> Cairo University
> Faculty of Science
> Department of Mathematics

< Mathematics Club />

Leaning into Lean

{ Proofs ≃ Programs }

---

def Mina Baslious : Speaker := by
exact ⟨MathMajor, SecondYear⟩

Mathematics Club

May 6, 2026

-- section --

What is a Formal Proof?

// Informal vs. Formal Proof

Informal (what we write)

- Arguments in natural language
- Steps justified by intuition
- Implicit lemmas and logic

Formal (what machines are able to check)

- Every step derived from axioms
- No implicit reasoning
- Machine-verifiable

Key Question

Can we make the gap between these two disappear?

// Why Does This Matter?

- **Errors in long proofs** are not hypothetical
- **Vladimir Voevodsky** (Fields Medal 2002): discovered an error in his own published work; began formalizing mathematics in Coq (Rocq Now)
- Modern ambition: *Four Color Theorem*, *Fermat's Last Theorem project* — real research-level mathematics being formalized now

-- section --

What is a Proof Assistant?

// Proof Assistants

A **proof assistant** is a program that:

1. Accepts a mathematical statement
2. Asks you to construct a proof interactively
3. Checks every step — it does not find proofs for you

Notable systems: **Lean 4**, Rocq, Agda, Isabelle

Why Lean?

> Mathlib: a large unified library of formalized mathematics
> Active community and ongoing research-level projects

-- section --

The Curry--Howard Correspondence

// Logic Encoded in Type Theory

The central idea of Lean's foundation:

$$\boxed{\textbf{Propositions} \longleftrightarrow \textbf{Types} \qquad \textbf{Proofs} \longleftrightarrow \textbf{Terms}}$$

This is not a metaphor. In Lean, **a proof literally is a term** whose type is the proposition it proves.

$$t : T \iff t \text{ is a witness to the truth of } T$$

// The Curry--Howard Correspondence

Logic	Type Theory	A proof is...
$\top$	`Unit`	a pregiven single term
$\bot$	`Null`	no term exists
$P \wedge Q$	`P × Q`	a pair (p, q)
$P \vee Q$	`P ⊕ Q`	`inl` p or `inr` q
$P \Rightarrow Q$	`P → Q`	a function
$\neg P$	`P → Null`	a function to absurdity
$\forall x,\ P(x)$	`(x : A) → P x`	a dependent function
$\exists x,\ P(x)$	`Σ x : A, P x`	a pair (witness, proof)

// Seeing the Correspondence Concretely

Claim: $P \Rightarrow P$ (trivially true)

Under Curry–Howard this is the type `P → P`. Its proof *is* the identity function:

```
theorem trivial (P : Prop) : P → P :=
  fun h ↦ h          -- the identity function, literally
```

Claim: $(P \wedge Q) \Rightarrow P$ (projection)

```
theorem and_elim_left (P Q : Prop) : P ∧ Q → P :=
  fun h ↦ h.left     -- extract the first component of the pair
```

// Dependent Types: the Key Upgrade

Ordinary functions: `A → B` (output a term from fixed type)

Dependent functions: `B : A → Type` (outputs a type that *possibly varies* with the input)

This is exactly what encodes **universal quantification**:

$$\forall x \in A,\ P(x) \quad \longleftrightarrow \quad \texttt{(x : A) → P x}$$

Here, `P : A → Prop`.
A proof is a function that, given *any* x, returns a proof of $P(x)$.

And **existential quantification**:

$$\exists x \in A,\ P(x) \quad \longleftrightarrow \quad \texttt{Σ x : A, P x}$$

A proof is a *dependent pair*: a specific witness a together with a proof that $P(a)$ holds. The witness is **mandatory**.

-- section --

Lean is Constructive

// Constructivity: A Consequence of Curry--Howard

Under the Curry–Howard encoding, LEM would be a term of type:

```
(P : Prop) → P ∨ ¬P
```

i.e. a function that, given *any* proposition `P`, produces either a proof of `P` or a proof of `¬P`.

This is impossible to write as a program

No algorithm decides arbitrary propositions — that would solve the halting problem. LEM as a *computable* function does not exist.

So Lean's core has **no Law of Excluded Middle by default**:

```
example (P : Prop) : P ∨ ¬P := by
  exact Classical.em P   -- must be invoked explicitly
```

LEM is *available*, but must be **consciously imported** — constructivity is not a philosophical stance here (in Lean), but a **consequence of taking Curry–Howard seriously**.

-- section --

Types in Lean

// The Type Hierarchy

Everything in Lean has a type — including types themselves:

```
#check Nat          -- Nat : Type
#check Type         -- Type : Type 1
#check Prop         -- Prop : Type
#check True         -- True : Prop
```

- `Prop`: the type of all propositions
- `Type`: the type of all data types
- The hierarchy (`Type 0`, `Type 1`, …) exists to avoid Russell's paradox

-- section --

Tactic Mode vs. Term Mode

// Two Ways to Write a Proof

Goal: prove $P \wedge Q \to R \leftrightarrow P \to Q \to R$

Term mode (write the proof object directly):

```
theorem Curry (P Q R : Prop) : P ∧ Q → R ↔ P → Q → R :=
  ⟨fun x y z ↦ x ⟨y, z⟩, fun x y ↦ (x y.1) y.2⟩
```

// Two Ways to Write a Proof

Goal: prove $P \wedge Q \to R \leftrightarrow P \to Q \to R$

Tactic mode (describe proof steps interactively):

```
theorem Curry (P Q R : Prop) : P ∧ Q → R ↔ P → Q → R := by
  constructor
  intro h p q
  exact h ⟨p, q⟩
  intro h pandq
  exact h pandq.left pandq.right
```

Both produce the same proof term. Tactics are scaffolding — Lean builds the term underneath.

// Common Tactics

Tactic	What it does
`intro h`	Introduce a hypothesis
`exact h`	Close goal with this term
`apply f`	Reduce goal using *f*
`rw [h]`	Rewrite using equation *h*
`have h : P`	Introduce intermediate goal
`simp`	Simplify automatically
`constructor`	Split a conjunction or existence
`cases h`	Case split on *h*

-- section --

Elaboration: What Lean Fills In For You

// The Gap Between Human and Machine

Mathematicians speak at a high level of abstraction. Machines need every detail made explicit.

The **elaborator** is Lean's engine that bridges this gap — it infers all the information you left implicit.

Consider the statement

If f is linear then $f(2x+y) = 2f(x) + f(y)$

This statement is clear to any mathematician. But how much is it leaving unsaid?

// Making It More Explicit --- Step by Step

Level 1 (what we say informally):

$$\text{If } f \text{ is linear then } f(2x + y) = 2f(x) + f(y)$$

// Making It More Explicit --- Step by Step

Level 1 (what we say informally):

$$\text{If } f \text{ is linear then } f(2x+y) = 2f(x) + f(y)$$

Level 2 (name the spaces):
If U, V are vector spaces and $f\colon U \to V$ is linear, then

$$\forall\, x, y \in U, \quad f(2x+y) = 2f(x) + f(y)$$

// Making It More Explicit --- Step by Step

Level 1 (what we say informally):

$$\text{If } f \text{ is linear then } f(2x+y) = 2f(x) + f(y)$$

Level 2 (name the spaces):
If U, V are vector spaces and $f\colon U \to V$ is linear, then

$$\forall\, x, y \in U, \quad f(2x+y) = 2f(x) + f(y)$$

Level 3 (name the field):
If K is a field, U, V are K-vector spaces, $f\colon U \to V$ linear, then

$$\forall\, x, y \in U, \quad f(2x+y) = 2f(x) + f(y)$$

// The Fully Explicit Version --- and Why It Hurts

Level 4 (explicit carrier sets $[U], [V]$):
If K is a field, U, V are K-vector spaces, $f\colon [U] \to [V]$ linear, then

$$\forall\, x, y \in [U], \quad f(2x + y) = 2f(x) + f(y)$$

// The Fully Explicit Version --- and Why It Hurts

Level 4 (explicit carrier sets $[U], [V]$):
If K is a field, U, V are K-vector spaces, $f\colon [U] \to [V]$ linear, then

$$\forall\, x, y \in [U], \quad f(2x + y) = 2f(x) + f(y)$$

Level 5 (every operation subscripted by its space):
If K is a field, U, V are K-vector spaces, $f\colon [U] \to [V]$ linear, then

$$f(2_K \cdot_U x \ +_U \ y) \ = \ 2_K \cdot_V f(x) \ +_V \ f(y)$$

// What Lean Actually Lets You Write

```
import Mathlib -- do not do that

variable
  {K : Type*} [Field K]
  {U : Type*} [AddCommGroup U] [Module K U]
  {V : Type*} [AddCommGroup V] [Module K V]
  {f : U →ₗ[K] V}

example : ∀x y, f (2 • x + y) = 2 • f x + f y := by simp
```

-- section --

Honest Limitations

// What Lean Cannot Do

- **It does not find proofs** — that is proof search / AI, a separate problem
- **Formalization is slow**: a one-page informal proof can take days to formalize
- **Mathlib is incomplete**: not all of mathematics is there yet
- **The learning curve is real**: the game server is a gentle on-ramp, production Lean is harder

The tradeoff

You get absolute certainty at the cost of significant effort. Whether that is worth it depends on what you are proving.

-- section --

Why Should You Care?

// The Case for Formalization

- **Error detection**: long proofs in modern mathematics are hard to check by hand
- **Deeper understanding**: formalizing forces you to make every implicit argument explicit
- **Collaboration**: Mathlib is a collective project — formalized proofs are reusable and composable
- **The future**: Will journals require formalized proofs for certain results?

-- section --

Where to Go Next

// Resources

- **Lean Game Server** — `adam.math.hhu.de` (where to start)
- **Theorem Proving in Lean 4** — `leanprover.github.io/theorem_proving_in_lean4` (free, official)
- **Mathematics in Lean** — `leanprover-community.github.io/mathematics_in_lean` (closer to Mathlib)
- **Mathlib documentation** — `leanprover-community.github.io/mathlib4_docs`

// Talks inspired this talk

- **Lean4 and the Curry-Howard Isomorphism (Luis Wirth)** — https://youtu.be/Sy_4z751YWI
- **Mario Carneiro: Lean4Lean: Formalizing the type theory of Lean** — https://youtu.be/GOrIMiI2zLA
- **The dawn of formalized mathematics** *by Andrej Bauer* — https://youtu.be/zp6WleEjHUg
- **Intensionality, Invariance, and Univalence, Steve Awodey** — https://youtu.be/gs3L1eSso_k

Questions?

Email: basiliousmina2@gmail.com
Website : https://minabasilious.github.io