"The only way to learn mathematics is to do mathematics."

— Paul Halmos

Lebanese Math Day: Proofs are Programs

How Types Let Us Prove Theorems and Secure Software

Daniel Dia
American University of Beirut (AUB)
Dep. of Electrical & Computer Engineering
https://danieldia.me/

A 60-Second Primer: Formal Verification

Verification: using mathematical proof — not testing — to guarantee a program is correct for *every* possible input, not just the ones you happened to try.

Specification: a precise mathematical statement of what a program *should* do, e.g. "`sort` returns a permutation of the input where each element $\leq$ the next." Verification proves the code matches the spec.

Two ways to get there:

- **Automated** (SAT/SMT solvers): push a button, no human effort, *but* limited to specific domains, and can fail to find a proof that actually exists
- **Interactive** (Lean, Rocq, Isabelle): a human guides the proof, the computer checks every step (slower, but works for *any* statement you can write down)

Lean's approach: automation, *inside* an interactive framework.

Section 1

What Is a Type?

What Is a Type?

Definition: A type is a specification of the collection of values (or programs) that a computation could possibly produce.

- `Nat`: the natural numbers
- `Bool`: `true` and `false`
- $\texttt{Nat} \to \texttt{Nat}$: functions from naturals to naturals

A term t of type T (written $t : T$) is a **witness**, i.e. concrete evidence that a value of type T can actually be built.

Today's claim: this idea of "witness" is doing more work than it looks like it's doing.

Types, More Broadly

Beyond `Nat` and `Bool`, "type" covers a lot of ground:

- **Basic types**: Nat, Int, Bool, String, and the two extremes Empty (by def. no values) and Unit (by def. exactly one value)
- **Function types**: Bool $\to$ Nat $\to \mathbb{Z}$, which is a function of two arguments
- **Product types**: Nat $\times$ String, a Nat *and* a String, together (tuple, kind of)
- **Sum types**: Nat $+$ String, a Nat *or* a String, tagged so you know which was used
- **Polymorphic types**: List Nat, Option Nat, or fully generic ones like $\alpha \to \alpha$
- **Type constructors**: List : Type $\to$ Type, i.e. a function that builds a *type* out of a type

Witnesses and Type Inhabitation

Type inhabitation problem: given a type σ, does a term of type σ exist? Fine for simpler type systems, but very problematic generally.

- `Nat` is inhabited: `0 : Nat` is a witness
- `Empty` is *not* inhabited: no constructor can build one
- In general, this problem is **undecidable**

Hang on to that last point (we'll see exactly why it's undecidable in a few slides)!

A Taste of Lambda Calculus

Lean's entire term language is built on the λ-calculus: functions, and one rule of computation.

Lambda abstraction: $\lambda x.\, M$ = "a function of x returning M"

Beta reduction: $(\lambda x.\, M)\, N \to_\beta M[N/x]$

Example: $(\lambda y.\, y \times y)\, 5 \to_\beta 5 \times 5 = 25$

```
#check fun x : Nat => x + 1        -- Nat -> Nat
#eval  (fun x : Nat => x + 1) 5    -- 6
```

Encoding Data as Functions (Church Encodings)

In the λ-calculus, everything is a function (even data!!!).

Church Booleans:

- $true = \lambda x.\, \lambda y.\, x$
- $false = \lambda x.\, \lambda y.\, y$

If-then-else:

$$if = \lambda b.\, \lambda x.\, \lambda y.\, b\, x\, y$$

A boolean *is* a choice function.

Church Numerals: n applies f exactly n times

- $0 = \lambda f.\, \lambda x.\, x$
- $1 = \lambda f.\, \lambda x.\, f\, x$
- $2 = \lambda f.\, \lambda x.\, f\, (f\, x)$

Successor:

$$succ = \lambda n.\, \lambda f.\, \lambda x.\, f\, (n\, f\, x)$$

A number *is* an iterator.

Two Roads to the Same Computation (1936)

Turing Machine (Alan Turing)

- Infinite tape, a head that reads/writes, state transitions
- Mutable state, step-by-step execution
- Lineage: FORTRAN, C, Java, von Neumann architecture

λ-Calculus (Alonzo Church)

- Functions as first-class values
- Computation by substitution (β-reduction)
- No mutable state, no side effects
- Lineage: LISP, ML, Haskell, Lean

Church–Turing Thesis: every function computable by a Turing machine is expressible in the λ-calculus, and vice versa. These two totally different ways of computation define the *exact same* class of computable functions.

The Typed Lambda Calculus

Problem: nothing stops nonsense like applying *not* to a number, or using 42 as a function.

Solution: assign a *type* to every term.

- $true$: Bool
- 3 : Nat
- λx:Bool. $not\ x$: Bool $\to$ Bool
- λx:Nat. λy:Nat. $x + y$: Nat $\to$ Nat $\to$ Nat

Typing rule: you may only apply $f : A \to B$ to an argument of type A. Applying a term of type other than A, e.g. `not 5`, becomes a compile-time type error.

Note: $\to$ is **right-associative**:

$$\text{Bool} \to \text{Nat} \to \mathbb{Z} \;=\; \text{Bool} \to (\text{Nat} \to \mathbb{Z})$$

A function of two arguments is really a function returning a function.

The Limitation of Simple Types

Simple types give us things like `List Nat`, but they can't really tell us *much* about what's inside:

- How many elements are in the list?
- Is the list sorted?
- Are all the elements positive?

The fix: let types depend on *values*.

- `Vector Nat n`: a list of *exactly* n naturals
- `Fin n`: the naturals strictly less than n

This is **Dependent Type Theory**, one hierarchy "ring" above everything we've seen so far, and what Lean actually really uses.

The Curry–Howard Isomorphism (1)

Types $\equiv$ Propositions.

Terms $\equiv$ Proofs.

This is the single most important idea in this talk. Checking a proof and type-checking a program are, formally, *the same act of computation*.

Type theory	Logic
Type A	Proposition A
Term $t : A$	Proof of A
$A \to B$	$A \implies B$
$A \times B$	$A \wedge B$
$A + B$	$A \vee B$
`Empty`	$\bot$ (False)
`Unit`	$\top$ (True)
Type inhabitation	Provability

The Curry–Howard Isomorphism (2)

Modus ponens, i.e. $A \implies B$ and A, conclude B:

```
def modus_ponens {A B : Prop} (h1 : A -> B) (h2 : A) : B :=
  h1 h2
```

Hypothetical syllogism, i.e. $(A \implies B) \implies (B \implies C) \implies (A \implies C)$, which is just function composition:

```
def chain {A B C : Prop} (f : A -> B) (g : B -> C) : A -> C :=
  fun h => g (f h)
```

The type signature *is* the theorem. The function body *is* the proof. Writing a program is constructing a proof.

Curry–Howard: More Examples

- $A \to A$ (identity): `fun x => x`
- $A \to B \to A$ (constant): `fun x y => x`
- Empty $\to A$ (ex falso): `fun x => absurd x`

Every one of these is, at the same time, a trivial program AND a theorem we just proved (remember: there is no difference between the two activities!).

Building a Proof, Step by Step (1)

Let's use the $A \times B \leftrightarrow A \wedge B$ row from three slides ago to build a proof by hand.

Claim: if you have A and B, you have B and A. Obvious to anyone: what term proves it?

```
theorem and_comm' {A B : Prop} (h : A ∧ B) : B ∧ A :=
  sorry
```

- `h : A ∧ B` is literally a *pair*: `h.1 : A` and `h.2 : B`
- **Hint**: what would let us swap the two halves of that pair?

Building a Proof, Step by Step (2)

Claim: if you have A and B, you have B and A.

```
theorem and_comm' {A B : Prop} (h : A ∧ B) : B ∧ A :=
  ⟨h.2, h.1⟩
```

- `⟨h.2, h.1⟩` builds a new pair (so B first, A second) straight from the "components" of `h`
- Lean's kernel accepts this term, no warnings
- **That acceptance is the entire claim of this talk**: a proof is nothing more than a term the type checker accepts

Type Inhabitation: Finding Witnesses

Recursive strategy (doesn't always terminate):

1. If $\sigma = \tau \to \upsilon$, try `fun x => _`
2. Look for a constant/variable that could produce σ
3. Build the term and recursively fill the remaining holes

Worked example: inhabit $(A \to B \to C) \to ((B \to A) \to B) \to A \to C$

```
def f : (A → B → C) → ((B → A) → B) → A → C :=
  fun p q a => p a (q (fun _ => a))
```

The payoff: by Church–Turing, Lean's term language is exactly as powerful as a Turing machine $\implies$ "does this type have an inhabitant" is exactly *as hard as the halting problem*. That's why inhabitation is undecidable.

Rules of Type Inference

Written as a fraction: if the premise (top) holds, the conclusion (bottom) holds. These are the rules Lean's kernel actually runs.

Variable:

$$\frac{}{C \vdash x : \sigma}\ \text{Var} \quad \text{if } x : \sigma \in C$$

Constant:

$$\frac{}{C \vdash c : \sigma}\ \text{Cst} \quad \text{if } c \text{ has type } \sigma$$

Application:

$$\frac{C \vdash t : \sigma \to \tau \quad C \vdash u : \sigma}{C \vdash t\, u : \tau}\ \text{App}$$

Abstraction:

$$\frac{C, x : \sigma \vdash t : \tau}{C \vdash (\lambda x : \sigma.\, t) : \sigma \to \tau}\ \text{Fun}$$

Four rules. Everything you'll see Lean check today reduces to these.

Type Judgments

Those fractions were all written in terms of one shape: **type judgment** $C \vdash t : \sigma$.

Read: "in context C, term t has type σ."

Context C: the local variables in scope and their types, $C = \{x_1 : \sigma_1,\ x_2 : \sigma_2,\ \ldots\}$, built up as you go, e.g. by the Fun rule.

Example:

$$\{x : \mathsf{Nat},\ y : \mathsf{Bool}\} \vdash x + 1 : \mathsf{Nat}$$

"Given a `Nat` named x and a `Bool` named y, the term $x + 1$ has type `Nat`." This is the exact notation the previous slide's fractions were built from.

Excluded Middle: Constructive vs. Classical

State $P \vee \neg P$. Sounds obvious, but *constructively*, to prove a disjunction you must *produce* one side and say which. Lean's core logic won't hand you $P \vee \neg P$ for free.

But it's not dogmatic about it, since you can just invoke it explicitly as an axiom:

```
open Classical in
example (P : Prop) : P ∨ ¬P := em P
```

Why it matters: every proof you write is honestly labelled, so you always know which steps are constructive (and therefore computable) and which ones borrowed a classical axiom.

Section 2

Lean in the Wild

From Terms to Tactics

Term-mode proofs are direct, but if done by hand, they get too complex very fast. Building a term for a real algebraic identity by chaining rewrite lemmas manually would be a nightmare.

Tactics: instead of writing the term directly, describe a *sequence of proof steps*, and Lean elaborates them into the term for you.

```
-- Term mode: you write the proof term yourself
theorem add_zero' (a : Nat) : a + 0 = a := rfl

-- Tactic mode: you describe steps, Lean builds the term
theorem mul_comm_ex (a b : Nat) : a * b = b * a := by
  rw [Nat.mul_comm]
```

Still an ITP, not black magic: every tactic call still produces a term underneath, and the kernel still checks it. Note that they don't bypass verification, tactics only automate *writing* the term.

Lean on Real Numbers

Curry–Howard isn't just for funny toy logic examples, it's literally how Lean proves "ordinary" algebra too.

```
example (a b c : ℝ) : a * (b * c) = b * (c * a) := by
  rw [mul_comm, mul_assoc]


example (a b : ℝ) : (a + b) * (a - b) = a ^ 2 - b ^ 2 := by
  rw [mul_sub, mul_comm, mul_add, mul_comm (a + b) b,
      mul_add, ← sub_sub, ← pow_two, ← pow_two,
      mul_comm b a, ← add_sub, sub_self, add_zero]
```

Each `rw` step is Lean checking a small equality proof, and so the kernel verifies every single rewrite.

Lean on Ring Axioms

Everything about a ring follows from the handful of axioms Lean already knows:

```
variable {R : Type*} [Ring R]
#check (add_assoc     : ∀ a b c : R, a + b + c = a + (b + c))
#check (add_comm      : ∀ a b : R,   a + b = b + a)
#check (zero_add      : ∀ a : R,     0 + a = a)
#check (neg_add_cancel: ∀ a : R,     -a + a = 0)
#check (mul_assoc     : ∀ a b c : R, a * b * c = a * (b * c))
#check (mul_add       : ∀ a b c : R, a * (b + c) = a * b + a * c)
```

Everything else, things like cancellation, $a \times 0 = 0$, double negation, etc. is *derived*, and is not assumed.

Let's derive one!

Let's Prove Something, Together (1)

Claim: $a \times 0 = 0$. This is *not* a ring axiom — we have to earn it.

We already have a helper lemma in hand: `add_left_cancel` (if $a + b = a + c$ then $b = c$). What do we do with it?

```
theorem mul_zero (a : R) : a * 0 = 0 := by
  sorry
```

Hint: What would let us cancel an `a * 0` from both sides?

Let's Prove Something, Together (2)

Claim: $a \times 0 = 0$. This is *not* a ring axiom (we have to "earn it"!).

We already have a helper lemma in hand: `add_left_cancel` (if $a + b = a + c$ then $b = c$). What do we do with it?

```
theorem mul_zero (a : R) : a * 0 = 0 := by
  have h : a * 0 + a * 0 = a * 0 + 0 := by
    rw [← mul_add, add_zero, add_zero]
  rw [add_left_cancel h]
```

Hint: What would let us cancel an `a * 0` from both sides?

Formalizing Real Mathematics

This same `rw` / `have` machinery, at scale, is currently "formalizing" (i.e. formally verifying) serious mathematics.

- **Mathlib**: Lean's community library (over **1.5 million lines** of code), algebra through category theory, all mechanically checked by Lean
- **Fermat's Last Theorem**: an active, ongoing Lean formalization project
- **Terence Tao** has publicly used and endorsed Lean, including a Lean-based collaborative proof of the Polynomial Freiman–Ruzsa conjecture

De Bruijn Criterion: Everything routes through the same tiny, hand-verifiable kernel, regardless of the thousands of lines of tactics which may be buggy. If the kernel accepts it, the theorem is true.

Section 3

The Substructural Zoo

Structural Rules in Logic

Structural rules govern how the hypothesis context Γ may be manipulated (i.e. meta-rules *about* reasoning, and not necessarily about specific propositions).

Weakening:

$$\frac{\Gamma \vdash A}{\Gamma, B \vdash A}$$

"You can *ignore* hypotheses." A variable may go unused.

Contraction:

$$\frac{\Gamma, A, A \vdash B}{\Gamma, A \vdash B}$$

"You can *duplicate* a hypothesis." A variable may be used more than once.

In Lean's core logic: **both rules hold freely**.
What happens if we take one away?

The Substructural Zoo

Selectively removing structural rules yields different *resource disciplines* (i.e. type systems that track *how many times* a value is used):

System	Weakening?	Contraction?	How many times?
Normal types	✓	✓	Any number
Affine types	✓	×	**At most once**
Relevant types	×	✓	**At least once**
Linear types	×	×	**Exactly once**

Each row corresponds to its own "logic", each with its own "Curry-Howard" reading.

Logic as a Language for Security Properties

Once “used exactly once” is a *type*, real-world safety rules become actual propositions you can state and ones Lean can check:

- **Affine**: a value can be dropped, never duplicated. This is exactly how Rust’s ownership system works, the being to turn use-after-free and double-free into type errors
- **Linear**: a value *must* be used exactly once, e.g. a file handle that must eventually close, a lock that must eventually release
- **Protocol state**: an action is only well-typed *after* a prerequisite step, e.g. “you cannot send data before connecting” or similar

Let’s see that last one, formalized.

Formalizing Software Security in Lean

A protocol state machine, where the type itself depends on the connection state:

```
inductive ConnectionState where
  | disconnected : ConnectionState
  | connected : ConnectionState

-- The TYPE of a connection depends on its state
@[reducible] def Connection : ConnectionState → Type :=
  fun state => match state with
    | .disconnected => Unit
    | .connected    => Nat

-- Can only call send on a CONNECTED connection --
-- enforced by the type, not by a runtime check
def send (conn : Connection .connected) (msg : String) : IO Unit :=
  IO.println s!"[handle {conn}] sending: {msg}"
```

The security property “never send before connecting”, which used to be a convention, is now a compile-time type error to throw.

Section 4

Closing

Summary

- A type is a specification; a term is a witness that it can be satisfied
- **Curry–Howard**: propositions are types, proofs are programs, type-checking is proof-checking
- Lean's logic machinery also proves algebra, and scales to large-scale research mathematics (Mathlib, FLT, Tao, etc.)
- Removing structural rules gives a *family* of "logics", with each being able to express a different (real-world) safety property, all mechanically checked

Bottom line: Types are a language for modeling reality, proving theorems, and guaranteeing software security.

Questions & Discussion

Questions? Reach out to me or join our community through:

Signal: `danieldia.25519`
PROOF101 Website: `https://danieldia.me/proofs`
Email: `dmd13@mail.aub.edu`

Explore the Tools

Try it yourself — all three are free.

Lean 4

lean-lang.org

Mathlib

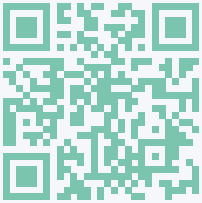
mathlib-initiative.org

Natural Number Game

adam.math.hhu.de

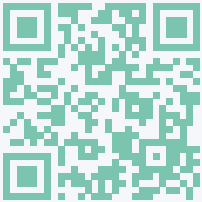
Keep Going

Two more places to keep this going.



PROOF101 — Spring 2027

open to all students



These slides

`danieldia.me`

"The only way to learn mathematics is to do mathematics."

— Paul Halmos

Lebanese Math Day: Proofs are Programs

How Types Let Us Prove Theorems and Secure Software

Daniel Dia
American University of Beirut (AUB)
Dep. of Electrical & Computer Engineering
https://danieldia.me/