Lecture 24 – The Origin of Computer Science

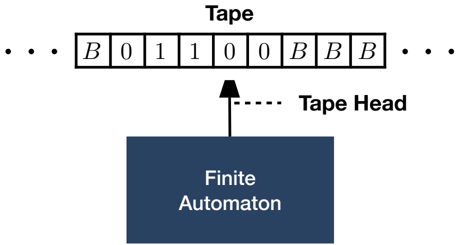
COSE215: Theory of Computation

Jihyeok Park

PLRG

2024 Spring

Recall



- A **Turing machine (TM)** is a finite automaton with a **tape**.
- A language accepted by a TM is **Recursively Enumerable**.
- A standard **TM** is the **most powerful model of computation**.
- Why did **Alan Turing** invent the **TM**?
- Why is TM the **origin of Computer Science**?

Contents

Contents

Gödel's Incompleteness Theorem

PLRG

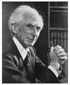
David Hilbert
(1862 – 1943)

I argue that any **mathematical statement** is **True** or **False**!

Russell's Paradox

Really? How about the following statement? **True** or **False**?

$$\text{Let } R = \{x \mid x \notin x\}, \text{ then } R \in R?$$



Bertrand Russell
(1872 – 1970)

David Hilbert
(1862 – 1943)

Okay.. Then, let's **add more axioms** to avoid such paradoxes!
(e.g., **ZFC** - **Z**ermelo–**F**raenkel set theory with Axiom of **C**hoice)

1st Gödel's Incompleteness Theorem (1931)

Unfortunately, I proved that there always exists a statement that is **True** but **Unprovable** under **any set of axioms**.

Kurt Gödel
(1906 – 1978)

Example: Continuum Hypothesis

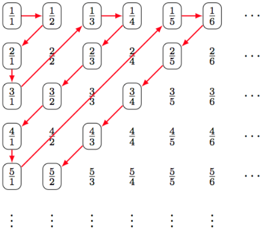
- **Cardinality**: The number of elements in a set.

$$|\{3, 42, 7\}| = 3$$

- A set is **countably infinite** if there is a **bijection** between the set and the set of natural numbers (the cardinality of natural numbers is $\aleph_0$).
  - The set of **non-negative even numbers** is **countably infinite**.

$$\mathbb{N} \underset{f^{-1}}{\overset{f}{\rightleftarrows}} \{n \in \mathbb{N} \mid n \geq 0 \land n \equiv 0 \pmod 2\} \text{ where } f(n) = 2n \text{ and } f^{-1}(n) = \frac{n}{2}$$

  - The set of **rational numbers** is **countably infinite**.



Example: Continuum Hypothesis

- A set of **real numbers** between 0 and 1 is **uncountably infinite** and its cardinality ($\aleph_1 = 2^{\aleph_0}$) is strictly larger than the set of natural numbers ($\aleph_1 > \aleph_0$) because of **Cantor's diagonal argument**:

n	$f(n)$											
1	0	.	**3**	1	4	1	5	9	2	6	5	3	...
2	0	.	3	**7**	3	7	3	7	3	7	3	7	...
3	0	.	1	4	**2**	8	5	7	1	4	2	8	...
4	0	.	7	0	7	**1**	0	6	7	8	1	1	...
5	0	.	3	7	5	0	**0**	0	0	0	0	0	...
⋮	⋮												

- **Continuum Hypothesis**: There is no set whose cardinality is strictly between $\aleph_0$ and $\aleph_1$:

$$\nexists \aleph.\ \aleph_0 < \aleph < \aleph_1$$

- Kurt Gödel and Paul Cohen showed we **CANNOT** either prove or disprove the **Continuum Hypothesis** using the standard axioms of set theory, **ZFC** (**Z**ermelo-**F**raenkel set theory with the **A**xiom of **C**hoice).

Gödel Numbering

- **Gödel Numbering**: Assign a unique number to each symbol and string in a formal language.

Symbol	$\sim$	$\vee$	$\supset$	$\exists$	$=$	0	s	(	)	,	$+$
Number	1	2	3	4	5	6	7	8	9	10	11
Symbol	$\times$	x	y	z	p	q	r	P	Q	R	
Number	12	13	14	15	16	17	18	19	20	21	

- We will use **prime numbers** to encode strings:

$$\texttt{encode}(x_1 \cdots x_n) = \prod_{i=1}^{n} p_i^{x_i}$$

where p_i is the i-th prime number.

- For example, $\texttt{encode}(\texttt{0=0}) = 2^6 \times 3^5 \times 5^6 = 243,000,000$.
- Gödel used this idea to encode **formulas** and **proofs** in **first-order logic**, and then proved his famous **Incompleteness Theorem**.[1]

[1] https://en.wikipedia.org/wiki/Godel's_incompleteness_theorems

Contents

Entscheidungsproblem – Decision Problem

Entscheidungsproblem — "Decision Problem" (1928)

David Hilbert
(1862 – 1943)

I argue another one: there always exists an **algorithm** that takes a statement as an input and **decides** whether it is **True** or **False**!

Disproof using "Turing Machine" (1936)

Inspired by **Gödel's Numbering**, I defined **"Turing Machines"** as **computation** and proved such an algorithm does **not exist**.

Alan Turing
(1912 – 1954)

Disproof using "Lambda Calculus" (1936)

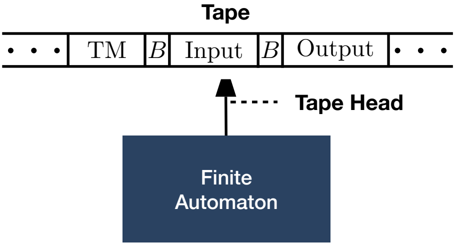
Inspired by **Gödel's Numbering**, I defined **"Lambda Calculus"** as **computation** and proved such an algorithm does **not exist**.

Alonzo Church
(1903 – 1995)

- **Turing Machine** is the origin of **computers**.
- **Lambda Calculus** is the origin of **programming languages**.

Universal Turing Machine (UTM)

- **Alan Turing**'s definition of computation – **Turing Machines (TMs)**.
- Inspired by **Gödel Numbering**, he defined an **encoding** of TMs that can be **enumerated by natural numbers**.
- Then, he defined a **Universal Turing Machine (UTM)** that can simulate any TM with any input:



- **UTM** was **the most important invention in computer science** because it was the first time we can write a **program (software)** instead of building a new **machine (hardware)** to solve a new problem.

Disproof using Turing Machine

- Assume a TM A solves the **Decision Problem**.
- We can build a TM H that solves the **Halting Problem** by using A:

$$\forall \text{ TM } M.\ \forall w \in a^*.\ H(M, w) = \begin{cases} \texttt{halt} & \text{if } A(\text{“}M \text{ halts on } w\text{”}) \\ \texttt{loop} & \text{otherwise} \end{cases}$$

- Consider the following enumeration of TMs:

$H(M_i, w_i)$	w_1	w_2	w_3	$\cdots$
M_1	halt	loop	halt	$\cdots$
M_2	halt	halt	loop	$\cdots$
M_3	loop	halt	halt	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

- Consider the TM F s.t. $\forall i.\ \mathtt{F}(w_i) = \begin{cases} \texttt{loop} & \text{if } H(M_i, w_i) = \texttt{halt} \\ \texttt{halt} & \text{otherwise} \end{cases}$
- Then, F is not in the enumeration (i.e., $F \neq M_i$ for all i). It contradicts the **enumerability of TMs**. So, A **does not exist.**

Lambda Calculus

- **Alonzo Church**'s definition of computation is the **Lambda Calculus (LC)**:

$$\begin{array}{rcll} \Lambda \ni E & ::= & x & \text{(Variable)} \\ & \mid & \lambda x.\ E & \text{(Abstraction)} \\ & \mid & E\ E & \text{(Application)} \end{array}$$

- **Computations** are done by β-**reduction**:

$$(\lambda x.\ E)\ E' \rightarrow E[x \mapsto E']$$

- For example,

$$(\lambda x.\ (\lambda y.\ x\ y))\ z \rightarrow \lambda y.\ z\ y$$

- A **computable function** is a **lambda term**.
- If there is no more possible β-reduction, the term is in **normal form**.

Lambda Calculus – Church Encoding

- However, there is no **data structures** or **control flows** in LC.
- Surprisingly, we can **encode** them – **Church Encoding**:

Boolean Values and Operations

$$\begin{aligned} \text{true} &= \lambda x.\ \lambda y.\ x \\ \text{false} &= \lambda x.\ \lambda y.\ y \\ \text{and} &= \lambda b_1.\ \lambda b_2.\ b_1\ b_2\ \text{false} \\ \text{or} &= \lambda b_1.\ \lambda b_2.\ b_1\ \text{true}\ b_2 \end{aligned}$$

Natural Numbers and Operations

$$\begin{aligned} 0 &= \lambda f.\ \lambda x.\ x \\ 1 &= \lambda f.\ \lambda x.\ f\ x \\ 2 &= \lambda f.\ \lambda x.\ f\ (f\ x) \\ 3 &= \lambda f.\ \lambda x.\ f\ (f\ (f\ x)) \\ \text{plus} &= \lambda n_1.\ \lambda n_2.\ \lambda f.\ \lambda x.\ n_1\ f\ (n_2\ f\ x) \\ \text{times} &= \lambda n_1.\ \lambda n_2.\ \lambda f.\ \lambda x.\ n_1\ (n_2\ f)\ x \\ \text{exp} &= \lambda n_1.\ \lambda n_2.\ n_2\ n_1 \end{aligned}$$

Control Flows

$$\begin{aligned} \text{if} &= \lambda b.\ \lambda e_1.\ \lambda e_2.\ b\ e_1\ e_2 \\ \text{Y} &= \lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x)) \end{aligned}$$

Pairs

$$\begin{aligned} \text{pair} &= \lambda x.\ \lambda y.\ \lambda f.\ f\ x\ y \\ \text{fst} &= \lambda p.\ p\ (\lambda x.\ \lambda y.\ x) \\ \text{snd} &= \lambda p.\ p\ (\lambda x.\ \lambda y.\ y) \end{aligned}$$

Lists

$$\begin{aligned} \text{nil} &= \lambda c.\ \lambda n.\ n \\ \text{cons} &= \lambda h.\ \lambda t.\ \lambda c.\ \lambda n.\ c\ h\ (t\ c\ n) \\ \text{head} &= \lambda l.\ l\ (\lambda h.\ \lambda t.\ h) \\ \text{isnil} &= \lambda l.\ l\ (\lambda h.\ \lambda t.\ \text{false})\ \text{true} \end{aligned}$$

Lambda Calculus – Church Encoding

$$\begin{aligned} 0 &= \lambda f.\ \lambda x.\ x \\ 1 &= \lambda f.\ \lambda x.\ f\ x \\ 2 &= \lambda f.\ \lambda x.\ f\ (f\ x) \\ 3 &= \lambda f.\ \lambda x.\ f\ (f\ (f\ x)) \end{aligned}$$

$$\begin{aligned} \text{plus} &= \lambda n_1.\ \lambda n_2.\ \lambda f.\ \lambda x.\ n_1\ f\ (n_2\ f\ x) \\ \text{times} &= \lambda n_1.\ \lambda n_2.\ \lambda f.\ \lambda x.\ n_1\ (n_2\ f)\ x \\ \text{exp} &= \lambda n_1.\ \lambda n_2.\ n_2\ n_1 \end{aligned}$$

For example, we can compute $1 + 1$ as follows:

$$\begin{aligned} \text{plus}\ 1\ 1 &= (\lambda n_1.\ \lambda n_2.\ \lambda f.\ \lambda x.\ n_1\ f\ (n_2\ f\ x))\ 1\ 1 \\ &\rightarrow \lambda f.\ \lambda x.\ 1\ f\ (1\ f\ x) \\ &= \lambda f.\ \lambda x.\ (\lambda f.\ \lambda x.\ f\ x)\ f\ ((\lambda f.\ \lambda x.\ f\ x)\ f\ x) \\ &\rightarrow \lambda f.\ \lambda x.\ (\lambda f.\ \lambda x.\ f\ x)\ f\ (f\ x) \\ &\rightarrow \lambda f.\ \lambda x.\ f\ (f\ x) \\ &= 2 \end{aligned}$$

The **normal form** (computational result) of (plus 1 1) is 2.

Disproof using Lambda Calculus

PLRG

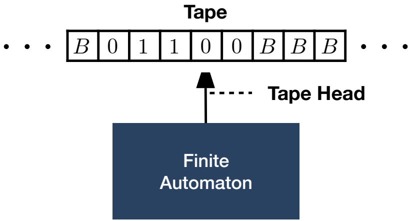
- Church proved that there is **no computable function** that can decide whether two **lambda terms** are **equivalent** or not:

$$\exists\ \text{eq?} \in \Lambda.\ \forall\ E_1, E_2 \in \Lambda.\ (\text{eq?}\ E_1\ \ E_2) \rightarrow \begin{cases} \text{true} & \text{if } E_1 \equiv E_2 \\ \text{false} & \text{otherwise} \end{cases}$$

  where $E_1 \equiv E_2$ means E_1 and E_2 are equivalent, i.e., they have the same **normal form** (computational result).
- For example, (plus 1 1) and 2 are equivalent in LC because they have the same normal form.
- It means that there is no computable function that can **decide** whether a **lambda term** has a given **normal form** or not.
- We skip the proof here.

Contents

Church-Turing Thesis



(Turning Machine)

$$\equiv \quad \Lambda \ni E ::= x \quad \text{(Variable)} \;\mid\; \lambda x.\, E \quad \text{(Abstraction)} \;\mid\; E\ E \quad \text{(Application)}$$

(Lambda Calculus)

- **LC** has the same computational power as **TMs**. (**Turing Complete**)
- **Church-Turing Thesis**:

  Any **real-world computation** *can be translated into an equivalent computation involving a* **Turing machine** *or can be done using* **lambda calculus**.

Summary

1. Gödel's Incompleteness Theorem
    - Example: Continuum Hypothesis
    - Gödel Numbering
2. Entscheidungsproblem – Decision Problem
    - Disproof using Turing Machine
    - Disproof using Lambda Calculus
3. Church-Turing Thesis

Next Lecture

- Undecidability

Jihyeok Park
jihyeok_park@korea.ac.kr
https://plrg.korea.ac.kr