From Abstraction to Computation: Understanding Lambda Calculus

Pratham Gupta Gavish Bansal
Sehaj Ganjoo Krishna Agarwal

Indian Institute of Science, Bengaluru

April 4, 2025

Outline

Outline I

Historical Background

Hilbert's program:

- Formalize all of mathematics in a consistent axiomatic system. Such that:
    - **Consistency:** No contradictions can be derived from the axioms.
    - **Completeness:** All true statements can be derived from the axioms.
    - **Decidability:** There exists a mechanical procedure to determine the truth of any statement.

Leibniz's Ideal:

- A "universal language" to express all possible problems.
- A decision method to solve all problems in this language.

By the early 20th century, set theory and first-order logic (Frege, Russell, Zermelo) fulfilled point (1).
However, point (2) remained open—this became the **Entscheidungsproblem** ("decision problem"):

Can all problems be solved mechanically?

Historical Background

Alonzo Church and Alan Turing independently proved that no general algorithm can decide the truth of all mathematical statements.
In order to do so, they had to formalize "computability.

- Church (1936): Introduced lambda calculus as a formal model of computation
- Turing (1936/37): Introduced Turing machines as an alternative model.
- Turing (1937): Proved both models are equivalent—defining the same class of computable functions
- Class of functions computable by these is called **Recursive Functions**.

Motivation

- Addressing Russell's paradox in set theory.
- Establishing a formal system for computability.
- Laying the groundwork for functional programming languages.

Church-Turing Thesis

Any natural / reasonable notion of computation realizable in the physical world can be simulated by a TM (or equivalently, by lambda calculus)

Outline I

Basic Concepts

- **Variables:** $x, y, z, \ldots$
- **Abstraction:** $\lambda x.M$ (function definition)
- **Application:** (MN) (function application)

Formal Definition of λ-Terms

Definition

The set of λ-terms is defined inductively:

1. Any variable x is a λ-term.
2. If M and N are λ-terms, then (MN) is a λ-term called an **application**
3. If M is a λ-term and x is a variable, then $\lambda x.M$ is a λ-term called a λ-abstraction

Formal Definition of λ-Terms

Some Valid Lambda Expressions:

- x
- $\lambda x.\lambda x.\, x$
- xy
- $x\lambda y.(x(yy))$
- $\lambda x.\lambda y.\lambda z.\,(x\ (y\ z))$
- $\lambda\lambda x.x$ (invalid)

Notation Conventions

- Left-association of application:

$$(((F\ M_1)M_2)\dots M_n) \quad \text{is written as} \quad FM_1\dots M_n.$$

e.g. $wxyz$ is $((wx)y)z$.

- Right-association of abstractions:

$$\lambda x_1.\lambda x_2.\cdots\lambda x_n.M \quad \text{is written as} \quad \lambda x_1\cdots x_n.M.$$

e.g. $\lambda x.xy$ is $\lambda x.(xy)$, and $\lambda x.\lambda x.x$ is $\lambda xx.x$.

Semantics

λ**x.M** defines a function, where:

- x is the formal parameter of the function.
- M is the body of the function.
- $M \to M_1 M_2$, function application, similar to calling function M_1 and setting its formal parameter to M_2.

Example:

- $\lambda x. (x + 1)$ defines a function that adds 1 to its argument.
- $(\lambda x. (x + 1))\ 2$ evaluates as $(2 + 1)$, which is 3.

Q. How can $+$ function be defined if abstractions only accept 1 parameter?

Currying

Definition

In lambda calculus, a abstraction takes only one argument.
Currying(named after Haskell Curry) is the process of transforming a function that takes multiple arguments into a sequence of functions, each taking a single argument.

Example:

$$\lambda x.\lambda y.(x + y)$$

$$(\lambda x.\lambda y.(x + y))\ 10\ 20 \to (\lambda y.(10 + y))\ 20 \to (10 + 20) \to 30$$

Currying

Consider the function: $f : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$
If we fix the first argument, we get a function $F_x : \mathbb{N} \to \mathbb{N}$

$$F_x(y) = f(x, y) \forall y \in \mathbb{N}$$

So we can write $F_x = \lambda y.f(x, y)$ and then the function $x \mapsto F_x$ can be written as:

$$F = \lambda x.F_x = \lambda x.\lambda y.f(x, y)$$

Hence we obtain

$$(F\ M)N \to_\beta F_M N \to_\beta f(M, N)$$

Currying

Hence we have ability to transform a function that takes multiple arguments into a sequence of functions, each taking a single argument. As a notational convention, we can write:

$$\begin{aligned} F &= \lambda x_1.\lambda x_2.\cdots\lambda x_n.f(x_1, x_2, \ldots, x_n) \\ &= \lambda x_1 x_2 \cdots x_n.f(x_1, x_2, \ldots, x_n) \end{aligned}$$

Free Variables

Intuitively

Free Variables(FV): Variables that are not bound by any abstraction.

- Is x free in $\lambda x.x$? (No)
- Is x free in $(\lambda x.xy)x$? (Yes)
  $(\lambda x.xy)x \to (xy)$

Definition

For any λ-term M, the set of free variables is denoted as $\mathrm{FV}(M)$ and is defined as:

- If $M = x$ then: $\mathrm{FV}(\mathrm{x}) = \{\mathrm{x}\}$
- If $M = (M_1 M_2)$, then:$\mathrm{FV}(M) = \mathrm{FV}(M_1) \cup \mathrm{FV}(M_2)$,
- If $M = \lambda x.M_1$, then:$\mathrm{FV}(M) = \mathrm{FV}(M_1) \setminus \{x\}$

Bound Variables

Intuitively
Bound Variables(BV): Variables that are not free.
Bound variables are declared within a λ-abstraction.

- $(\lambda x.\lambda y.x y)(\lambda z.\mathbf{x}z)$

Definition

For any λ-term M, the set of bound variables is denoted as $\mathrm{BV}(M)$ and is defined as:

- If $M = x$ then: $\mathrm{BV}(x) = \emptyset$
- If $M = (M_1 M_2)$, then:$\mathrm{BV}(M) = \mathrm{BV}(M_1) \cup \mathrm{BV}(M_2)$,
- If $M = \lambda x.M_1$, then:$\mathrm{BV}(M) = \mathrm{BV}(M_1) \cup \{x\}$

Combinators

Definition

A λ-term M is closed or a **Combinator** if it has no free variables, i.e., $\text{FV}(M) = \emptyset$.

$$\text{For } M_1 = \lambda x.(xy), \quad \text{FV}(M_1) = \{y\}, \quad \text{BV}(M_1) = \{x\}.$$

$$\text{For } M_2 = \lambda x.(\lambda y.(x)), \quad \text{FV}(M_2) = \emptyset, \quad \text{BV}(M_2) = \{x, y\}.$$

Note that:

- M_2 has no free variables and is a combinator.
- If we rename a bound variable in a term, It has no effect on the behavior of the term.

Outline I

α Equivalence

Q. What does it mean for two functions to be equivalent?

- Is $\lambda x.\mathtt{xy} \equiv \lambda y.\mathtt{yx}$?

- α-equivalence is when two functions vary only by the names of bound variables.
- $M_1 \equiv_\alpha M_2$ if M_1 can be obtained from M_2 by renaming bound variables.

Substitution

Our goal: Reduce expressions by replacing variables with terms.
e.g $(\lambda x.x)\ 2 \to 2$

Solution : Substitution Operation

- Notation: $M[x := N]$.
- Replacing all free occurrences of a variable x in M by a term N.

Example:

- $(x + 1)[x := 2] = (2{+}1)$
- $(\lambda x.(x + 1))[x := \lambda y.z] = (\lambda x.(x + 1))$ (no free x)
- $(\lambda x.(xt))[t := \lambda y.z] = (\lambda x.(x\lambda y.z))$

Substitution

Definition

A substitution $\varphi = [x_1 := N_1, \ldots, x_n := N_n]$ is a finite set of pairs where each x_i is a variable and N_i is a λ-term.

Definition

Given a substitution φ and any variable x_i, φ_{-x_i} is a new substitution obtained by removing the pair (x_i, N_i) from φ.

Substitution Rules

Given any λ-term M and a substitution φ, the result of applying φ to M is denoted by $M[\varphi]$ and is defined as follows:

1. If M is a variable x, then:

$$M[\varphi] = \begin{cases} N_i & \text{if } x = x_i \text{ for some } i, \\ x & \text{otherwise.} \end{cases}$$

2. If M is of the form $(M\ N)$, then:

$$M[\varphi] = (M[\varphi]\ N[\varphi])$$

3. If M is of the form $\lambda y.M_1$, then:

$$M[\varphi] = \begin{cases} \lambda y.M_1[\varphi] & \text{if } y \notin \{x_i\}, \\ \lambda y.M_1[\varphi_{-y}] & \text{if } y = x_i \text{ for some } i. \end{cases}$$

Capturing

Example

$(\lambda x.yx)[y := \lambda z.xz]$

- Result ? : $(\lambda x.(\lambda z.xz)x)$
- $x \in FV(\lambda z.xz)$
- $x \in BV(\lambda x.(\lambda z.xz)x)$

This is called Variable Capture.

Capturing occurs when a free variable unintentionally becomes bound due to renaming or substitution, altering the meaning of an expression.
Problem: Function's behavior is altered.

α-Conversion

Solution: Use α-conversion to rename bound variables before substitution.

$$(\lambda x.\, yx)[y := \lambda z.\, xz]$$

Direct substitution causes variable capture, so we first apply α-conversion:

$$\lambda x.\, yx \to_\alpha \lambda w.\, yw$$

Now perform the substitution:

$$(\lambda w.\, yw)[y := \lambda z.\, xz] = \lambda w.\, (\lambda z.\, xz)w$$

Definition of α-Conversion

Immediate Alpha-Conversion ($\rightarrow_\alpha$) allows renaming bound variables in lambda terms while preserving meaning.

Rules:

- $\lambda x.M \rightarrow_\alpha \lambda y.M[x := y]$, if $y \notin FV(M) \cup BV(M)$.
- If $M \rightarrow_\alpha N$, then $MQ \rightarrow_\alpha NQ$ and $PM \rightarrow_\alpha PN$.
- If $M \rightarrow_\alpha N$, then $\lambda x.M \rightarrow_\alpha \lambda x.N$.

Alpha-Equivalence: The reflexive, transitive closure of $\rightarrow_\alpha$ is denoted as $\equiv_\alpha$, meaning two terms are identical up to renaming.

Outline I

Beta-Reduction

Definition

The relation $\to_\beta$ called immediate β-reduction is the smallest relation satisying the property for all λ-terms M, N, P, Q

$$(\lambda x.M)N \to_\beta M[x := N] \text{ where M is safe for substitution } [x := N].$$

$$\text{if } M \to_\beta N \text{ then } MQ \to_\beta NQ \text{ and } PM \to_\beta PN$$

$$\text{if } M \to_\beta N \text{ then } \lambda x.M \to_\beta \lambda x.N$$

- Transitive closure of $\to_\beta$ is denoted as $\to_\beta^+$.
- reflexive and transitive closure of $\to_\beta$ is denoted as $\to_\beta^*$.
- β-**conversion** denoted by $\overset{*}{\leftrightarrow}$ is smallest equivalence relation such that:

$$\overset{*}{\leftrightarrow} = (\to_\beta \cup \to_\beta^{-1})$$

Examples of β-Reduction

Example

$$(\lambda x.x)y \to_\beta (x)[x := y] \to_\beta y$$

$$(\lambda xy.y)uv = (\lambda x.(\lambda y.y)u)v \to_\beta ((\lambda y.y)[x := u])v = (\lambda y.y)v \to_\beta v$$

example

Let $\omega = \lambda x.(xx)$ then

$$\Omega = \omega\omega = (\lambda x.(xx))(\lambda x.(xx)) \to_\beta (\lambda x.(xx))[x := \lambda x.(xx)] = \omega\omega = \Omega$$

This example shows that β-reduction may be infinite. This is what gives lambda calculus its power.

Example of β-Reduction

Example

β-reduction can have growing terms also:

$$(\lambda x.xxx)(\lambda x.xxx) \to_\beta (\lambda x.xxx)[x := \lambda x.xxx] = (\lambda x.xxx)(\lambda x.xxx)(\lambda x.xxx)$$

$$\to_\beta (\lambda x.xxx)(\lambda x.xxx)(\lambda x.xxx)(\lambda x.xxx)$$

$$\to_\beta (\lambda x.xxx)(\lambda x.xxx)(\lambda x.xxx)(\lambda x.xxx)\cdots$$

Church–Rosser Theorem

Theorem

The following properties hold for the λ-calculus:

- **Confluence:** *If $M \to_\beta^* N_1$ and $M \to_\beta^* N_2$, then there exists N_3 such that $N_1 \to_\beta^* N_3$ and $N_2 \to_\beta^* N_3$.*
- **Church-Rosser Property:** *for any two λ-terms M and N, if $M \xleftrightarrow{*}_\beta N$, then there exists a λ-term P such that:*

$$M \to_\beta^* P \quad \text{and} \quad N \to_\beta^* P$$

Note

For proof of equivivalence of two, please refer to the Appendix section.

Church–Rosser Theorem: Visual Representation

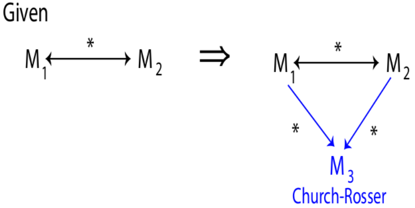

Figure: Church–Rosser Property

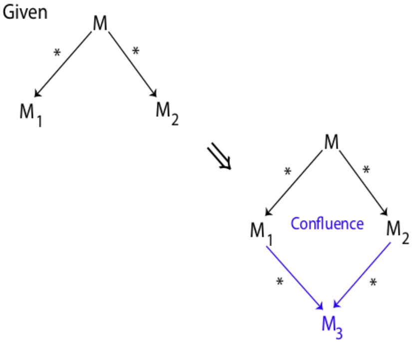

Figure: Confluence Property

Outline I

Combinators I, K, and S

Let us define the following combinators:

- $\mathbf{I} = \lambda x.x$ (identity function)
- $\mathbf{T} = \mathbf{K} = \lambda xy.x$ (True)
- $\mathbf{F} = \mathbf{K}_* = \lambda xy.y$ (False)

We can see some interesting properties of these combinators:

- $\mathbf{I}M \to_\beta M$
- $\mathbf{K}MN \to_\beta M$
- $\mathbf{K}_*MN \to_\beta N$

Conditional

Let us define the conditional operator:

$$\text{if then else} = \lambda bxy.bxy$$

then for all $\lambda - terms$ we have:

- if $\mathbf{T}\ M\ N \to_\beta M$
- if $\mathbf{F}\ M\ N \to_\beta N$

Example:

$$\begin{aligned}
\text{if } \mathbf{T} \text{ then } P \text{ else } Q &= (\text{if then else})\mathbf{T}PQ \\
&= (\lambda bxy.bxy)\mathbf{T}PQ \\
&\to_\beta ((\lambda xy.bxy)[b := \mathbf{T}])PQ = (\lambda xy.\mathbf{T}xy)PQ \\
&\to_\beta ((\lambda y.\mathbf{T}xy)[x := P])Q = (\lambda y.\mathbf{T}Py)Q \\
&\to_\beta (\mathbf{T}Py)[y := Q] = \mathbf{T}PQ \\
&= \mathbf{K}PQ \xrightarrow{+}_\beta P,
\end{aligned}$$

Boolean Operations

- **And:** And $b_1 b_2 =$ if b_1 then (if b_2 then **T** else **F**) else **F**
- **Or:** Or $b_1 b_2 =$ if b_1 then **T** else (if b_2 then **T** else **F**)
- **Not:** Not $b =$ if b then **F** else **T**

Constructing Ordered Pairs

For any two λ-terms M and N, consider the combinators π_1 and π_2 defined as:

$$\begin{aligned}
\langle M, N \rangle &= \lambda z.\, z\, M\, N \\
&= \lambda z.\, \text{if } z \text{ then } M \text{ else } N \\
\pi_1 &= \lambda z.\, zK \\
\pi_2 &= \lambda z.\, zK_*
\end{aligned}$$

Then, we have the following β-reductions:

$$\pi_1 \langle M, N \rangle \xrightarrow{\beta} M$$

$$\pi_2 \langle M, N \rangle \xrightarrow{\beta} N$$

$$\langle M, N \rangle T \xrightarrow{\beta} M$$

$$\langle M, N \rangle F \xrightarrow{\beta} N$$

Proof

Beta Reduction of $\pi_1\langle M, N\rangle$

$$\pi_1\langle M, N\rangle \xrightarrow{\beta} M$$

Proof: We have:

$$\begin{aligned}
\pi_1\langle M, N\rangle &= (\lambda z.\, zK)(\lambda z.\, zMN) \\
&\xrightarrow{\beta} (zK)[z := \lambda z.\, zMN] \\
&= (\lambda z.\, zMN)K \\
&\xrightarrow{\beta} (zMN)[z := K] \\
&= KMN \xrightarrow{\beta} M
\end{aligned}$$

Proof

Beta Reduction of $\pi_2\langle M, N\rangle$

$$\pi_2\langle M, N\rangle \xrightarrow{\beta} N$$

Proof: We have:

$$\begin{aligned}
\pi_2\langle M, N\rangle &= (\lambda z.\, zK_*)(\lambda z.\, zMN) \\
&\xrightarrow{\beta} (zK_*)[z := \lambda z.\, zMN] \\
&= (\lambda z.\, zMN)K_* \\
&\xrightarrow{\beta} (zMN)[z := K_*] \\
&= K_*MN \xrightarrow{\beta} N
\end{aligned}$$

Proof

Beta Reductions of Ordered Pair Application

$$\langle M, N\rangle T \xrightarrow{\beta} M, \quad \langle M, N\rangle F \xrightarrow{\beta} N$$

Proof: We have:

$$\begin{aligned}
\langle M, N\rangle T &= (\lambda z.\, zMN)T \\
&\xrightarrow{\beta} TMN \\
&\xrightarrow{\beta} M, \quad (\text{since } T = K)
\end{aligned}$$

$$\begin{aligned}
\langle M, N\rangle F &= (\lambda z.\, zMN)F \\
&\xrightarrow{\beta} FMN \\
&\xrightarrow{\beta} N, \quad (\text{since } F = K_*)
\end{aligned}$$

Outline I

Church Numerals

Definition

Church Numerals $\mathbf{c_0}, \mathbf{c_1}, \mathbf{c_2}, \ldots$ are defined by:

$$\mathbf{c_n} = \lambda fx.f^n x$$

Observe:

- $\mathbf{c_0} = \lambda fx.x = \mathbf{K}_*$
- $\mathbf{c_1} = \lambda fx.fx$
- $\mathbf{c_n} Fz = (\mathbf{c_n} F)z = ((\lambda fx.f^n(x))F)z \to_\beta^+ F^n(z)$

Iteration

Definition

The **Iteration** combinator **Iter** is defined as:

$$\text{Iter} = \lambda nfx.nfx$$

Notice that:

- Iter combinator is same as the if then else combinator. This means that if we pass a boolean to the Iter combinator then it will behave like the if then else combinator.
- $\textbf{Iter}\ \mathbf{c_n} fx = \lambda fx.f^n x$
  proof:
  $$\textbf{Iter}\ \mathbf{c_n} fx \rightarrow \mathbf{c_n} fx = (\lambda fx.f^n x)fx = f^n x$$

Arithmetic Operations

- **Successor:**

$$\mathbf{Succ_c} = \lambda nfx.f(nfx)$$

- **IsZero:**

$$\mathbf{IsZero_c} = \lambda x.x(\mathbf{KF})\mathbf{T}$$

- **Addition:**

$$\mathbf{Add} = \lambda mn.\mathbf{Iter}\ m\ \mathbf{Succ_c}\ n$$

- **Multiplication:**

$$\mathbf{Mult} = \lambda mn.\mathbf{Iter}\ m\ \mathbf{Add_c}\ n$$

- **Exponentiation:**

$$\mathbf{Exp} = \lambda mn.\mathbf{Iter}\ n\ \mathbf{Mult_c}\ m$$

Arithmetic Operations

Example: Zero Check

$$\mathbf{IsZero_c} = \lambda n.n(\lambda x.\mathbf{F})\mathbf{T}$$

Proof:

- For $n = \mathbf{c_0}$:

$$\mathbf{IsZero_c c_0} = (\lambda n.n(\lambda x.\mathbf{F})\mathbf{T})\mathbf{c_0}$$

Substituting $\mathbf{c_0} = \lambda fx.x$:

$$\rightarrow_\beta \mathbf{c_0}(\lambda x.\mathbf{F})\mathbf{T} = (\lambda fx.x)(\lambda x.\mathbf{F})\mathbf{T}$$

$$\rightarrow_\beta (\lambda x.x)\mathbf{T} = \mathbf{T}$$

Hence, $\mathbf{IsZero_c c_0} \rightarrow_\beta \mathbf{T}$.

Arithmetic Operations

Evaluating $\text{IsZero}_c \mathbf{c_1}$

- For $n = \mathbf{c_1}$, we have:

$$\text{IsZero}_c \mathbf{c_1} = (\lambda n.n(\lambda x.\mathbf{F})\mathbf{T})\mathbf{c_1}$$

- Substituting $\mathbf{c_1} = \lambda fx.fx$:

$$\rightarrow_\beta \mathbf{c_1}(\lambda x.\mathbf{F})\mathbf{T}$$

$$= (\lambda fx.fx)(\lambda x.\mathbf{F})\mathbf{T}$$

- Beta reduction:

$$\rightarrow_\beta (\lambda x.\mathbf{F})\mathbf{T}$$

$$\rightarrow_\beta \mathbf{F}$$

- Hence, we conclude:

$$\text{IsZero}_c \mathbf{c_1} \rightarrow_\beta \mathbf{F}$$

Arithmetic Operations

Example: Successor

$$\mathbf{Succ_c} = \lambda nfx.f(nfx)$$

Proof:

- For $n = \mathbf{c_k}$:

$$\mathbf{Succ_c c_k} = (\lambda nfx.f(nfx))\mathbf{c_k}$$

Substituting $n = \mathbf{c_k} = \lambda fx.f^k(x)$:

$$\rightarrow_\beta^* (\lambda fx.f(\mathbf{c_k} fx)) = (\lambda fx.f(\lambda fx.f^k(x))fx)$$

$$\rightarrow_\beta^* (\lambda fx.f(f^k(x))) = (\lambda fx.f^{k+1}(x)) = \mathbf{c_{k+1}}$$

The Predecessor Function

$$\begin{aligned} \text{Pred}(0) &= 0, \\ \text{Pred}(n+1) &= n. \end{aligned}$$

- More challenging to define.
- Kleeneś solution in his famous 1936 paper, uses pairs:

$$\text{Pred}_K = \lambda n.\pi_2(\text{Iter } n\ \lambda z.\langle \text{Succ}(\pi_1 z), \pi_1 z\rangle\ \langle c_0, c_0\rangle).$$

The Predecessor Function

Why Kleene's Predecessor Function Works?

We have:

$$(\lambda z.\langle \mathrm{Succ}(\pi_1 z), \pi_1 z\rangle)^0 \langle c_0, c_0\rangle \xrightarrow{\beta} \langle c_0, c_0\rangle.$$

We claim that:

$$(\lambda z.\langle \mathrm{Succ}(\pi_1 z), \pi_1 z\rangle)^{n+1} \langle c_0, c_0\rangle \xrightarrow{\beta} \langle c_{n+1}, c_n\rangle.$$

The Predecessor Function

$$Claim : (\lambda z.\langle \mathrm{Succ}(\pi_1 z), \pi_1 z\rangle)^{n+1}\langle c_0, c_0\rangle \xrightarrow{\beta} \langle c_{n+1}, c_n\rangle.$$

Proof by Induction on n:

Base Case: $n = 0$

$$\begin{aligned}(\lambda z.\langle \mathrm{Succ}(\pi_1 z), \pi_1 z\rangle)\langle c_0, c_0\rangle &\xrightarrow{\beta} \langle \mathrm{Succ}(\pi_1\langle c_0, c_0\rangle), \pi_1\langle c_0, c_0\rangle\rangle \\ &\xrightarrow{\beta} \langle \mathrm{Succ}(c_0), c_0\rangle \\ &\xrightarrow{\beta} \langle c_1, c_0\rangle.\end{aligned}$$

The Predecessor Function

$$\textit{Claim} : (\lambda z.\langle \mathrm{Succ}(\pi_1 z), \pi_1 z\rangle)^{n+1}\langle c_0, c_0\rangle \xrightarrow{\beta} \langle c_{n+1}, c_n\rangle.$$

Inductive Step: Assume true for n, show for $n+1$

$$\begin{aligned}
&(\lambda z.\langle \mathrm{Succ}(\pi_1 z), \pi_1 z\rangle)^{n+2}\langle c_0, c_0\rangle \\
&= (\lambda z.\langle \mathrm{Succ}(\pi_1 z), \pi_1 z\rangle)((\lambda z.\langle \mathrm{Succ}(\pi_1 z), \pi_1 z\rangle)^{n+1}\langle c_0, c_0\rangle) \\
&\xrightarrow{\beta} (\lambda z.\langle \mathrm{Succ}(\pi_1 z), \pi_1 z\rangle)\langle c_{n+1}, c_n\rangle \\
&\xrightarrow{\beta} \langle c_{n+2}, c_{n+1}\rangle.
\end{aligned}$$

Predecessor Functions

- **Kleene's predecessor function:**

$$\mathrm{Pred}_K = \lambda n.\pi_2(\mathrm{Iter}\ n\ \lambda z.\langle \mathrm{Succ}(\pi_1 z), \pi_1 z\rangle\ \langle c_0, c_0\rangle).$$

- **Alternative definition of predecessor:**

$$\mathrm{Pred}_c = \lambda xyz.\ x(\lambda pq.\ q(py))(Kz)I.$$

Outline I

The Factorial: n!

- $fact(0) = 1$
- $fact(n) = n * fact(n-1)$

Factorial Function

```
int fact(int n) {
 if (n == 0) return 1;
 else return n * fact(n - 1);
}
```

fact=(λ n. if (IsZero n) then 1 else Mult N (fact (Pred n)))

Fixed-Point Combinators

- Fixed-point combinators allow us to define recursive functions in the lambda calculus, giving us a way to express recursion without explicit self-reference. We want a ability to be able to do this:

$$func(x) := \text{If [base condition] then base else func(y)}$$

- Our issue is that functions in lambda calculus are not named hence we cannot have the function be named and refrence called itself.
- **Fixed-point combinators** solves the problem by taking the function as an argument and returning a fixed point of that function.

Turing Θ-Combinator

$$\mathbf{\Theta} := (\lambda xy.y\,(xxy))\,(\lambda xy.y\,(xxy))$$

Turing's Θ-Combinator

- We define *Turing* **Θ***-combinator* as:

$$\mathbf{\Theta} := (\lambda xy.y\,(xxy))\,(\lambda xy.y\,(xxy))$$

- Now, for any λ-term F, we have:

$$\mathbf{\Theta}F \xrightarrow{+}_{\beta} F(\mathbf{\Theta}F)$$

Proof

Writing $A = (\lambda xy.y(xxy))$. Thus, $\mathbf{\Theta} = AA$. Now,

$$\begin{aligned}\mathbf{\Theta}F &= AAF = ((\lambda xy.y(xxy))A)F \\ &\to_{\beta} (\lambda y.y(AAy))F \\ &\to_{\beta} F(AAF) \\ &= F(\mathbf{\Theta}F)\end{aligned}$$

The conbinator takes our function as outputs equivalent of $\lambda u.(F(F(\ldots(F(u)))))$

Defining Recursive Functions

- To define a recursive function G such that

$$GX \to_\beta M(X, G)$$

- Let $F = \lambda gx.M(x, g)$ and define $G = \Theta F$.

Example: Factorial Function

Define:

$$F = \lambda g\ n.\, \text{if } (\text{IsZero } n) \text{ then } c_1 \text{ else Mult } n\ \big(g\ (\text{Pred } n)\big)$$

Then $G = \Theta F$ represents the factorial function.

Proof: Factorial Function

- Let's prove that G represents the factorial function by induction.
- Base case: $n = 0$

$$Gc_0 \to_\beta F(G)c_0 \to_\beta c_1$$

- Inductive step: Assume true for n, show for $n + 1$
- Inductive hypothesis:

$$Gc_n \to_\beta c_{n!}$$

$$Gc_{n+1} \to_\beta F(G)c_{n+1} \to_\beta \mathrm{Mult}(c_{n+1})(Gc_n) \to_\beta \mathrm{Mult}(c_{n+1})(c_{n!})$$

$$Gc_{n+1} \to_\beta F(G)c_{n+1} \to_\beta \mathrm{Mult}(c_{n+1})(Gc_n) \to_\beta \mathrm{Mult}(c_{n+1})(c_{n!})$$

- Hence, G computes the factorial function.

Outline I

Computable Functions

First we need to define what we mean by computable functions.
We will use definition given by (a la Herbrand-Kleene-Gödel) of recursive functions.

- **Base Functions:** Zero, successor, and projection functions.
- **Closure under Composition:** If f and g are computable, then $h(x_1, \ldots, x_n) = f(g(x_1, \ldots, x_n), \ldots)$ is computable.
- **Primitive Recursion:** If f is computable, then the function defined by:

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

$$h(n+1, x_1, \ldots, x_n) = g(n, h(n, x_1, \ldots, x_n))$$

  is also computable.
- **Minimization:** If f is computable, then the function defined by:

$$h(x_1, \ldots, x_n) = \min\{n : f(n, x_1, \ldots, x_n) = 0\}$$

  is also computable.

Base Functions

Base Functions

Base functions Z, S, and P_i^n are defined as:

1. Zero function:
$$Z(n) = 0 \forall n \in \mathbb{N}$$
2. Successor function:
$$S(n) = n + 1 \forall n \in \mathbb{N}$$
3. Projection function: For every $n \geq 1$ and every i with $1 \leq i \leq n$:
$$P_i^n(x_1, \ldots, x_n) = x_i$$

Composition

Definition

Given function $g : \mathbb{N}^m \to \mathbb{N}$ ($m \geq 1$) and any m functions $h_i : \mathbb{N}^n \to \mathbb{N}$ ($n \geq 1$), the **composition** of g and $h_1, \ldots, h_m$, denoted $g \circ (h_1, \ldots, h_m)$, is the function $f : \mathbb{N}^n \to \mathbb{N}$ given by:

$$f(x_1, \ldots, x_n) = g(h_1(x_1, \ldots, x_n), \ldots, h_m(x_1, \ldots, x_n)),$$

where $x_1, \ldots, x_n \in \mathbb{N}$.

Primitive Recursion

Definition

Given any functions $g : \mathbb{N}^m \to \mathbb{N}$ and $h : \mathbb{N}^{m+2} \to \mathbb{N}$ ($m \geq 1$), the function $f : \mathbb{N}^{m+1} \to \mathbb{N}$ is defined by **primitive recursion** as:

$$f(0, x_1, \ldots, x_m) = g(x_1, \ldots, x_m)$$

$$f(n+1, x_1, \ldots, x_m) = h(f(n, x_1, \ldots, x_m), n, x_1, \ldots, x_m)$$

for all $n, x_1, \ldots, x_m \in \mathbb{N}$.
If $m = 0$, then g is some fixed natural number, and we have:

$$f(0) = g, \quad f(n+1) = h(f(n), n).$$

Minimization

Definition

Given any function $g : \mathbb{N}^{m+1} \to \mathbb{N}$ ($m \geq 0$), the function $f : \mathbb{N}^m \to \mathbb{N}$ is defined as follows:

$$f(x_1, \ldots, x_m) = \text{the least } n \in \mathbb{N} \text{ such that } g(n, x_1, \ldots, x_m) = 0,$$

and is **undefined** if there is no such n satisfying this condition.

Notation

We say f is **defined by minimization** from g, and write:

$$f(x_1, \ldots, x_m) = \mu x \, [g(x, x_1, \ldots, x_m) = 0].$$

For short, we write $f = \mu g$.

Computable Functions

Definition

Definition 3.17 (Herbrand–Gödel–Kleene). The set of *partial computable* (or *partial recursive*) functions is the smallest set of partial functions (defined on $\mathbb{N}^n$ for some $n \geq 1$) which contains the base functions and is closed under:

1. Composition.
2. Primitive recursion.
3. Minimization.

Computable (Recursive) Functions

The set of *computable* (or *recursive*) functions is the subset of partial computable functions that are **total functions** (i.e., defined for all inputs).

Computable Functions

Kleene Normal Form

It can be proven every partial computable function $f : \mathbb{N}^m \rightarrow \mathbb{N}$ is computable as:

$$f = g \circ \mu h,$$

for some *primitive recursive functions* $g : \mathbb{N} \rightarrow \mathbb{N}$ and $h : \mathbb{N}^{m+1} \rightarrow \mathbb{N}$.

- The significance of this result is that f is built from **total functions** using composition and primitive recursion, with only a single minimization at the end.
- Before stating the main theorem, we need to define what it means for a numerical function to be definable in λ-calculus.

Lambda-Definability

Definition

A function $f : \mathbb{N}^n \to \mathbb{N}$ is said to be λ-**definable** if there exists a closed λ-term F such that:

1. $F\, c_{m_1} \cdots c_{m_n}$ has a normal form if and only if $f(m_1, \ldots, m_n)$ is defined.
2. If defined, $F\, c_{m_1} \cdots c_{m_n} \to_\beta c_{f(m_1,\ldots,m_n)}$.

Theorem Overview

Theorem

If a (total) function $f : \mathbb{N}^n \to \mathbb{N}$ is computable, then it is λ-definable.
If a (partial) function $f : \mathbb{N}^n \to \mathbb{N}$ is partial computable, then it is λ-definable.

Since the definition of computable functions is also same for turing machince the above theorem also tells us that the lambda calculus has same power as a Turing machine.

- Every total computable function is λ-definable.
- Every partial computable function is also λ-definable.
- This establishes the equivalence between the lambda-calculus and Turing machines.

Proof Outline

Step 1: Base Case

- The base functions are λ-definable.
- $\mathbf{Z_c}$ computes Z, and $\mathbf{Succ_c}$ computes S.
- The function U_i^n given by $U_i^n = \lambda x_1 \ldots x_n . x_i$ computes P_i^n.

Step 2: Closure under Composition

If g is λ-defined by G and $h_1, \ldots, h_m$ are λ-defined by $H_1, \ldots, H_m$, then $g \circ (h_1, \ldots, h_m)$ is λ-defined by:

$$F = \lambda x_1 \ldots x_n . G(H_1 x_1 \ldots x_n) \ldots (H_m x_1 \ldots x_n)$$

Step 3: Closure under Primitive Recursion

- If f is defined by primitive recursion from g and h, and G and H λ-define them, then:

$$F = \lambda n x_1 \dots x_m . \pi_1 (\mathrm{Iter}\, n \lambda z . \langle H \pi_1 z \pi_2 z x_1 \dots x_m, \mathbf{Succ_c}(\pi_2 z) \rangle \langle G x_1 \dots x_m, c_0 \rangle)$$

- This ensures F λ-defines f.

Step 3: Closure under Primitive Recursion

Proof: We will prove by induction

$$(\lambda z.\langle H\pi_1 z\pi_2 z\mathbf{c_{n_1}} \dots \mathbf{c_{n_m}}, \mathbf{Succ_c}(\pi_2 z)\rangle)^n\langle G\mathbf{c_{n_1}} \dots \mathbf{c_{n_m}}, \mathbf{c_0}\rangle \rightarrow_\beta \langle \mathbf{c}_{f(n,n_1,\dots,n_m)}, \mathbf{c_n}\rangle$$

Base case: $n = 0$

$$(\lambda z.\langle H\,\pi_1 z\,\pi_2 z\,\mathbf{c}_{n_1}\cdots\mathbf{c}_{n_m}, \mathbf{Succ_c}(\pi_2 z)\rangle)^0\,\langle G\mathbf{c}_{n_1}\cdots\mathbf{c}_{n_m}, \mathbf{c}_0\rangle$$

$$\xrightarrow{+}_\beta \langle G\mathbf{c}_{n_1}\cdots\mathbf{c}_{n_m}, \mathbf{c}_0\rangle = \langle \mathbf{c}_{g(n_1,\dots,n_m)}, \mathbf{c}_0\rangle = \langle \mathbf{c}_{f(0,n_1,\dots,n_m)}, \mathbf{c}_0\rangle.$$

Step 3: Closure under Primitive Recursion

$$
\begin{aligned}
&(\lambda z.\langle H\pi_1 z\pi_2 z\mathbf{c_{n_1}}\cdots\mathbf{c_{n_m}}, \mathbf{Succ_c}(\pi_2 z))\rangle^{n+1}\langle G\mathbf{c_{n_1}}\cdots\mathbf{c_{n_m}}, \mathbf{c_0}\rangle \\
&= (\lambda z.\langle H\pi_1 z\pi_2 z\mathbf{c_{n_1}}\cdots\mathbf{c_{n_m}}, \mathbf{Succ_c}(\pi_2 z))\rangle \\
&\quad (\lambda z.\langle H\pi_1 z\pi_2 z\mathbf{c_{n_1}}\cdots\mathbf{c_{n_m}}, \mathbf{Succ_c}(\pi_2 z))\rangle^{n}\langle G\mathbf{c_{n_1}}\cdots\mathbf{c_{n_m}}, \mathbf{c_0}\rangle) \\
&\xrightarrow{+}_\beta (\lambda z.\langle H\pi_1 z\pi_2 z\mathbf{c_{n_1}}\cdots\mathbf{c_{n_m}}, \mathbf{Succ_c}(\pi_2 z))\rangle\langle \mathbf{c}_{f(n,n_1,\ldots,n_m)}, \mathbf{c_n}\rangle \\
&\xrightarrow{+}_\beta \langle H\mathbf{c}_{f(n,n_1,\ldots,n_m)}\mathbf{c_n}\mathbf{c_{n_1}}\cdots\mathbf{c_{n_m}}, \mathbf{Succ_c}\mathbf{c_n}\rangle \\
&\xrightarrow{+}_\beta \langle \mathbf{c}_{h(f(n,n_1,\ldots,n_m),n,n_1,\ldots,n_m}, \mathbf{c_{n+1}}\rangle = \langle \mathbf{c}_{f(n+1,n_1,\ldots,n_m)}, \mathbf{c_{n+1}}\rangle.
\end{aligned}
$$

Step 4: Closure under Minimization

Suppose f is total and defined by minimization from g, where g is λ-defined by G. Define:

$$J = \lambda f x x_1 \dots x_m.\text{if } \mathbf{IsZero_c} G x x_1 \dots x_m \text{ then } x \text{ else } f(\mathbf{Succ_c} x) x_1 \dots x_m$$

$$F = \Theta J$$

Clearly:

$$F \mathbf{c}_n \mathbf{c}_{n_1} \cdots \mathbf{c}_{n_m} \xrightarrow{+}_\beta \begin{cases} \mathbf{c}_n & \text{if } g(n, n_1, \dots, n_m) = 0 \\ F \mathbf{c}_{n+1} \mathbf{c}_{n_1} \cdots \mathbf{c}_{n_m} & \text{otherwise} \end{cases}$$

- This ensures that F λ-defines f.
- Since F is total, some least n will be found.

Partial Computable Functions

To prove the result for partial computable functions, we use the Kleene normal form:

$$f = g \circ \mu h$$

where g and h are primitive recursive.

- Our previous proof ensures g and h are λ-definable.
- Minimization may fail, but since g is total, it remains well-defined.
- A rigorous proof is available in Hindley and Seldin (Chapter 4, Theorem 4.18).

Conclusion

- With some work, it is possible to show that lists can be represented in the λ-calculus:
  - We have to use the Pair combinators to store pair within a pair to form a linked list.
- Since the tape of turing machine can be represented as a list, we can simulate a Turing machine using the λ-calculus. (Tedious Task)
- The Construction of turing machine in λ-calculus mimics the proof that Turing machince computes a computable function.

Remark

- λ-calculus has the same power as Turing machines.
- This leads to undecidability results similar to the halting problem and Rice's theorem.

Scott-Curry Theorem: An analog of Rice's theorem follows as a corollary.

Outline I

Summary

- The lambda-calculus provides a minimalistic foundation for computation.
- Its syntax is based on variables, abstraction, and application.
- Substitution and α-conversion manage variable binding.
- β-reduction drives computation.
- Combinators, Church numerals, and fixed-point combinators illustrate its power.
- Every computable function is λ-definable.

Implications

- Equivalence to Turing machines shows universal computation.
- Fundamental for the design of functional programming languages.
- Offers insights into recursion and fixed-point theory.

Further Directions

- Study evaluation strategies (call-by-name, call-by-value).
- Explore type systems: Simply typed lambda-calculus.

Acknowledgements

These slides were made for seminar presentation for **Automata Theory and Computability** [UMC205] 2025 course at **Indian Institute of Science, Bengaluru** under the guidance of **Prof. Deepak D'Souza**.
We have refrenced:

- Proofs, Computability, Undecidability, Complexity, And the Lambda Calculus An Introduction [Jean Gallier and Jocelyn Quaintance]

Team Members:

- **Gavish Bansal:** gavishbansal@iisc.ac.in
- **Sehaj Ganjoo:** sehajganjoo@iisc.ac.in
- **Pratham Gupta:** prathamgupta@iisc.ac.in
- **Krishna Agarwal:** krishnaagarw@iisc.ac.in

Questions?

Thank you for your attention!

Any Questions?

Proof Sketch:(Part 1 $\Leftrightarrow$ Part 2)

Proof: (1)$\leftarrow$ (2)

Assume that (2) holds. Since $\xrightarrow{*}_\beta$ is contained in $\xleftrightarrow{*}_\beta$, if

$$M \xrightarrow{*}_\beta M_1 \quad \text{and} \quad M \xrightarrow{*}_\beta M_2,$$

then $M_1 \xleftrightarrow{*}_\beta M_2$.

Proof Sketch:(Part 1 $\Leftrightarrow$ Part 2)

Proof: (1)$\leftarrow$ (2)

Assume that (2) holds. Since $\xrightarrow{*}_\beta$ is contained in $\xleftrightarrow{*}_\beta$, if

$$M \xrightarrow{*}_\beta M_1 \quad \text{and} \quad M \xrightarrow{*}_\beta M_2,$$

then $M_1 \xleftrightarrow{*}_\beta M_2$.

Since (2) holds, there exists some λ-term M_3 such that

$$M_1 \xrightarrow{*}_\beta M_3 \quad \text{and} \quad M_2 \xrightarrow{*}_\beta M_3,$$

which is exactly statement (1).

Proof Sketch:(Part 1 $\Leftrightarrow$ Part 2)

Key Observation:

To prove that (1) implies (2), we use the fact:

$$\overset{*}{\leftrightarrow}_\beta = (\rightarrow_\beta \cup \leftarrow_\beta)^*$$

So, $M_1 \overset{*}{\leftrightarrow}_\beta M_2$ if and only if:

(a) $M_1 = M_2$, or

(b) There exists M_3 such that $M_1 \rightarrow_\beta M_3$ and $M_3 \overset{*}{\leftrightarrow}_\beta M_2$, or

(c) There exists M_3 such that $M_3 \rightarrow_\beta M_1$ and $M_3 \overset{*}{\leftrightarrow}_\beta M_2$.

Proof Sketch:(Part 1 $\Leftrightarrow$ Part 2)

Proof: (1)$\rightarrow$ (2): Induction on Number of Steps in $M_1 \overset{*}{\longleftrightarrow}_\beta M_2$

Case (a): $M_1 = M_2$
Then (2) holds trivially with $M_3 = M_1 = M_2$.

Case (b): $M_1 \rightarrow_\beta M_3$ and $M_3 \overset{*}{\longleftrightarrow}_\beta M_2$
By induction hypothesis, $\exists M_4$ such that:

$$M_3 \xrightarrow{*}_\beta M_4 \quad \text{and} \quad M_2 \xrightarrow{*}_\beta M_4$$

Hence,

$$M_1 \rightarrow_\beta M_3 \xrightarrow{*}_\beta M_4, \quad \text{so } M_1 \xrightarrow{*}_\beta M_4$$

Thus, (2) is satisfied.

Proof Sketch:(Part 1 $\Leftrightarrow$ Part 2)

Proof: (1)$\rightarrow$ (2): Induction on Number of Steps in $M_1 \overset{*}{\longleftrightarrow}_\beta M_2$

Case (c): $M_3 \rightarrow_\beta M_1$ and $M_3 \overset{*}{\longleftrightarrow}_\beta M_2$
By induction hypothesis, $\exists M_4$ such that:

$$M_3 \xrightarrow{*}_\beta M_4 \quad \text{and} \quad M_2 \xrightarrow{*}_\beta M_4$$

Since $M_3 \rightarrow_\beta M_1$ and $M_3 \xrightarrow{*}_\beta M_4$, by (1), $\exists M_5$ such that:

$$M_1 \xrightarrow{*}_\beta M_5 \quad \text{and} \quad M_4 \xrightarrow{*}_\beta M_5$$

Hence,

$$M_1 \xrightarrow{*}_\beta M_5 \quad \text{and} \quad M_2 \xrightarrow{*}_\beta M_4 \xrightarrow{*}_\beta M_5$$

proving (2).