Introduction to Lean and propositions-as-types

Jack Valmadre, 9 Feb 2023

What is Lean? What is it for?

Language for mathematicians to *write* and *mechanically check* proofs

Functional and strongly typed: Uses *type system* for proof verification

Why am I interested in it?

Fun! Learn some type theory, think more mathematically

Potential to build AI for theorem proving

Formal verification for programs written by Codex (Copilot), ChatGPT, ...

Why is proof verification important?

Huge pressure to be first

Reputation damage for large, incorrect claims

Faults in complex proofs can go unnoticed

Story 1: Fermat's last theorem (no integer solutions to $a^n + b^n = c^n$ for $n \geq 3$)

Conjectured 1637, proof published 1993, problem found 1994, final proof 1995

Story 2: "∞-groupoids as a model for a homotopy category"

Proof published 1989, problem reported 1998, ignored until 2013

Outline

Language introduction

Idea 1: Propositions-as-types

Idea 2: False and not-statements

Idea 3: Dependent types

Writing practical proofs

Simple types

Today, 2 types: natural numbers and propositions

```
#check 1          -- 1 : ℕ
#check 1 < 2      -- 1 < 2 : Prop
#check 1 = 2      -- 1 = 2 : Prop

variable x : ℕ
#check x + 2      -- x + 2 : ℕ
#check x = 2      -- x = 2 : Prop
```

Functions

Functions defined using *def* or *lambda*

Function type denoted with *arrow*

```
def add_one (x : ℕ) := x + 1
def add_one := λ (x : ℕ), x + 1   -- Equivalent

#check add_one       -- add_one : ℕ → ℕ
#check add_one 5     -- add_one 5 : ℕ
#eval add_one 5      -- 6
```

Currying

Functions with multiple arguments use *currying*

Think: *functools.partial()* in Python, or *std::bind()* in C++

```
def add (x y : ℕ) := x + y
def add := λ (x : ℕ), λ (y : ℕ), x + y    -- Equivalent

#check add             -- add : ℕ → ℕ → ℕ
#check add 1 5         -- add 1 5 : ℕ
#eval add 1 5          -- 6

#check add 1           -- add 1 : ℕ → ℕ
#check (add 1) 5       -- add 1 5 : ℕ
```

Idea 1: Propositions-as-types

Treat propositions as types, then...

Proof of proposition is *instance* of type

Function `f : p → q` maps *proof* of P to *proof* of Q

Definition of function with type `p → q` is a *proof* that P implies Q

Defining *function* is proving *implication*

Verifying proof is *type-checking* of program

Curry-Howard correspondence ("proofs as programs, programs as proofs")

A first theorem

Example: Show that Q follows from P and P → Q (*modus ponens*)

```
theorem thm (p q : Prop) : p → (p → q) → q :=
  assume hp : p,
  assume hpq : p → q,
  show q, from hpq hp
```

just syntactic sugar for...

```
theorem thm (p q : Prop) : p → (p → q) → q :=
  λ hp : p,
  λ hpq : p → q,
  hpq hp
```

Idea 2: False and not-statements

`false` is the empty type: there *does not exist a proof*

`¬p` is defined as a *function* type `p → false`

Interpretation: There does not exist a proof of P, since the set of proofs is mapped to the empty set, and only the empty set maps to the empty set

A second theorem

Prove that P implies ¬¬P (not not P)

```
-- Following statements equivalent by definition:
-- p → ¬¬p
-- p → ¬p → false
-- p → (p → false) → false

theorem implies_not_not (p : Prop) : p → ¬¬p :=
  assume hp : p,
  assume hnp : p → false,
  show false, from hnp hp
```

Constructive logic

What about the other direction... does ¬¬P imply P?

Not in general!

Requires law of excluded middle (LEM): *every statement is either true or false*

Gödel: There exist statements P such that both P and ¬P have no proof

Lean uses *constructive* logic rather than *classical* logic: *does not assume LEM**

*can use LEM for *decidable* statements; has option to enable classical logic

Idea 3: Dependent types

Used to define *function types* that depend on values

If `f` is a function that maps a value `x` to a value `y` of *type* `(t x)`,
then the *type* of `f` is expressed: `Π (x : s), (t x)`

```
-- Example: Map value to length-1 list of value.
def to_list (s : Type) (x : s) := [x]

#eval to_list ℕ 5   -- [5]
#check to_list ℕ    -- to_list ℕ : ℕ → list ℕ

#check to_list      -- to_list : Π (s : Type), s → list s
```

Dependent types and propositions

Dependent types that return *propositions* are logical *for-all* statements

```
#check Π (x : ℕ), x > 0    -- ∀ (x : ℕ), x > 0 : Prop
```

Interpretation: *Instance* of the dependent type is a *function* that maps *any value* to a *proof* of the *value-dependent proposition* (*instance* of the *value-dependent type*)

Theorems are often instances of a *dependent* proposition type (for-all statement):

```
theorem thm (p q : Prop) : p → (p → q) → q := ...

#check thm   -- thm : ∀ (p q : Prop), (p → q) → p → q
```

Practical proofs using tactics

Proof tactics define operations to manipulate expressions

Implemented as *meta-programming*, uses monad for tactic state

```
theorem thm (x y : ℤ) : (x+y) * (x-y) = x*x - y*y :=
begin
  rw mul_sub,
  repeat {rw mul_comm (x + y)},
  repeat {rw mul_add},
  rw ← sub_sub,
  simp [mul_comm, sub_self],
end
```

```
#print thm
```

```
theorem thm : ∀ (x y : ℤ), (x + y) * (x - y) = x * x - y * y :=
λ (x y : ℤ),
  (id_tag tactic.id_tag.rw (eq.rec (eq.refl ((x + y) * (x - y) = x * x - y * y)) (mul_sub (x + y) x y))).mpr
    ((id_tag tactic.id_tag.rw (eq.rec (eq.refl ((x + y) * x - (x + y) * y = x * x - y * y)) (mul_comm (x + y) x))).mpr
       ((id_tag tactic.id_tag.rw
           (eq.rec (eq.refl (x * (x + y) - (x + y) * y = x * x - y * y)) (mul_comm (x + y) y))).mpr
          ((id_tag tactic.id_tag.rw (eq.rec (eq.refl (x * (x + y) - y * (x + y) = x * x - y * y)) (mul_add x x y))).mpr
             ((id_tag tactic.id_tag.rw
                 (eq.rec (eq.refl (x * x + x * y - y * (x + y) = x * x - y * y)) (mul_add y x y))).mpr
                ((id_tag tactic.id_tag.rw
                    (eq.rec (eq.refl (x * x + x * y - (y * x + y * y) = x * x - y * y))
                       (sub_sub (x * x + x * y) (y * x) (y * y)).symm)).mpr
                   ((id_tag tactic.id_tag.simp
                       ((((λ (a a_1 : ℤ) (e_1 : a = a_1) (ă ă_1 : ℤ) (e_2 : ă = ă_1),
                             congr (congr_arg eq e_1) e_2)
                           (x * x + x * y - y * x - y * y)
                           (x * x - y * y)
                           ((λ [self : has_sub ℤ] (ă ă_1 : ℤ) (e_2 : ă = ă_1) (ă_2 ă_3 : ℤ)
                             (e_3 : ă_2 = ă_3), congr (congr_arg has_sub.sub e_2) e_3)
                              (x * x + x * y - y * x)
                              (x * x)
                              (((λ [self : has_sub ℤ] (ă ă_1 : ℤ) (e_2 : ă = ă_1) (ă_2 ă_3 : ℤ)
                                 (e_3 : ă_2 = ă_3), congr (congr_arg has_sub.sub e_2) e_3)
                                  (x * x + x * y)
                                  (x * x + x * y)
                                  (eq.refl (x * x + x * y))
                                  (y * x)
                                  (x * y)
                                  (mul_comm y x)).trans
                                 (add_sub_cancel (x * x) (x * y)))
                              (y * y)
                              (y * y)
                              (eq.refl (y * y)))
                           (x * x - y * y)
                           (x * x - y * y)
                           (eq.refl (x * x - y * y))).trans
                          (propext sub_left_inj)).trans
                         (propext (eq_self_iff_true (x * x)))))).mpr
                      trivial))))))
```

Review

Idea 1: Propositions-as-types (proof = instance, implication = function)

Idea 2: False and not-statements (empty type, map to empty type)

Idea 3: Dependent types (function type depends on value, for-all)

Tutorials and talks

Interactive tutorial: Natural Number Game (tactic mode)

Tutorial: Theorem proving in Lean

Workshop: Lean for the Curious Mathematician 2022 (with recordings)

Talk: Kevin Buzzard “The Future of Mathematics?”

Talk: Vladimir Voevodsky “Univalent Foundations”

Episodes of CoRecursive podcast: Edwin Brady, Bartosz Milewski, Niki Vazou