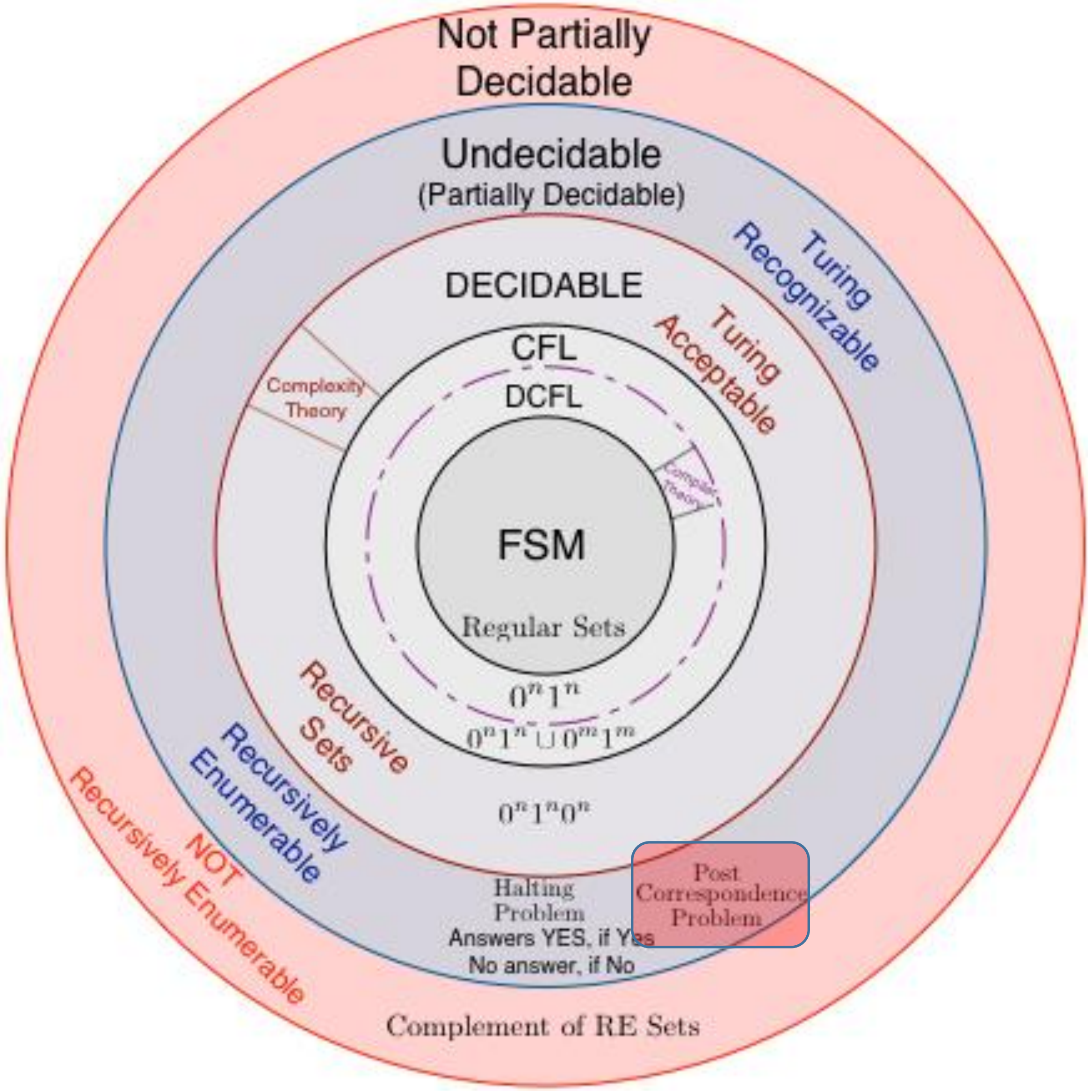
Theory of Computation

Computability Theory - continued

Lecture 13 - Manuel Mazzara



Not Partially Decidable
Undecidable
(Partially Decidable)
Turing Recognizable
DECIDABLE
Turing Acceptable
CFL
DCFL
Complexity Theory
FSM
Regular Sets
$0^n 1^n$
$0^n 1^n \cup 0^m 1^m$
Recursive Sets
Recursively Enumerable
$0^n 1^n 0^n$
NOT Recursively Enumerable
Halting Problem
Answers YES, if Yes
No answer, if No
Post Correspondence Problem
Complement of RE Sets

Post correspondence problem

- The **Post correspondence problem** is an **<u>undecidable problem</u>**
- Like the HP or the decision problem, but simpler to express and often used as an example
- **What does it mean that <u>the algorithm does not exist</u>?**
- **Computability theory is the field of study answering such questions**

The two questions we will explore

Do there exist computing formalisms more powerful than TMs?	Can we always solve problems by means of some mechanical device?
• **Church-Turing thesis**	• **Halting problem and undecidability**

TMs and programming languages

- Given a TM **M** it is possible to build a Pascal (or C or FORTRAN or…) program that simulates **M**
  - The computer runs the program with an **arbitrarily large amount of memory**

- Given any Pascal (or…) program it is possible to build a TM **M** that computes the same function computed by the program

→ **TMs have the same expressive power as high-level programming languages**

Church-Turing thesis (1)

There is no formalism to model any mechanical calculus that is more powerful than the TM or equivalent formalisms

- **It is not a theorem, but a thesis**
- In principle, it should be checked every time anyone comes up with a new computational model
- Indeed it is done, e.g. quantum computing
  - Quantum computing does not break the thesis

Church-Turing thesis: consequence

Any algorithm can be coded in terms of a TM

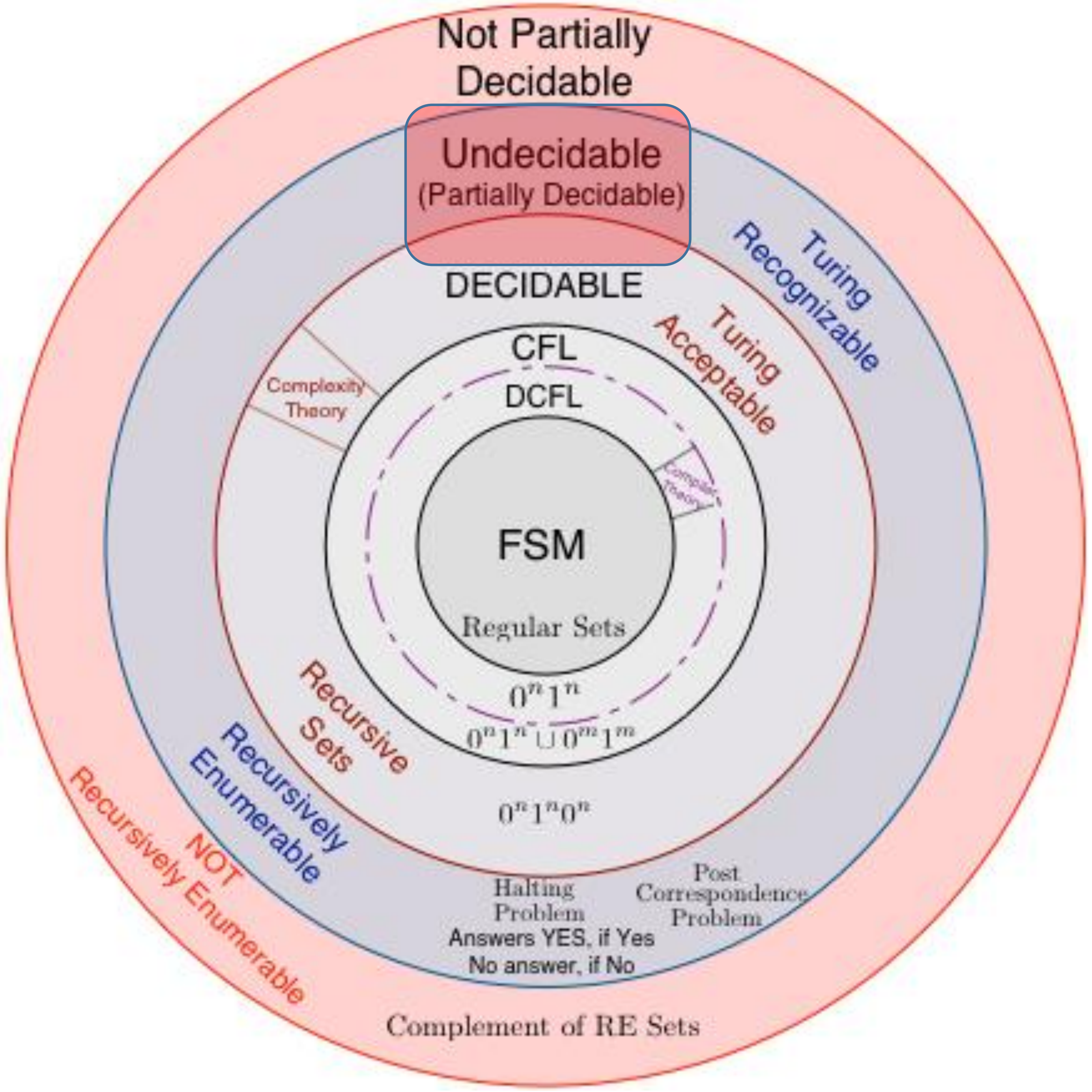
(or an equivalent formalism)

- No algorithm can solve problems that cannot be solved by a TM
- The TM is the most powerful computer that we have and will ever have
- Until a **counterexample** comes out!

Mechanical computation

The notion of TM exactly captures the idea of **mechanical computation**

The problems that can be **solved algorithmically** (“automatically”) are those that can be solved by TMs



Not Partially Decidable
Undecidable
(Partially Decidable)
Turing Recognizable
DECIDABLE
Turing Acceptable
CFL
DCFL
Complexity Theory
FSM
Regular Sets
0^n1^n
$0^n1^n \cup 0^m1^m$
Recursive Sets
$0^n1^n0^n$
Recursively Enumerable
NOT Recursively Enumerable
Halting Problem
Answers YES, if Yes
No answer, if No
Post Correspondence Problem
Complement of RE Sets

TMs and programmable computers

- A TM is a device to solve a given predefined problem
  - A TM can be seen as an **abstract special purpose non-programmable computer**

- Two important questions:
  1. **Can TMs model programmable computers?**
  2. **Can TMs compute all functions from $\mathbb{N}$ to $\mathbb{N}$?**

Are TMs countable?

Algorithmic enumeration

- Given a set **S** we can algorithmically enumerate (**E**) it if we can find **a bijection between S and $\mathbb{N}$**

- **E : S $\leftrightarrow$ $\mathbb{N}$**

- E can be calculated through an algorithm (i.e., a TM by Church-Turing thesis)
- Example: Algorithmic enumeration of $\{a,b\}^*$

$$
\begin{array}{cccccccccccc}
\{ & \varepsilon & a & b & aa & ab & ba & bb & aaa & aab & aba & \ldots\} \\
 & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \\
\{ & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & \ldots\}
\end{array}
$$

Enumeration of TMs

TMs can be algorithmically enumerated

- It is possible to give an effective (computable) **one-to-one pairing between natural numbers and Turing machines**

- This is called an **effective enumeration**

Enumeration of TMs

- There is an algorithm that enumerates TMs **<u>in lexicographical order</u>**

- Some hypotheses (no loss of generality):
  - Single tape TM
  - Unique alphabet A (e.g., |A| = 3, A = {0, 1, _})

Enumerating TMs (1)

- Let us first ignore TMs with a single state
- TMs with two states are:

	0	1	_
q_0	$\perp$	$\perp$	$\perp$
q_1	$\perp$	$\perp$	$\perp$

MT_0

	0	1	_
q_0	$\perp$	$\perp$	$\perp$
q_1	$\perp$	$\langle q_0, 0, S\rangle$	$\perp$

MT_1

……

Enumerating TMs (2)

- How many two-state TMs ?

$\rightarrow$ δ**: Q $\times$ A $\rightarrow$ Q $\times$ A $\times$ {R,L,S}** $\cup$ **{$\perp$ }**

- In general: how many functions f: D $\rightarrow$ R?

$\rightarrow$ $|R|^{|D|}$ ($\forall$ x$\in$D we have |R| choices}

… so with |Q| = 2, |A| = 3, $(2*3*3+1)^{(2*3)} = 19^6$ TMs with 2 states

- Let us sort these TMs: $\{M_0, M_1, \ldots M_{19^6-1}\}$

Enumerating TMs (3)

- Analogously we can sort the $(3*3*3+1)^{(3*3)}$ TMs with 3 states and so on
- We obtain an enumeration **E: {TMs} $\leftrightarrow \mathbb{N}$**
  - **First all the one-state machine, then two-state, three-state...**
- The enumeration E is algorithmic (or *effective*):
  - we can write a program in C (i.e., a TM…) that, given *n*, produces the n-th TM
  - and vice versa

Gödelization

- **E(M) is called the Gödel number of M and E a Gödelization**

- In 1931 Kurt Gödel established a representation between a formal system and the set of natural numbers to prove his famous incompleteness theorem

Yes, TMs are countable

Our two questions are still open!

1. **Can TMs model programmable computers?**

2. **Can TMs compute all functions from $\mathbb{N}$ to $\mathbb{N}$?**

- <u>**The existence of Gödelization helps us in getting an answer**</u>

Programmable computers

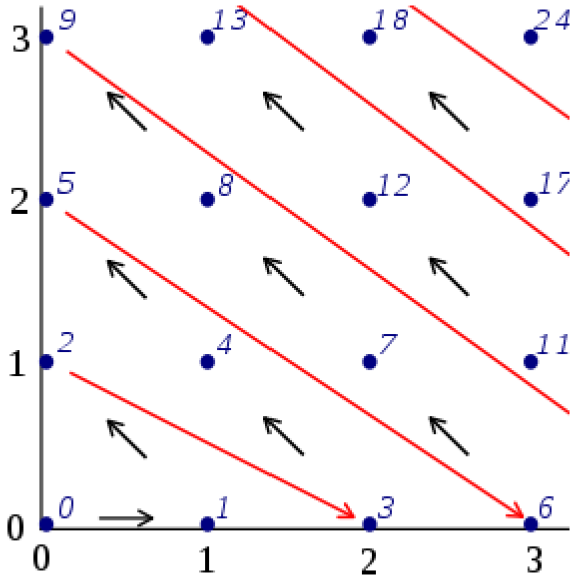
- **Can TMs model programmable computers?**
  - There exists a **Universal Turing Machine (UTM)**

- The UTM computes the function $\mathbf{g(y,x)=f_y(x)}$
  - $f_y(x)$ = function computed by the y-th TM on input *x*
  - UTM has two inputs: the **program** and the **data**

Does it look like a Von Neumann Machine?

UTM exists!

- UTM does not seem to belong to the family $\{M_y\}$ because **f_y is a function of one variable**, while ***g*** **is a function of two variables**
- $\mathbb{N} \times \mathbb{N} \leftrightarrow \mathbb{N}$ (same cardinality, I can enumerate, for example, **rational numbers**)



UTM is a programmable computer

- The TM is a very **abstract and simple model of a computer**

- Analogy:
  - **TM: computer with a single, built-in program**
    - An “ordinary” TM always executes the same algorithm, i.e., it always computes the same function
  - **UTM: computer with program stored in memory**
    - y = program
      x = input to the program

UTM is a model of stored-program computer

TMs and computability

- **Can TMs model programmable computers?**
  - **YES, we have UTM!**

- **Can TMs compute all functions from $\mathbb{N}$ to $\mathbb{N}$?**
  - **NO!**
  - **Why?**

Limits of computability

- Can TMs compute **all** functions from $\mathbb{N}$ to $\mathbb{N}$?
- Does the UTM compute all functions from $\mathbb{N}$ to $\mathbb{N}$?
- **No, there are functions that cannot be computed by any UTM**
  - **There are problems that cannot be solved algorithmically**
  - Let us see the mathematical proof!

Cardinality of functions

- How many are the functions **f: $\mathbb{N} \to \mathbb{N}$** ?
- $\{f: \mathbb{N} \to \mathbb{N}\} \supseteq \{f: \mathbb{N} \to \{0,1\}\} \Rightarrow$

$|\{f: \mathbb{N} \to \mathbb{N}\}| \geq |\{f: \mathbb{N} \to \{0,1\}\}| = |\wp(\mathbb{N})| = 2^{\aleph_0}$

- $\aleph_0$ = cardinality of the set of all the natural numbers
  - **“aleph-zero”**
- **$2^{\aleph_0}$ = cardinality of the set of all real numbers**
- Georg Cantor's **Theory of Transfinite Numbers**

Problems vs Solutions

- The **set of problems**:
- $|\{f: \mathbb{N} \rightarrow \mathbb{N}\}| \geq |\{f: \mathbb{N} \rightarrow \{0,1\}\}| = |\wp(\mathbb{N})| = \mathbf{2^{\aleph_0}}$
- The set of functions computed by TM $\{f_y: \mathbb{N} \rightarrow \mathbb{N}\}$ is by definition **enumerable (Gödelization)**
- $\mathbf{|\{f_y: \mathbb{N} \rightarrow \mathbb{N}\}| = \aleph_0 < 2^{\aleph_0}}$
- "Most" of the problems cannot be solved algorithmically!
- **There are infinitely many more problems than programs!**

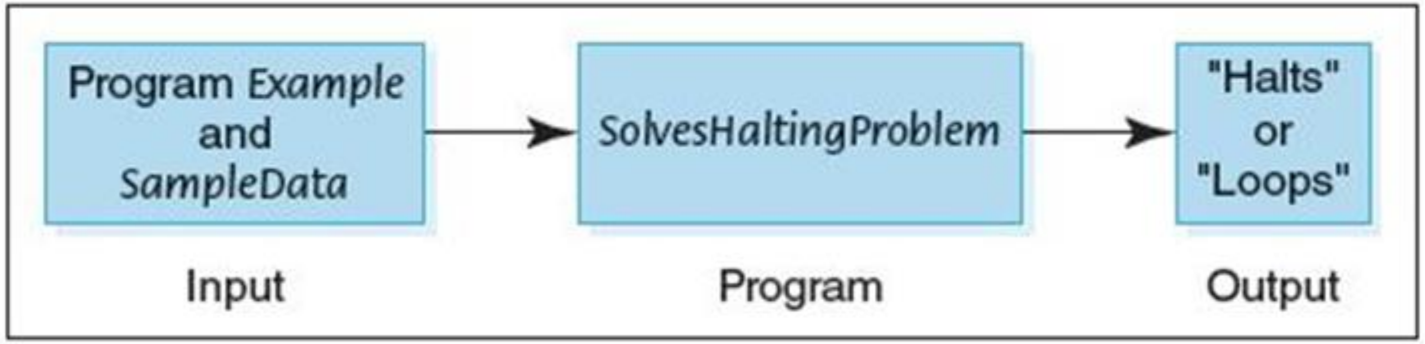
Theoretical Computer Science

Halting Problem

Lecture 13 - Manuel Mazzara

Halting Problem

- Given a **program** and an **input to the program**, determine if the given program **will eventually stop** with this particular input



Remarks

Whether a **particular program** halts on a **particular input** or not is computable in many cases

A test to find this out for **all possible combinations of programs and inputs does not exist**

From the formal and intuitive proof, you will see that **programs that analyse programs can be made to analyse themselves, leading to the impossibility**

Halting Problem, formally (1)

- The "**halting problem**":
  - I build a program
  - I give it some input data
  - I know that in general the program might not terminate its execution *("run into a loop")*

$\rightarrow$ Can I determine **in advance (statically)** if this will occur?

- This problem can be expressed in terms of TMs:
  - Given a function:

    $\mathbf{g(y,x) = 1}$ **if** $\mathbf{f_y(x) \neq \perp}$**,** $\mathbf{g(y,x) = 0}$ **if** $\mathbf{f_y(x) = \perp}$

$\rightarrow$**Is there a TM that computes *g*?**

Halting Problem, formally (2)

There is no TM which can compute the *total* function **g**: $\mathbb{N} \times \mathbb{N} \rightarrow \{0,1\}$ defined as:

$$\mathbf{g(y,x)} = \quad \mathbf{if}\ \mathbf{f_y(x)} \neq \perp \quad \mathbf{then 1}$$
$$\mathbf{else\ 0}$$

- $f_y(x) \neq \perp$ means that M_y comes to halt in a final state on reading x so that $f_y(x)$ is defined

Informally

No TM can decide, for <u>any</u> TM M and input x, whether M halts on input x

- No TM can decide whether any TM will halt in a final state for any input value
- **It is always possible to build a TM that will eventually terminate if and only if it reaches a final state (<u>emulation</u>)**
  - **This is probably the first "naïve" implementation of the HP you may think of (but it works only for positive answers)**

Decidable vs undecidable

- **A TM that computes this g(y,x) does not exist**
  - That's why a computer (which is a program) **cannot warn us that our program will run into an infinite loop on certain data** (while it can easily signal a missing "}")
- Some example:
  - Determining if an arithmetic expression is **well parenthesized is a solvable** (**decidable**) problem (PDA)
  - Determining if any given program will **run into an infinite loop on any given input** is an algorithmically unsolvable (**undecidable**) problem (no TM can do)

(Some)Proof techniques

- Direct proof
  - by axioms, theorems…
- Proof by induction
  - base case, inductive case…
- Constructive proof
  - Provide and example (or counterexample)
- Proof by contradiction
  - *reductio ad absurdum*
- Proof by diagonalization
  - Often used in *computability proofs*

Proof by diagonalization

- The original **diagonalization argument** was used the first time by **Georg Cantor in 1891** to prove that **R** has greater cardinality than $\mathbb{N}$

- It is also used to prove the undecidability of the Halting Problem

$$
\begin{aligned}
s_1 &= 0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\ldots \\
s_2 &= 1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\ldots \\
s_3 &= 0\,1\,0\,1\,0\,1\,0\,1\,0\,1\,0\ldots \\
s_4 &= 1\,0\,1\,0\,1\,0\,1\,0\,1\,0\,1\ldots \\
s_5 &= 1\,1\,0\,1\,0\,1\,1\,0\,1\,0\,1\ldots \\
s_6 &= 0\,0\,1\,1\,0\,1\,1\,0\,1\,1\,0\ldots \\
s_7 &= 1\,0\,0\,0\,1\,0\,0\,0\,1\,0\,0\ldots \\
s_8 &= 0\,0\,1\,1\,0\,0\,1\,1\,0\,0\,1\ldots \\
s_9 &= 1\,1\,0\,0\,1\,1\,0\,0\,1\,1\,0\ldots \\
s_{10} &= 1\,1\,0\,1\,1\,1\,0\,0\,1\,0\,1\ldots \\
s_{11} &= 1\,1\,0\,1\,0\,1\,0\,0\,1\,0\,0\ldots \\
\vdots\;\; & \;\;\;\vdots\,\vdots\,\vdots\,\vdots\,\vdots\,\vdots\,\vdots\,\vdots\,\vdots\,\vdots\,\vdots\;\ddots
\end{aligned}
$$

$$
s = 1\,0\,1\,1\,1\,0\,1\,0\,0\,1\,1\ldots
$$

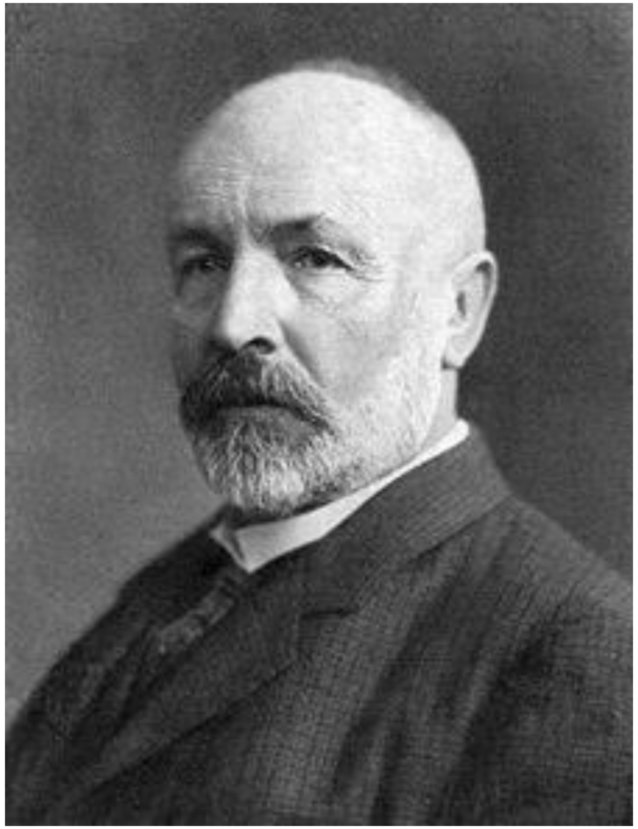
Diagonal argument

- Russel’s paradox, Cantor’s and Gödel’s results and Halting Problem are intimately related
  - **Russell's paradox** uses a diagonalisation argument
  - Turing knew **Cantor's diagonalisation proof**
  - Gödel's incompleteness theorem uses a diagonalisation argument

- Gödel's result is deeply related to the proof of undecidability of the Halting Problem

Russell's paradox (1)

- Discovered by Bertrand Russel in 1901
- It shows that the *naïve set theory* created by **Georg Cantor** leads to a contradiction
  - Georg Cantor states that *any definable collection is a set*
- Let R **be the set of all sets that are not members of themselves**



Georg Cantor 1845-1918

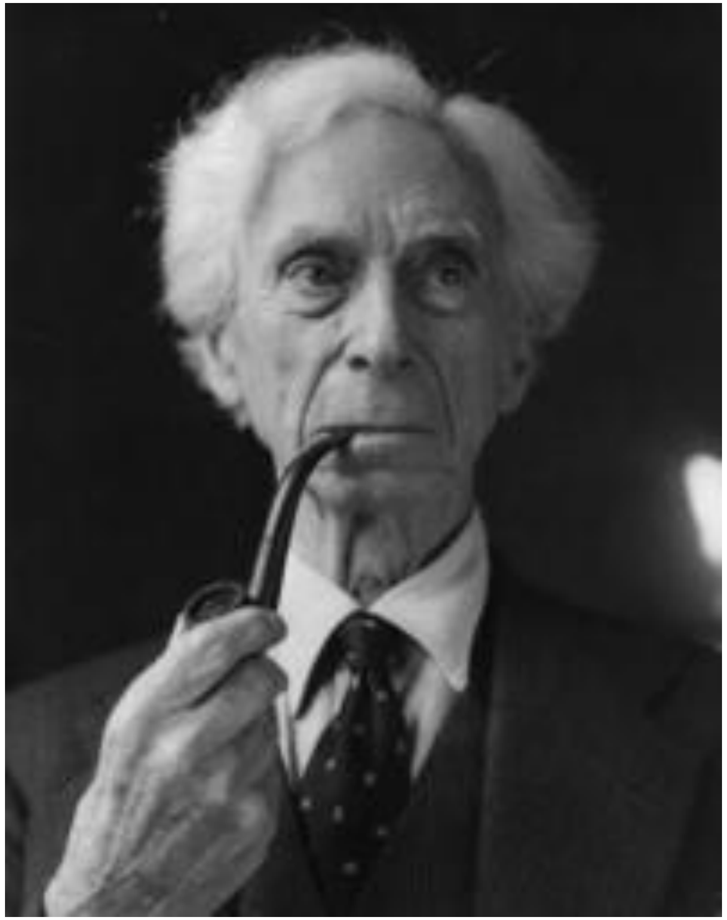
$$R' = \{R', 2, 4, 6, 8, 10, \ldots\}$$

EXAMPLE: R' is member of itself, therefore is not in R

Russell's paradox (2)

- **<u>What about R itself?</u>**
  - If R **<u>qualifies as a member of itself</u>**, it would contradict its own definition as *a set containing all sets that are not members of themselves*
  - If such a set **<u>is not a member of itself</u>**, it would qualify as a member of itself by the same definition

- **Zermelo-Fraenkel set theory (ZF)** deal with Russell's Paradox



Bertrand Russel:1872-1970

$$\textbf{Let } \boldsymbol{R} = \{\boldsymbol{x} \mid \boldsymbol{x} \notin \boldsymbol{x}\}, \textbf{ then } \boldsymbol{R} \in \boldsymbol{R} \iff \boldsymbol{R} \notin \boldsymbol{R}$$

Formally