# Computer-Assisted Formalization of Mathematics

Week 02 - Type theory and Curry-Howard

## First Half

- Cantor, Curry, Howard and friends
- Basic blocks of type theory
- Curry-Howard correspondence

## Second Half

- Lean worksheet

Jaume de Dios Pont - jdedios@nyu.edu

https://nyulean26.github.io/

Cantor, Curry, Howard and friends

Basic blocks of type theory

Curry-Howard Correspondence

Tactics

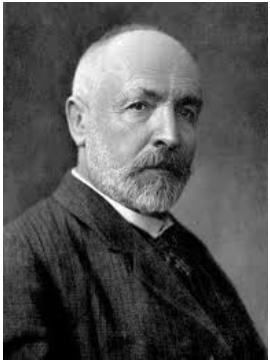
# Proof verifiers: Part programming language part proof

1. Proof verifiers are programming languages.
2. They allow us to verify mathematics.
3. They also allow us to verify computations (algorithms).

**What's the best way to do both?**

# ZFC: The classical foundations of Math

- ZFC – believed to be self-consistent.
- Category theorists sprinkle extra large sets (Grothendieck axioms).
- Multiple choices of foundations: They don't affect 99.9% of mathematics
- To formalize we need a **choice**.



*"Everything is a set"*
– Georg Cantor*

* maybe

# Set Theory (ZFC)

**Ingredients**

- **Primitives:** membership $x \in y$, equality $x = y$.
- **Objects:** sets; everything is a set.
- **Axioms:** Empty Set, Pairing, Union, Extensionality, Infinity, ...

**Examples**

$$0 := \varnothing, \qquad 1 := \{\varnothing\}, \qquad 2 := \{\varnothing, \{\varnothing\}\}, \ \ldots$$

# Set Theory (ZFC)

**Ingredients**

- **Primitives:** membership $x \in y$, equality $x = y$.
- **Objects:** sets; everything is a set.
- **Axioms:** Empty Set, Pairing, Union, Extensionality, Infinity, ...

**Examples**

$$0 := \varnothing, \qquad 1 := \{\varnothing\}, \qquad 2 := \{\varnothing, \{\varnothing\}\}, \ \ldots$$

$$(a, b) := \big\{\{a\}, \{a, b\}\big\} \quad \text{(Kuratowski)} \qquad X \to Y := \{f \in X \times Y, \ \forall x, \ \exists!, \ y : (x, y) \in f\}$$

# Set Theory (ZFC)

**Ingredients**

- **Primitives:** membership $x \in y$, equality $x = y$.
- **Objects:** sets; everything is a set.
- **Axioms:** Empty Set, Pairing, Union, Extensionality, Infinity, ...

**Examples**

$$0 := \varnothing, \qquad 1 := \{\varnothing\}, \qquad 2 := \{\varnothing, \{\varnothing\}\}, \ \ldots$$

$$(a, b) := \big\{\{a\}, \{a, b\}\big\} \quad \text{(Kuratowski)} \qquad X \to Y := \{f \in X \times Y, \ \forall x, \exists!, \ y : (x, y) \in f\}$$

**Why**

- Uniform foundation: **nearly all** mainstream mathematics can be encoded in ZFC.
- Simple core with powerful expressive reach (numbers, functions, spaces).
- *But:* proofs are **external** objects; the system talks about math objects, not proofs.

# Untyped $\lambda$-Calculus (Church, 1930s)

**Ingredients**

- Variables $x, y, z, \ldots$
- Abstraction $\lambda x.\, e$ (function taking $x$ and returning $e$)
- Application $(f\ x)$ (apply function $f$ to $x$)

**Examples**

$$(\lambda x.\, x + 1)\, 2 \longrightarrow 3 \qquad (\lambda x.\lambda y.\, x + y)\, 2\, 3 \longrightarrow 5$$

A. Church

# Untyped $\lambda$-Calculus (Church, 1930s)

**Ingredients**

- Variables $x, y, z, \ldots$
- Abstraction $\lambda x.\, e$ (function taking $x$ and returning $e$)
- Application $(f\ x)$ (apply function $f$ to $x$)

**Examples**

$$(\lambda x.\, x + 1)\, 2 \longrightarrow 3 \qquad (\lambda x.\lambda y.\, x + y)\, 2\, 3 \longrightarrow 5$$

A. Church

**Why**

- A minimal, elegant model of **computation**; equivalent power to Turing machines.
- *But:* too permissive (self-application, $(\lambda x.\, xx)(\lambda x.\, xx)$), no static guarantees.

# Foundations of math: Everything is a ___

*"Everything is a set"*
**Set-Theoretic (ZFC)**

*"Everything is a map"*
**$\lambda$-Calculus (Church)**

# Foundations of math: Everything is a ___

*"Everything is a set"*
**Set-Theoretic (ZFC)**

- $0 := \varnothing$
- $1 := \{\varnothing\}$
- $2 := \{\varnothing, \{\varnothing\}\}$
- $\vdots$
- $n + 1 := n \cup \{n\}$

*"Everything is a map"*
**$\lambda$-Calculus (Church)**

# Foundations of math: Everything is a ___

*"Everything is a set"*
**Set-Theoretic (ZFC)**

- $0 := \varnothing$
- $1 := \{\varnothing\}$
- $2 := \{\varnothing, \{\varnothing\}\}$
- $\vdots$
- $n + 1 := n \cup \{n\}$

*"Everything is a map"*
**$\lambda$-Calculus (Church)**

- $0 := \lambda f.\lambda x.\, x$
- $1 := \lambda f.\lambda x.\, f(x)$
- $2 := \lambda f.\lambda x.\, f(f(x))$
- $\vdots$
- $n := \lambda f.\lambda x.\, f^n(x)$

# Foundations of math: Everything is a ___

*"Everything is a set"*
**Set-Theoretic (ZFC)**

- $0 := \varnothing$
- $1 := \{\varnothing\}$
- $2 := \{\varnothing, \{\varnothing\}\}$
- $\vdots$
- $n + 1 := n \cup \{n\}$

**Ill-posed:** $0 \in 3$, $\mathbb{R} \notin +$.

*"Everything is a map"*
**$\lambda$-Calculus (Church)**

- $0 := \lambda f.\lambda x.\, x$
- $1 := \lambda f.\lambda x.\, f(x)$
- $2 := \lambda f.\lambda x.\, f(f(x))$
- $\vdots$
- $n := \lambda f.\lambda x.\, f^n(x)$

**Nonterminating:** $(\lambda x.\, xx)(\lambda x.\, xx)$

**Both are too general!**

# Two Worlds, One Goal

**Sets (ZFC)**

- Math as **collections** and membership ($\in$).
- Can encode numbers, structures, categories, ...
- Proofs live in the meta-theory.

**Computation ($\lambda$-calculus)**

- Math as **transformations/programs**.
- Direct link to algorithms and evaluation.
- Too unstructured when untyped.

# Two Worlds, One Goal

**Sets (ZFC)**

- Math as **collections** and membership ($\in$).
- Can encode numbers, structures, categories, ...
- Proofs live in the meta-theory.

**Computation ($\lambda$-calculus)**

- Math as **transformations/programs**.
- Direct link to algorithms and evaluation.
- Too unstructured when untyped.

**We want a system good for math objects and proof objects.**

Cantor, Curry, Howard and friends

Basic blocks of type theory

Curry-Howard Correspondence

Tactics

# Untyped $\lambda$-calculus: Alpha, beta, and eta conversion

**$\alpha$-conversion: rename a bound variable**

$$\lambda x.\, x + y \equiv_\alpha \lambda z.\, z + y.$$

The new name must not capture a free variable.

**Reduction example**

$$(\lambda g.\lambda x.\, g\, x)\, f$$

# Untyped $\lambda$-calculus: Alpha, beta, and eta conversion

**$\alpha$-conversion: rename a bound variable**

$$\lambda x.\, x + y \equiv_\alpha \lambda z.\, z + y.$$

The new name must not capture a free variable.

**$\beta$-conversion: substitute the argument**

$$(\lambda x.\, x + 1)\, 3 \longrightarrow_\beta 3 + 1.$$

**Reduction example**

$$(\lambda g.\lambda x.\, g\, x)\, f$$

$$\downarrow\beta$$

$$\lambda x.\, f\, x$$

# Untyped $\lambda$-calculus: Alpha, beta, and eta conversion

**$\alpha$-conversion: rename a bound variable**

$$\lambda x.\, x + y \equiv_\alpha \lambda z.\, z + y.$$

The new name must not capture a free variable.

**$\beta$-conversion: substitute the argument**

$$(\lambda x.\, x + 1)\, 3 \longrightarrow_\beta 3 + 1.$$

**$\eta$-conversion: remove a redundant wrapper**

$$\lambda x.\, f\, x \longrightarrow_\eta f \qquad \text{when } x \text{ is not free in } f.$$

**Reduction example**

$$(\lambda g.\lambda x.\, g\, x)\, f$$
$$\downarrow\beta$$
$$\lambda x.\, f\, x$$
$$\downarrow\eta$$
$$f$$

*normal form*

# Termination and normal forms

$\beta$-**Normal Form** An expression that connot be further simplified.

They don't always exist!

$$(\lambda x.\, x)\, y \longrightarrow_\beta y \qquad \underbrace{(\lambda x.\, x\, x)(\lambda x.\, x\, x)}_{\Omega} \longrightarrow_\beta \Omega \longrightarrow_\beta \cdots$$

(Stops) (Loops forever)

# Termination and normal forms

$\beta$-**Normal Form** An expression that connot be further simplified.

They don't always exist!

$$(\lambda x.\, x)\, y \longrightarrow_\beta y \qquad \underbrace{(\lambda x.\, x\, x)(\lambda x.\, x\, x)}_{\Omega} \longrightarrow_\beta \Omega \longrightarrow_\beta \cdots$$

(Stops) (Loops forever)

**Confluence:** (Church-Rossler Theorem) different $\beta-$reductions always be reduced back together.

**Why do we care** Can we decide definitional equality? Can we reduce proofs?

# Typed lambda calculus

Untyped lambda calculus lets us apply any term to any other term. Types tell us which applications make sense.

If $f : A \to B$ and $a : A$, then $f\ a : B$. The input must have the type the function expects.

$(\lambda x : \mathbb{N}.\, x + 1)\, 3$ **Valid:** the input has type $\mathbb{N}$.

$(\lambda x : \mathbb{N}.\, x + 1)$ `"hello"` **Type error:** a string is not a natural number.

Types also describe what we are trying to construct. We will use this to express mathematical statements as types and their proofs as terms.

# Types and Terms

**Ingredients**

- Judgments: $\Gamma \vdash t : T$ (term $t$ has type $T$ under context $\Gamma$).
- Examples of types: $\mathbb{N}, \mathbb{Z}, \mathbb{R}, \ldots$
- Terms *inhabit* types: $3 : \mathbb{N}$, $\langle 2, 3 \rangle : \mathbb{N} \times \mathbb{N}$.
- **Type checking** decides if $t : T$ is well-formed.

**Examples**

$$\lambda x : \mathbb{N}.\, x + 1 : \mathbb{N} \to \mathbb{N} \qquad \langle 2, 3 \rangle : \mathbb{N} \times \mathbb{N}$$

**Why it matters**

- Types provide **static guarantees**.
- **Idea:** In Lean, each lemma is a **term** whose **type** is the statement.

# Product Types $A \times B$

## Ingredients

- Pairing:
  $a : A$ and $b : B$,
  $\langle a, b \rangle : A \times B$.
- Projections:
  $\pi_1 : A \times B \to A$
  $\pi_2 : A \times B \to B$.

## Examples

$\langle 2, 3 \rangle : \mathbb{N} \times \mathbb{N}, \qquad \pi_1 \langle a, b \rangle = a$

```
/-- Make a pair (Nat × String) -/
def p : Nat × String := (2, "hi")

/-- Projections fst, snd -/
def fst {A B} (x : A × B) : A := x.fst
def snd {A B} (x : A × B) : B := x.snd

/-- Swap components (A × B) → (B × A) -/
def swap {A B} (x : A × B) : B × A := (x.snd, x.fst)

#eval fst p                    -- 2
#eval (swap p).fst             -- "hi"
```

# Sum Types $A \oplus B$ ($\vee$)

## Ingredients

- Injections:
  $\mathsf{inl} : A \to A \oplus B$
  $\mathsf{inr} : B \to A \oplus B$.

## Examples

$$\mathsf{inl}(2 : \mathbb{N}) : \mathbb{N} \oplus \mathbb{Z}$$

$$\mathsf{inr}(-5) : \mathbb{N} \oplus \mathbb{Z}.$$

```
/-- Two example values of Nat ⊕ String -/
def leftVal  : Nat ⊕ String := Sum.inl 42
def rightVal : Nat ⊕ String := Sum.inr "ok"

/-- Case analysis on a sum to produce a String -/
def showSum : (Nat ⊕ String) → String
| Sum.inl n  => s!"nat:{n}"
| Sum.inr s  => s!"str:{s}"

#eval showSum leftVal           -- "nat:42"
#eval showSum rightVal          -- "str:ok"
```

# Function Types $A \to B$

**Ingredients**

- **Elimination** (application): $f : A \to B$ and $a : A$ then $f\ a : B$.
- **Introduction:** For $e : B$ there is $(\text{fun } \_ \mapsto e) : A \to B$
- **Currying:** $A \times B \to C \simeq A \to (B \to C)$.

```
/-- Simple function ℕ → ℕ -/
def succ1 : Nat → Nat := fun x => x + 1

/-- Function composition (A → B) → (B → C) → (A → C) -/
def compose {A B C} (g : B → C) (f : A → B) : A → C :=
  fun a => g (f a)

#eval succ1 5                    -- 6
#eval compose (· + 10) succ1 7   -- (7+1)+10 = 18
```

# Empty and Unit Types ($\bot, \top$)

**Ingredients**

- **Unit** (a.k.a. $\top$): exactly one inhabitant ().
- **Empty** (a.k.a. $\bot$): no inhabitants.

Cantor, Curry, Howard and friends

Basic blocks of type theory

Curry-Howard Correspondence

Tactics

# Curry–Howard: Proofs $\iff$ Terms in Typed $\lambda$-calculus

- **Propositions $\leftrightarrow$ Types**
  **Proofs $\leftrightarrow$ Terms**.
- "To prove $P$" $\equiv$ "Exhibit a term of type $P$".
- Logic becomes **internal**: proofs are **first-class** objects in the system.

| **Logic** | **Computation** |
|---|---|
| Proposition ($P$) | Type ($P$) |
| Proof of $P$ | Term of type $P$ |
| Implies ($P \to Q$) | Function ($P \to Q$) |
| AND ($P \wedge Q$) | Pair ($P \times Q$) |
| OR ($P \vee Q$) | Sum type ($P + Q$) |
| FALSE ($\bot$) | Empty type |
| TRUE ($\top$) | Unit type |

# Curry Howard Correspondence

**Building blocks**

| Logic | Type Theory | Logical fact | Term |
|---|---|---|---|
| AND | Product type | $a \wedge b \implies a$ | $\pi_1 : A \times B \to A$ |
| OR | Sum type | $a \implies a \vee b$ | $\text{inl} : A \to A \oplus B$ |
| $\implies$ | Map type | $((a \wedge b) \implies c) \implies (a \implies (b \implies c))$ | Currying |
| | Empty type | $\bot \implies a$ | $\bot \to A$ |
| | Constant map | $a \implies \top$ | $(x \mapsto ()) : A \to \top$ |

# First-order logic and type theory

$\exists x, P(x)$ and $\forall x, P(x)$ are **dependent** types (more next week)

$\forall x, P(x)$ is a *function* type mapping $x$ to a proof of $P(x)$

$$x \mapsto p \in P(x)$$

$\exists x, P(x)$ is a *pair* type containing a witness $x$ and a proof of $P(x)$

$$\langle x, p \rangle \text{ where } p \in P(x)$$

the type of the second part **depends** on the first part!

# From Here to Lean

- Modern proof assistants = **typed $\lambda$-calculus** + **inductive** + **dependent** types.
- In Lean:
  - Prop: universe of propositions.
  - A lemma is a **term** whose **type** is the statement.
  - **Proof Irrelevance:** For any $P$ : Prop and any $p, q : P$, we impose $p = q$.
- A new look at theorems in Lean

```
example (a b: Prop):
    a∧b → b∧a
    := fun ⟨x,y⟩ ↦ ⟨y,x⟩
```

# Functions can transform evidence

If `hP : P` and `hPQ : P → Q`, then

$$\underbrace{\texttt{hPQ}}_{\text{takes evidence of } P} \quad \underbrace{\texttt{hP}}_{\text{evidence of } P} \quad : \quad Q.$$

```
example (P Q : Prop) : P → (P → Q) → Q :=
  fun hP hPQ => hPQ hP
```

- **Curry–Howard:** proving $P$ means providing a term of type $P$.
- An implication is a function taking a proof to a proof.
- Negation fits too: $\neg P$ means $P \to$ False.

**A statement is a type; a proof is the requested term.**

# Two more classes of types in Lean

Lean admits two other classes of types: **inductive types** and **dependent types**.

**Inductive types.** Types generated by named constructors, such as `Nat` and `List`, which support recursive definitions and proofs by induction.

**Dependent types.** Types that may mention values, such as `Vector A n`, letting a type record specifications like the length of a vector.

**We develop both in Week 3.**

Cantor, Curry, Howard and friends

Basic blocks of type theory

Curry-Howard Correspondence

Tactics

# Tactics build proof terms

A goal asks for a term of a particular type. Tactics help us construct that term, one step at a time.

Suppose we have $h_P : P$ and $h_{PQ} : P \to Q$, and we want to prove $Q$.

$$\underbrace{\texttt{?proof}}_{\text{need a proof of } Q} \longrightarrow \underbrace{h_{PQ}\ \texttt{?input}}_{\text{now need a proof of } P} \longrightarrow \underbrace{h_{PQ}\ h_P}_{\text{complete proof of } Q}$$

Each step fills part of the term and leaves goals for whatever is missing. A goal also records which variables and hypotheses we may use.

**Lean checks the completed term to verify the proof.**

# AND, OR, and introducing assumptions

The shape of a statement tells us what its proof needs.

| Statement | To prove it | To use a proof of it |
|---|---|---|
| $P \wedge Q$ | Use `constructor`, then prove both goals. | Use `rcases` to name both parts. |
| $P \vee Q$ | Choose `left` or `right`. | Use `rcases` to make two branches. |
| $P \to Q$ | `introduce` $P$, then prove $Q$. | `apply` it, then prove $P$. |

When we assume $P$, its proof becomes available in the current context. In an OR case split, each branch has its own assumption.

# What are those question marks?

Lean uses **metavariables** for terms it has not filled in yet.

- `x : Nat` is an input we already have.
- `?w : Nat` is a natural number Lean still needs to determine.
- The missing term could be a value, a proof, or a type.

For $\exists n : \mathbb{N},\ n = 7$, we need a witness and a proof that it equals 7:

$$\texttt{?w} : \mathbb{N} \qquad \text{and} \qquad \texttt{?w} = 7.$$

Both occurrences of `?w` stand for the same unknown. Once it is set to 7, Lean uses 7 in both places.

**A question mark means Lean still needs a term here.**