is said to be C-complete if $\mathcal{A} \leq_C S$ for every c.e. set $\mathcal{A}$.

Theorem. *The set $\mathcal{K}$ is T-complete.*

C.E. Degrees

Definition. A T-degree is computably enumerable (c.e.) if it contains a c.e. set.

Post's Problem

In 1944, Post asked whether there are any c.e. degrees strictly between 0 and 0′? Note that, informally, this is the question whether there exist *undecidable* problems that are *less difficult* than the *Halting Problem*.

Definition. (Post's Problem) Is there a c.e. degree $\boldsymbol{c}$ such that $\boldsymbol{0 < c < 0'}$?

Post's Program

To solve the problem, Post defined a list of goals that should be attained:

A. Define a *property* of a c.e. set.
B. Prove that any set with this property is *undecidable* and *T-incomplete*.
C. Prove that there *exist* c.e. sets with this property.

However, Post did not succeed in attaining his program.

Post's Problem was solved in 1956 by Friedberg and Muchnik who independently discovered and applied the *Finite-Injury Priority Method*.

The Priority Method and Priority Arguments

In 1956, Friedberg and Muchnik simultaneously and independently upgraded the Post-Kleene's *Method of Finite Extensions* into a subtler one, the *Finite-injury Priority Method*. By applying it, they obtained a positive answer to *Post's Problem*.

The Priority Method in General

Let P be a property sensible of sets. Is there a c.e. set S with the property P?

The *Priority Method* tries to construct S step by step, in an infinite sequence of stages. At each stage i it constructs a finite set S_i , an approximation of S. Each S_i is obtained by adding new elements into $S_{i\text{-}1}$ and/or banning certain elements from entering S_i. So, we want to have $S_{i\text{-}1} \subseteq S_i$ for every i, and finally $\cup_i S_i = S$.

The basic guidelines are:

1. Rewrite P as $R_0 \wedge R_1 \wedge R_2 \wedge \ldots$ so that S will fulfil P *iff* S will fulfil every requirement R_i . An R_i can be fulfilled by carrying out finitely many instructions of the form ``add $x$ into $S_i$'' or ``ban x from S_i''.
2. With different Rs associate different priorities, e.g. $R_{i\text{-}1}$ is of higher priority than R_i.
3. A fulfilled R can be injured, i.e., turned into unfulfilled, only by a *higher-priority* R.
4. Allow any R to be injured only *finitely* many times.

Ch 15. The Arithmetical Hierarchy

- There exists a different view of sets of natural numbers.
- This view allows us to define *arithmetical classes* of sets.
- It also gives rise to the *Arithmetical hierarchy*, the hierarchy of arithmetical classes.
- The hierarchy is closely connected with the *Jump hierarchy*.

Decidability of Relations

Definition. A k-ary relation R on a set S is decidable (or semi-decidable, or undecidable) if the corresponding set $R \subseteq S^k$ is decidable (or semi-decidable, or undecidable).

Example. The relation $R(e,x,s) \equiv$ ``Turing machine T_e halts on x in at most s steps.'' is decidable.

Theorem. *A set $\mathcal{A} \subseteq \mathbb{N}$ is c.e. if and only if $\mathcal{A} = \{x \in \mathbb{N} \mid \exists y \ R(x,y)\}$ for some decidable relation R on $\mathbb{N}$.*

The Arithmetical Hierarchy

Definition. A set $\mathcal{A}$ is an arithmetical set if $\mathcal{A} = \{x \in \mathbb{N} \mid F(x)\}$, such that , for some $n \geqq 0$, the predicate $F(x) = \exists y_1 \forall y_2 \exists y_3 \ldots Q y_n \, R(x, y_1, y_2, y_3, \ldots, y_n)$ or $F(x) = \forall y_1 \exists y_2 \forall y_3 \ldots Q y_n \, R(x, y_1, y_2, y_3, \ldots, y_n)$, and R is a decidable relation.

Definition. The arithmetical classes Σ_n, Π_n, and Δ_n are defined as follows:

Σ_n = class of all sets $\{x \in \mathbb{N} \mid F(x)\}$, where
$F(x) = \exists y_1 \forall y_2 \exists y_3 \ldots Q y_n \, R(x, y_1, y_2, y_3, \ldots, y_n)$ for some decidable relation R;

Π_n = class of all sets $\{x \in \mathbb{N} \mid F(x)\}$, where
$F(x) = \forall y_1 \exists y_2 \forall y_3 \ldots Q y_n \, R(x, y_1, y_2, y_3, \ldots, y_n)$, for some decidable relation R;

Δ_n = class of all sets $\{x \in \mathbb{N} \mid F(x)\}$ that are in $\Sigma_n \cap \Pi_n$.

Theorem. *For any $n \geqq 0$, the following hold:*

$\Sigma_n \subset \Sigma_{n+1}$

$\Sigma_n \subset \Pi_{n+1}$

$\Pi_n \subset \Pi_{n+1}$

$\Pi_n \subset \Sigma_{n+1}$

$\Delta_n \subset \Sigma_n$

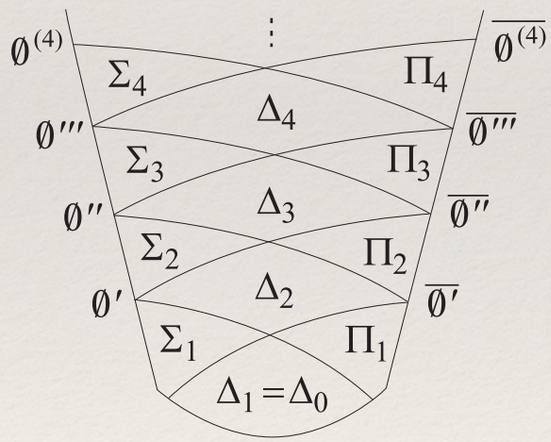
$\Delta_n \subset \Pi_n$

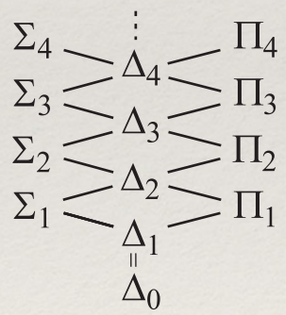
The Link with the Jump Hierarchy

Definition. A set $\mathcal{A}$ is Σ_n-complete if $\mathcal{A} \in \Sigma_n$ and $\mathcal{X} <_m \mathcal{A}$ for every $\mathcal{X} \in \Sigma_n$. Similarly are defined Π_n-complete and Δ_n-complete sets.

Theorem. *Let $\mathcal{A} \subseteq \mathbb{N}$ and $n \geqq 0$. Then*:

$$\emptyset^{(n)} \text{ is } \Sigma_n\text{-complete for } n \geqq 0$$
$$\mathcal{A} \in \Sigma_{n+1} \Leftrightarrow \mathcal{A} \text{ is } \emptyset^{(n)}\text{-c.e.}$$
$$\mathcal{A} \in \Delta_{n+1} \Leftrightarrow \mathcal{A} \leq_T \emptyset^{(n)}$$





Empty slide.