Kris Zhang
Bowen Dai
Mentor: Sita Gakkhar

Theorem Proving in Lean 4

Outline

What is Lean 4?

- A modern interactive theorem prover developed by Microsoft Research.
- Combines functional programming + dependent type theory.
- Ensures proofs are mechanically verified.

Why Theorem Proving?

- Guaranties mathematical correctness through machine-checked proofs.
- Removes ambiguity and hidden steps in handwritten arguments.
- Finding new proofs by using machines and enabling automotive proofs.
- Proposition are types, terms are proofs.

Types, Terms, and Goal

- **Types** classify data and mathematical objects (e.g. `Nat`, `Real`, `Fin n → R`).
- **Terms** are values inhabiting a type (e.g. `(fun i : Fin 3 => i.val)`).
- **Expressions** combine terms to produce new terms.
- **Goal** a statement or proposition that the user is trying to prove or a term of a specific type that needs to be constructed

How Lean Checks Proofs

- Finding a proof means a construction of that type, and the programs are a collection of types.

Example

```
-- Goal: Prove that `P → P` is true
-- The proof starts with a single goal: ` P → P`
-- `by` keyword enters tactic mode

by
  -- The `intro` tactic takes a proposition `P` from the hypotheses
  intro P -- This transforms the goal into `P  P`
  -- The goal now is to prove `P` given the hypothesis `P`

  -- `exact` fills the hole with a known value.
  exact P -- This tactic fills the goal by using the hypothesis `P`.
          -- The proof is now complete.
```

Essential Tactics

- `intro` — introduce assumptions and parameters.
- `apply` — reduce the goal using a lemma/theorem.
- `rw` — rewrite using an equality.
- `simp` — simplify expressions via rewriting and algebraic rules.
- `induction` — perform inductive proofs on `Nat` and inductive types.
- For linear algebra, tactics like `simp`, `rw`, and `linarith` are crucial for manipulating inequalities.

Example: A Simple Proof

Goal

Prove: $n + 0 = n$

```
theorem add_zero (n : Nat) : n + 0 = n := by
  induction n with
  | zero =>
      simp
  | succ k ih =>
      rw [Nat.add_succ, ih]
```

Example: A Simple Proof

```
Goal at case zero

        case zero
             0 + 0 = 0


Goal at case succ k

        case succ
        k : Nat
        ih : k + 0 = k
             k + 1 + 0 = k + 1
```

Lemmas and Theorems

- Lean organizes proofs into `lemma` and `theorem` declarations.
- These are reusable components: later proofs can simply `apply` them.
- As a example, in this project I added:
  - a dot product on `Fin n → R`:

```
def dot {n : N} (x y : Fin n → R) : R :=
  i, x i * y i
```

  - feasibility predicates `PrimalFeasible` and `DualFeasible`;
  - helper lemmas comparing two dot products componentwise.

Equational Reasoning and Typeclasses

- Lean supports chained equational reasoning with `calc` blocks.
- Typeclasses capture algebraic structure: ordered fields, modules, matrices, etc.
- For linear algebra, `Matrix`, `Fintype`, and `Module` instances make operations like `mulVec` and finite sums over `Finset.univ` (union of finite set) work smoothly.

Example

```
calc
  a + b = b + a := by
    simpa [Nat.add_comm]
```

Weak Duality: Statement

Weak Duality Theorem

For any primal feasible x and dual feasible y:

$$c^\top x \leq b^\top y.$$

- The primal is $\max\{c^\top x \mid x \geq 0,\ Ax \leq b\}$.
- The dual is $\min\{b^\top y \mid y \geq 0,\ A^\top y \geq c\}$.
- Weak duality says: every dual objective value is an upper bound on every primal value.
- This is the starting point for strong duality and LP sensitivity analysis.

High-Level Proof Idea

- Start with primal feasibility: $Ax \leq b$ and $x \geq 0$.
- Dual feasibility: $A^\top y \geq c$ and $y \geq 0$.
- Multiply by nonnegative vectors:

$$y^\top Ax \leq y^\top b, \qquad c^\top x \leq y^\top Ax.$$

- Combine the inequalities to obtain:

$$c^\top x \leq b^\top y.$$

- In Lean, each of these steps is implemented as a lemma about dot products and componentwise inequalities on functions `Fin n → R`.

Lean Formalization of Weak Duality

- Vectors: `Fin n → R`, matrices: `Matrix (Fin m) (Fin n) R`.
- `PrimalFeasible` and `DualFeasible` encode all linear constraints.
- Key helper lemmas:
  - `dot_le_of_le_componentwise` (monotonicity of dot product);
  - `y_mulAx_le_yb` for $y^\top Ax \le y^\top b$;
  - `x_mulATy_ge_cx` for $c^\top x \le x^\top A^\top y$.
- The only missing algebraic step is a symmetry identity relating $(A^\top y, x)$ and (Ax, y), which I currently leave as a `sorry` (a placeholder for a proof I plan to fill in later), In Lean, `sorry` is a special keyword that means "assume this goal is solvable and move on": it keeps the file compiling during development, but it marks an unfinished proof that must be removed in a final, fully verified development.

Weak duality in Lean (code only)

Weak duality in Lean (simplified)

```
theorem weak_duality {m n : N}
  (A : Matrix (Fin m) (Fin n) R)
  (b : Fin m → R)
  (c : Fin n → R)
  (x : Fin n → R)
  (y : Fin m → R)
  (hx : PrimalFeasible A b x)
  (hy : DualFeasible A c y) :
  dot c x  dot b y := by
  -- use feasibility to get:
  have h1 : dot (A.mulVec x) y <= dot b y := ...
  have h2 : dot c x <= dot (A.mulVec y) x := ...
  -- matrix transpose identity (work in progress):
  have htranspose :
      dot (A.mulVec y) x = dot (A.mulVec x) y := by
    sorry
  -- combine them with `le_trans` after htranspose
  ...
```

Statement of the Theorem

Fermat's Little Theorem (classical):

$$\text{If } p \text{ is prime and } p \nmid a, \quad \text{then} \quad a^{p-1} \equiv 1 \pmod{p}.$$

In Lean, we express this using natural numbers and modulo:

```
theorem fermat_little_theorem
  {p : ℕ} (hp : p.Prime) (a : ℕ) (h : p ∣ a → False) :
  a^(p-1) % p = 1
```

High-Level Proof Idea

The mathematical strategy:

1. Work in the ring $\mathbb{Z}/p\mathbb{Z}$ where arithmetic mod p is built in.
2. Use the assumption $p \nmid a$ to show that a is nonzero in $\mathbb{Z}/p\mathbb{Z}$.
3. A nonzero element of a field is a unit, so $(a : \mathbb{Z}/p\mathbb{Z})$ is a unit.
4. Units in $\mathbb{Z}/p\mathbb{Z}$ correspond exactly to naturals coprime to p.
5. Apply Euler's theorem: $a^{\varphi(p)} \equiv 1 \pmod{p}$ for $\gcd(a, p) = 1$.
6. For prime p, $\varphi(p) = p - 1$.

Lean carries out exactly this program.

Full Lean Proof

```
import Mathlib
theorem fermat_little_theorem
  {p : N} (hp : p.Prime) (a : N) (h : p ∣ a -> False) :
  a^(p-1) % p = 1 := by
  have hne : (a : ZMod p) =/= 0 := by
    intro h
    have : p ∣ a := (ZMod.natCast_eq_zero_iff a p).mp h
    exact h this

  haveI : Fact p.Prime := ⟨hp⟩
  have h_unit : IsUnit (a : ZMod p) := IsUnit.mk0 (a : ZMod p) hne
  have h_copr : a.Coprime p :=
    (ZMod.isUnit_iff_coprime a p).1 h_unit
  have h := Nat.pow_totient_mod_eq_one (hp.one_lt) h_copr
  have h_eq : p.totient = p - 1 := Nat.totient_prime hp
  rwa [h_eq] at h
```

Explanation of the Tactics

Key Lean constructions used:

- `have hne : (a : ZMod p) =/= 0` Introduces a lemma showing $(a : \text{ZMod } p) \neq 0$. Uses the equivalence `ZMod.natCast_eq_zero_iff`.
- `haveI : Fact p.Prime := ⟨hp⟩` use the proof hp : p.Prime as a typeclass instance so Lean can automatically use the fact that p is prime in later lemmas.
- `IsUnit.mk0 (a : ZMod p) hne` Constructs a unit from a non-zero element of a field.
- `(ZMod.isUnit_iff_coprime a p).1 h_unit`
  Converts a unit in `ZMod p` back to a coprimeness condition on naturals.
- `Nat.pow_totient_mod_eq_one`
  Euler's theorem for coprime natural numbers.
- `Nat.totient_prime hp`
  Rewrites $\varphi(p)$ into $p - 1$.

Functionalities of Lean Used in this proof

This example highlights several powerful features of Lean:

- **Type-driven mathematics**: Switching from $\mathbb{N}$ to `ZMod p` allows modular arithmetic structures to be used automatically.
- **Typeclass inference**: Lean automatically retrieves algebraic facts (e.g., that `ZMod p` is a field when p is prime).
- **Library integration**: Mathlib provides standard number theory results (`totient_prime`, Euler's theorem, etc.).
- **Structured proofs**: Small lemmas (`hne`, `h_unit`, `h_copr`) guide Lean through the reasoning steps.

Conclusion

`Formalizing and verifying mathematics` - creating machine-checked proofs and contributing to large libraries like mathlib.

- `High-assurance software and hardware verification` - proving correctness of algorithms, protocols, compilers, and system components.
- `Automated discovery of proof steps` - Lean's tactic framework and automation tools (e.g., simp, rw) can automatically fill in routine or complex proof steps.