Type Theory Basics for Lean

Malvin Gattinger

31 March 2026, Weekly Lean Seminar Amsterdam

Outline

Overview

Warm-up proof

Want to follow along? Get `https://malv.in/2026/cwi-talk.lean` and copy it to `https://live.lean-lang.org`

```
-- Let's prove this using tactics
theorem combine {n b s : Prop} :
    (n → b) → (n → s) → (n → (b ∧ s)) := by
  sorry
```

Overview

Simple Types

```
-- Primitive types:
#check Bool
#check Nat
#check String
#check Type

-- Function types:
#check Nat → Bool

-- Pairs:
#check Bool × String
#check (true, "hello")
```

Programming with Types

```
def odd : Nat → Bool
  | 0 => false
  | 1 => true
  | k+2 => odd k

def count : Nat → String
  | 0 => ""
  | k+1 => "X" ++ count k
```

Programming with Types

```
def odd : Nat → Bool
  | 0 => false
  | 1 => true
  | k+2 => odd k

def count : Nat → String
  | 0 => ""
  | k+1 => "X" ++ count k

def combo :
  (Nat → Bool) → (Nat → String) → Nat → (Bool × String)
  | f, g => fun b => ⟨f b, g b⟩

#eval combo odd count 6
```

Type Theory

If `f` has the type `a → b`,
and `x` has the type `a`,
then `f x` has the type `b`.

Type Theory

If `f` has the type `a → b`,
and `x` has the type `a`,
then `f x` has the type `b`.

Formally, we write the Application rule like this:

$$\frac{\vdash f : a \to b \qquad \vdash x : a}{\vdash f\ x : b}\ \textsc{App}$$

Simply Typed Lambda Calculus

We may also have a context Γ.

Application:

$$\frac{\Gamma \vdash f : a \to b \qquad \Gamma \vdash x : a}{\Gamma \vdash f\ x : b}\ \textsc{App}$$

Simply Typed Lambda Calculus

We may also have a context Γ.

Application:

$$\frac{\Gamma \vdash f : a \to b \qquad \Gamma \vdash x : a}{\Gamma \vdash f\ x : b}\ \textsc{App}$$

Variable:

$$\frac{}{\Gamma, x : a \vdash x : a}\ \textsc{Var}$$

Simply Typed Lambda Calculus

We may also have a context Γ.

Application:

$$\frac{\Gamma \vdash f : a \to b \qquad \Gamma \vdash x : a}{\Gamma \vdash f\ x : b}\ \textsc{App}$$

Variable:

$$\frac{}{\Gamma, x : a \vdash x : a}\ \textsc{Var}$$

Abstraction:

$$\frac{\Gamma, x : a \vdash M : b}{\Gamma \vdash \lambda x{:}a.\, M : a \to b}\ \textsc{Abs}$$

Simply Typed Lambda Calculus

We may also have a context Γ.

Application:

$$\frac{\Gamma \vdash f : a \to b \qquad \Gamma \vdash x : a}{\Gamma \vdash f\ x : b}\ \textsc{App}$$

Variable:

$$\frac{}{\Gamma, x : a \vdash x : a}\ \textsc{Var}$$

Abstraction:

$$\frac{\Gamma, x : a \vdash M : b}{\Gamma \vdash \lambda x{:}a.\, M : a \to b}\ \textsc{Abs}$$

The good news: You do not need to know these rules.
But Lean uses them to check what you wrote.

Overview

Let's prove it again

```
-- Let's prove it again, but with terms!
theorem combine' {b n s : Prop} :
    (n → b) → (n → s) → (n → (b ∧ s)) := sorry
```

It's the same!

Did you ever feel like writing a program is similar to giving a proof?

> *"Every good idea will be invented twice."*
>
> Phil Wadler

Propositions as Types

- To prove $p \to q$, we assume am a proof of p to then prove q.
  To make a value of type `a -> b`, we make a function from type `a` to type `b`.
- To prove $p \wedge q$, we need to prove p and prove q.
  To make a value of type `(a,b)`, we make a value of type `a` and a value of type `b`.
- To prove $p \vee q$, we either provide a proof of p or a proof of q.
  To make a value of type `Sum a b` we make a value of type `a` or a value of type `b`.

This is the Curry–Howard correspondence relating logics and types.

Propositions correspond to types, and proofs to programs.

Type Theory = Logic = Programming

Prove that the following is valid in propositional logic:

$$(p \to q) \to ((q \to r) \to (p \to r))$$

Type Theory = Logic = Programming

Prove that the following is valid in propositional logic:

$$(p \to q) \to ((q \to r) \to (p \to r))$$

Find a lambda calculus expression of this type:

$$(\pi \to \rho) \to ((\rho \to \gamma) \to (\pi \to \gamma))$$

Type Theory = Logic = Programming

Prove that the following is valid in propositional logic:

$$(p \to q) \to ((q \to r) \to (p \to r))$$

Find a lambda calculus expression of this type:

$$(\pi \to \rho) \to ((\rho \to \gamma) \to (\pi \to \gamma))$$

Write a program of the type:

```
(a -> b) -> ((b -> c) -> (a -> c))
```

. . .

It's the same question! Finding a term is the same as finding a proof.

. . .

In fact, having the term is enough!

Propositions as Types Overview

Logic	Programming
Proposition	Type
Proof	Term
checking a proof	type checking
finding a proof	finding a term
atomic propositions	type variables
implication	functions
conjunction	pairs
disjunction	sum types
quantifiers	dependent types
provable / valid	inhabited

Overview

Dependent Types: Motivation for Programming

```
def mult : List Nat → List Nat → List Nat
  | [], [] => []
  | (x::xs), (y::ys) => x * y :: mult xs ys
```

Dependent Types: Motivation for Programming

```
def mult : List Nat → List Nat → List Nat
  | [], [] => []
  | (x::xs), (y::ys) => x * y :: mult xs ys
```

How to tell Lean that both lists must have the same length?

Solution: a dependent type!

```
def mult' :
  {k : Nat} → Vector Nat k → Vector Nat k → Vector Nat k
  | 0,   ⟨⟨[]⟩   ,h1⟩, ⟨⟨[]⟩   ,_ ⟩ => ⟨⟨[]⟩,h1⟩
  | k+1, ⟨⟨x::xs⟩,h1⟩, ⟨⟨y::ys⟩,h2⟩ =>
    let rest := @mult' k ⟨⟨xs⟩, by grind⟩ ⟨⟨ys⟩, by grind⟩
    ⟨⟨x*y :: rest.1.1⟩ , by grind⟩
```

Even better:

```
def mult''
  {k : Nat} : Vector Nat k → Vector Nat k → Vector Nat k
  := Vector.zipWith (· * ·)
```

Dependent Types: Motivation for Proving

How do you prove $\exists x, Px$ and $\forall x, Px$?

What type should correspond to quantifiers?

Dependent Types: Motivation for Proving

How do you prove $\exists x, Px$ and $\forall x, Px$?

What type should correspond to quantifiers?

Let's prove this, using dependent functions and pairs!

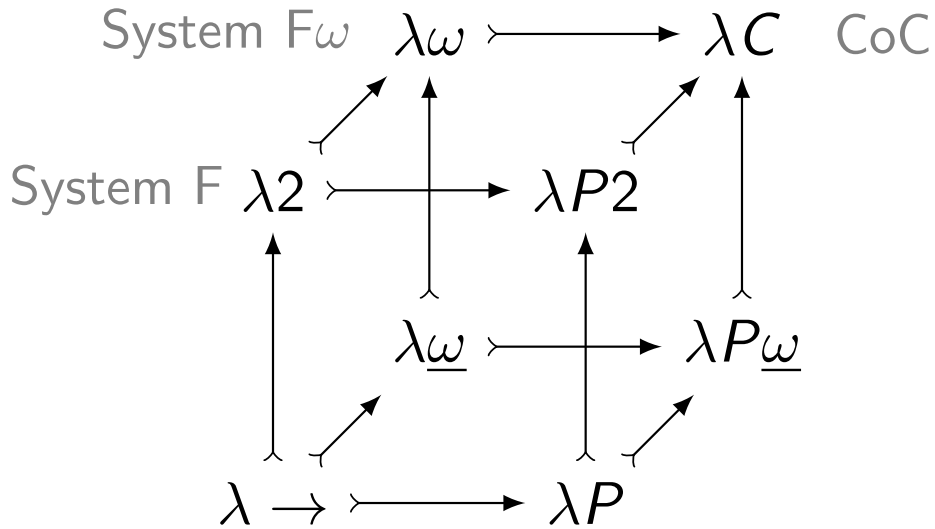
```
theorem qExample {a : Type} {P Q : a → Prop}:
    (∀ (x : a), P x → Q x) → (∃ (z : a), P z) → ∃ z, Q z := by
  sorry
```

The Lambda Cube

Multiple ways to extend the simply-typed lambda calculus ($\lambda \rightarrow$).

The Lambda Cube

Multiple ways to extend the simply-typed lambda calculus ($\lambda \rightarrow$).



The Lambda cube, by Henk Barendregt (1991).

- go right: types of terms — dependent types: `Vector Nat 3`
- go up: terms of types — polymorphism: `1`, `+`, $\top$
- go diagonal: types of types — type operators: `Option`