

Introduction to Mathematical Formalization

Chenyi Li

**School of Mathematical Sciences
Peking University**

Outline

1. Mathematical Formalization
2. Lean Statements and Proofs
3. Formalizing Mathematical Optimization
4. Theorem Search and LLM-Based Proving

- **Fermat's Last Theorem** (September 2026)
 - Anthropic released a complete Lean formalization produced by Claude.
- **The three-dimensional Kakeya conjecture** (September 2026)
 - AMSS, Project Numina, Nankai University, and ByteDance Seed completed a Lean formalization of the three-dimensional Hausdorff-dimension result.
- **Navier–Stokes equations** (September 2026)
 - OpenAI released an AI-generated proof and Lean formalization of finite-time breakdown in three dimensions with smooth external forcing.
- **OpenAI's mathematics release** (October 2026)
 - OpenAI released a collection of mathematical manuscripts, including Lean formalizations for many of the results.

Formalization makes a supplied proof checkable.

$$P : \text{Prop}, \quad t : P.$$

- The user specifies a formal **statement** P and submits a candidate **proof** t .
- Lean's kernel checks whether t is a correct proof of P .
- The checker does not automatically discover a proof of P .
- The user must also ensure that P expresses the intended mathematical claim.

Lean: an interactive theorem prover

- Based on dependent type theory.
- **Formal proofs:** mathematical arguments expressed in a language that Lean can check for logical correctness.
- **Interactive proving:** the editor displays the current assumptions and goals.

Mathlib4: a library of formalized mathematics

- Built by an active open-source community.
- **428,312 declarations** as of October 9, 2026.



Statements and Proofs in Lean

```
theorem Convex_first_order_condition {s : Set E}
  (h : HasFDerivAt f (f' x) x) (hf : ConvexOn ℝ s f) (xs : x ∈ s) : Problem Statement
  ∀ y ∈ s, f x + f' x (y - x) ≤ f y := by
  intro y ys
  have hpos : ∀ t, (t > 0) ∧ t < 1 → f y - f x ≥ t⁻¹ * (f (x + t * (y - x)) - f x) := by
    rintro t (ht1, ht2); field_simp
    obtain hf2 := hf.2 xs ys (LT.lt.le (sub_pos.mpr ht2)) le_of_lt ht1 (by simp)
    rw [div_le_iff, ht1]; simp at hf2
    have : x + t * (y - x) = (1 - t) * x + t * y := by
      rw [smul_sub, sub_smul, one_smul, add_sub, add_sub_right_comm]
    rw [this]
    linarith
  have : Tendsto (fun t ↦ t⁻¹ * (f (x + t * (y - x)) - f x)) (ℕ[>] 0) (ℕ (f' x (y - x))) := by
    apply Filter.Tendsto.mono_left (Fderiv_line_deriv h (y - x))
    exact nhds_right'.le_nhds_ne 0
  have : f y - f x ≥ f' x (y - x) := by
    apply le_of_tendsto this
    rw [eventually_iff_exists_mem]
    use Ioo 0 1
    constructor
    · apply Ioo_mem_nhdsWithin_Ioi (by simp)
    intro t1 ht1; exact hpos t1 ht1
  linarith
```

Premises

Formal Proof

Tactics

- **Statement:** objects, hypotheses, conclusion.
- **Proof:** typed term establishing the statement.
- **Tactics:** commands that construct the proof.
- **Kernel:** checks that the proof term has the type specified in the theorem statement.

Example: convexity and derivative data imply a first-order inequality.

LLMs and Lean Feedback

- **Goal:** prove a smoothness inequality from derivative and Lipschitz assumptions.
- **Lean error:** the requested rewrite pattern does not occur in the target.
- **Next step:** find a theorem that connects these assumptions to the required bound.

```
▼ Lsmooth.lean:56:47
▼ Tactic state
1 goal
  E : Type u_1
  inst¹ : NormedAddCommGroup E
  inst² : NormedSpace ℝ E
  f : E → ℝ
  f' : E → E →L[ℝ] ℝ
  l : NNReal
  hd : ∀ (x : E), HasFDerivAt f (f' x) x
  hl : LipschitzWith l f'
  x y : E
  ⊢ f y ≤ f x + (f' x) (y - x) + l / 2 * ‖y - x‖ ^ 2

▼ Messages (0/1)
▼ Lsmooth.lean:56:17
tactic 'rewrite' failed, did not find instance of the
pattern in the target expression
  LipschitzWith ?m.3619 ?m.3618
  E : Type u_1
  inst¹ : NormedAddCommGroup E
  inst² : NormedSpace ℝ E
  f : E → ℝ
  f' : E → E →L[ℝ] ℝ
  l : NNReal
  hd : ∀ (x : E), HasFDerivAt f (f' x) x
  hl : LipschitzWith l f'
  x y : E
  ⊢ f y ≤ f x + (f' x) (y - x) + l / 2 * ‖y - x‖ ^ 2
```

An LLM proposes proof steps. Lean checks each step and reports the remaining goals.

Outline

1. Mathematical Formalization
- 2. Lean Statements and Proofs**
3. Formalizing Mathematical Optimization
4. Theorem Search and LLM-Based Proving

An Example: Infinitely Many Primes

```
theorem infinite_primes (n : ℕ) :  
  ∃ p, p.Prime ∧ p > n := by  
  -- Proof
```

$$\forall n \in \mathbb{N}, \quad \exists p \in \mathbb{N}, \quad p \text{ is prime and } p > n.$$

- $(n : \mathbb{N})$ introduces an arbitrary natural number.
- $\exists p$ requires a natural-number witness p .
- $p.\text{Prime} \wedge p > n$ requires proofs of both properties.

Terms and Types

```
#check 1  
#check (1 : ℝ)  
#check Real.sqrt 2 -- Real.sqrt 2 : ℝ
```

Term	Type
1	$\mathbb{N}$
(1 : $\mathbb{R}$)	$\mathbb{R}$
Real.sqrt 2	$\mathbb{R}$

- $t : T$ means that the term t has type T .
- The annotation $(1 : \mathbb{R})$ tells Lean to interpret 1 as a real number.

Definitions

```
def myNumber : Nat := 42  
def double (n : Nat) : Nat := n * 2
```

- `def` introduces a named definition.
- `myNumber : Nat := 42` defines a natural-number constant.
- `double` has input `n : Nat`, output type `Nat`, and body `n * 2`.

Function Types

```
def f (x : ℕ) := x + 3
#check f      -- f : ℕ → ℕ
```

$$f: \mathbb{N} \rightarrow \mathbb{N}, \quad f(x) = x + 3.$$

- $\mathbb{N} \rightarrow \mathbb{N}$ specifies the input and output types.
- Lean infers the output type from the expression $x + 3$.
- `f 5` is Lean's notation for the mathematical expression $f(5)$.

Types of Types

```
#check Nat  
#check Type
```

- `Nat : Type` means that `Nat` is itself a type.
- `Type : Type` places types in a hierarchy of universes.
- A variable such as `G : Type*` can denote an arbitrary type.

Checking and Evaluating Expressions

```
#check myNumber    -- myNumber :  $\mathbb{N}$   
#check double      -- double :  $\mathbb{N} \rightarrow \mathbb{N}$   
  
#eval double 5     -- 10
```

Command	Result
#check myNumber	myNumber : $\mathbb{N}$
#check double	double : $\mathbb{N} \rightarrow \mathbb{N}$
#eval double 5	10

- #check reports a type.
- #eval computes the value of an executable expression.

Propositions

```
#check 2 + 2 = 4          -- Prop
#check ∀ n : ℕ, n + 0 = n -- Prop
```

- Prop is the type of propositions.
- $2 + 2 = 4$ is an equality proposition.
- $\forall n : \mathbb{N}, n + 0 = n$ is a universally quantified proposition.

Quantifiers and Hypotheses

Lean expression	Meaning
$\forall n : \mathbb{N}, P\ n$	$P(n)$ holds for every natural number n
$\exists p : \mathbb{N}, P\ p$	some natural number p satisfies $P(p)$
$P \wedge Q$	both propositions hold
$P \rightarrow Q$	a proof of P gives a proof of Q
$x \neq 0$	x is nonzero

- A universally quantified statement must cover every allowed input.
- An existential statement requires a witness and a proof of its property.

Defining a Proposition

```
def myFermatLastTheorem : Prop :=  
  ∀ x y z n : ℕ, n > 2 ∧ x * y * z ≠ 0 → x ^ n + y ^ n ≠ z ^ n  
  
#check myFermatLastTheorem  -- Prop
```

For natural numbers x, y, z, n ,

$$n > 2 \text{ and } xyz \neq 0 \implies x^n + y^n \neq z^n.$$

- `def ... : Prop` gives a name to this proposition.
- The hypotheses are $n > 2$ and $xyz \neq 0$. The conclusion is $x^n + y^n \neq z^n$.
- Defining the proposition leaves its proof as a separate task.

Proof Terms

```
theorem easy : 2 + 2 = 4 := rfl
#check easy    -- easy : 2 + 2 = 4
```

$$P : \text{Prop}, \quad h : P.$$

- $P : \text{Prop}$ declares a proposition. A term $h : P$ is a proof of P .
- `easy` names a proof of the proposition $2 + 2 = 4$.
- `rfl` proves this equality because both sides reduce to 4.

Unfinished Proofs

```
theorem hard : myFermatLastTheorem := sorry  
#check hard    -- hard : myFermatLastTheorem
```

- sorry supplies a placeholder and produces a warning.
- hard has the stated type, but its proof is still unfinished.

Definitions and Mathematical Structures

Lean	Role in a mathematical development
<code>def</code>	introduces a mathematical object or operation
<code>structure</code>	bundles related data and mathematical properties
<code>class</code>	declares a structure whose instances Lean can infer
<code>instance</code>	provides an instance of a typeclass

- `Group G` specifies group operations and laws on the type `G`.
- `[Group G]` makes this structure available through instance inference.

A Structure: Two-Dimensional Points

```
structure Point2D where
```

```
  x : Nat
```

```
  y : Nat
```

```
def origin : Point2D := { x := 0, y := 0 }
```

```
example : origin.x = 0 := by  
  rfl
```

- Point2D is a new type with two natural-number fields.
- { x := 0, y := 0 } constructs a value of this type.
- origin.x projects the first coordinate.

An Instance: A Default Point

```
instance : Inhabited Point2D where
  default := origin

#check (default : Point2D)

example : (default : Point2D).x = 0 := by
  rfl
```

- `Inhabited T` is a typeclass that supplies a default value of type `T`.
- The instance declaration chooses `origin` for `Point2D`.
- `default` uses this instance automatically, so its first coordinate is 0.

Theorem Statements and Parameters

```
theorem The_first_example {G : Type*} [Group G] {a b c d e f : G}
  (h : a * b = c * d) (h' : e = f) : a * (b * e) = c * (d * f) := by
```

$$ab = cd, \quad e = f \quad \implies \quad a(be) = c(df).$$

- G is the underlying type. $[Group\ G]$ supplies its group structure.
- $a\ b\ c\ d\ e\ f$ are elements of G .
- h and h' name the equality assumptions.
- `:= by` begins a tactic proof of the proposition after the colon.

Explicit and Implicit Arguments

Parameter	Interpretation
<code>{G : Type*}</code>	Lean usually infers the type from the arguments
<code>[Group G]</code>	typeclass inference supplies the group structure
<code>{a b c d e f : G}</code>	Lean infers these elements from the hypotheses
<code>(h : a * b = c * d)</code>	the caller supplies this equality proof

- Parentheses mark explicit arguments. Curly braces mark implicit arguments.
- Square brackets request an instance through typeclass inference.

Reading a Theorem Statement

```
theorem add_eq_of_pointwise_eq
  {α β : Type} [Add β]
  {f g : α → β}
  (h : ∀ x, f x = g x) (x : α) (c : β) :
  f x + c = g x + c := by
  rw [h x]
```

$$(\forall x, f(x) = g(x)) \implies f(x) + c = g(x) + c.$$

- $\{\alpha \ \beta : \text{Type}\}$ and $\{f \ g : \alpha \rightarrow \beta\}$ are implicit parameters.
- $[Add \ \beta]$ supplies addition on the output type.
- $(h : \dots)$, $(x : \alpha)$, and $(c : \beta)$ are explicit arguments.
- The conclusion follows by rewriting with the equality $h \ x$.

The Proof State

```
G : Type u
inst : Group G
a b c d e f : G
h : a * b = c * d
h' : e = f
⊢ a * (b * e) = c * (d * f)
```

- Above $\vdash$: the objects, structures, and hypotheses available in the proof.
- After $\vdash$: the proposition that remains to be proved.
- The Infoview updates this state after each tactic.

Rewriting with Equalities

```
example (a b : ℝ) (h : a = b) : a + 1 = b + 1 := by
  rw [h]
```

In the group proof, $h' : e = f$ changes the goal

$$a(be) = c(df) \quad \text{to} \quad a(bf) = c(df).$$

- With $h : a = b$, `rw [h]` replaces a by b in the goal.
- The real-number example closes because both sides become identical.

Associativity and Rewrite Direction

```
rw [h']  
rw ←[ mul_assoc]  
rw [h]  
exact mul_assoc c d f
```

Tactic	Remaining goal
rw [← mul_assoc]	$(ab)f = c(df)$
rw [h]	$(cd)f = c(df)$
exact mul_assoc c d f	no goals

- `mul_assoc` states $(ab)f = a(bf)$. `←` uses it in reverse.
- This exposes the subexpression ab , so `rw [h]` can replace it by cd .

Calculation Chains

```
example (a b c : ℝ) (h1 : a = b) (h2 : b = c) : a = c := by
  calc a = b := h1
      _ = c := h2
```

$$a = b, \quad b = c \quad \Rightarrow \quad a = c.$$

- `h1` proves $a = b$ and `h2` proves $b = c$.
- `_` continues the chain from the preceding right-hand side.
- `calc` combines these steps by transitivity to prove $a = c$.

Exact and Apply

```
example (p : Prop) (h : p) : p := by
  exact h
```

```
example (p q : Prop) (hpq : p → q) (hp : p) : q := by
  apply hpq
  exact hp
```

- `exact h` closes a goal when `h` has the required type.
- `apply hpq` uses `hpq : p → q` to reduce the goal `q` to `p`.
- Then `exact hp` supplies the required proof of `p`.

Introducing Hypotheses

```
example (p q : Prop) (h : p → q) : p → q := by
  intro hp
  exact h hp
```

Before intro hp	After intro hp
$h : p \rightarrow q$ $\vdash p \rightarrow q$	$h : p \rightarrow q, \quad hp : p$ $\vdash q$

- `intro hp` assumes `p`. The term `h hp` then proves `q`.

Proving a Conjunction

```
example (p q : Prop) (hp : p) (hq : q) : p ∧ q := by
  constructor
  · exact hp
  · exact hq
```

- `constructor` replaces the goal $p \wedge q$ with two goals.
- The first branch proves p using `hp`.
- The second branch proves q using `hq`.
- `·` and indentation separate the two branches.

Using a Conjunction

```
example (p q : Prop) (h : p ∧ q) : q := by
  obtain ⟨_, ⟩hq := h
  exact hq
```

- $h : p \wedge q$ contains proofs of both propositions.
- `obtain ⟨_, hq⟩ := h` names the second component `hq`.
- The underscore ignores the first component. `exact hq` proves the goal.

Definitional Equality and Simplification

```
example : 2 + 3 = 5 := by rfl
```

```
example : 5 + 0 = 5 := by simp
```

- `rfl` proves equality when both sides reduce to the same expression.
- `simp` uses simplification lemmas to reduce the goal.
- Computation reduces $2 + 3$ to 5. Simplification removes $+ 0$.

Polynomial Identities

`example (a b : ℝ) : (a + b)^2 = a^2 + 2*a*b + b^2 := by ring`

$$(a + b)^2 = a^2 + 2ab + b^2.$$

- `ring` normalizes both sides as polynomial expressions.
- The identity holds for every pair of real numbers, so no extra hypotheses appear.

Natural Number Arithmetic

`example (n : ℕ) (h : n > 3) : n ≥ 4 := by omega`

For $n \in \mathbb{N}$, $n > 3$ implies $n \geq 4$.

- omega uses the discrete arithmetic of natural numbers and integers.
- There is no natural number strictly between 3 and 4.

Linear Inequalities

`example (x y : ℝ) (h1 : x > 1) (h2 : y > 1) : x + y > 2 := by linarith`

$$x > 1, \quad y > 1 \quad \Rightarrow \quad x + y > 2.$$

- `linarith` combines the linear inequality hypotheses.
- Adding the two strict inequalities yields the desired lower bound.

The Prime Proof: Constructing a Witness

```
let N := n.factorial + 1
let p := N.minFac
use p
have hp : p.Prime := by
  apply Nat.minFac_prime
  have h1 : n.factorial ≥ 1 := Nat.factorial_pos n
  omega
refine ⟨hp, ?⟩_
```

$$N = n! + 1 \geq 2, \quad p = \text{minFac}(N).$$

- `let` names local values. `use p` supplies the existential witness.
- `have hp` proves primality using `Nat.minFac_prime` and $N \neq 1$.
- `refine ⟨hp, ?⟩_` leaves the inequality $p > n$ as the remaining goal.

The Prime Proof: Contradiction

```
by_contra hle
have h_p_dvd_fac : p | n.factorial :=
  Nat.dvd_factorial hp.pos (Nat.not_lt.mp hle)
have h_p_dvd_N : p | N := Nat.minFac_dvd N
have h_p_dvd_one : p | 1 :=
  (Nat.dvd_add_iff_right h_p_dvd_fac).mpr h_p_dvd_N
exact hp.not_dvd_one h_p_dvd_one
```

If $p \leq n$, then $p \mid n!$ and $p \mid (n! + 1)$, hence $p \mid 1$.

- `by_contra` assumes the negation of $p > n$.
- `.mpr` takes $p \mid (n! + 1)$ to $p \mid 1$, using the fact that $p \mid n!$.
- `hp.not_dvd_one` contradicts $p \mid 1$.

Outline

1. Mathematical Formalization
2. Lean Statements and Proofs
- 3. Formalizing Mathematical Optimization**
4. Theorem Search and LLM-Based Proving

A Lean library for mathematical optimization

<https://github.com/optsuite/optlib>

- **Analysis:** gradients, subdifferentials, convexity, and L -smoothness.
- **Optimality:** feasible sets, constraint qualifications, and KKT necessary conditions.
- **Convergence and complexity:**
 - Gradient and subgradient methods.
 - Proximal gradient and Nesterov acceleration.
 - Block-coordinate methods and ADMM.

Formalizing an Optimization Algorithm

$$\min_{x \in E} f(x), \quad x_{k+1} = x_k - \alpha_k \nabla f(x_k).$$

- **Problem:** the space E , objective f , and its gradient.
- **Regularity:** differentiability, convexity, and smoothness for the analysis.
- **Iteration:** the initial point x_0 and recurrence defining the sequence $\{x_k\}$.
- **Step sizes:** positivity and upper bounds needed for the descent inequality.

Lean interface: a `class` records the problem and iteration. An `instance` connects a concrete algorithm to it.

Gradient Descent: Algorithm Definition

Fixed-step gradient descent

$$x_{k+1} = x_k - t \nabla f(x_k).$$

```
variable {E : Type*} [NormedAddCommGroup E] [InnerProductSpace ℝ E] [CompleteSpace E]
class unconstrained_problem (f : E → ℝ)
variable {f : E → ℝ} {pro : unconstrained_problem f}
class Gradient_Descent (pro : unconstrained_problem f) (x0 : E) :=
  (x : ℕ → E) (t : ℝ) (l : NNReal)
  (update : ∀ k : ℕ, x (k + 1) = x k - t • (gradient f) (x k))
  (hl : l > 0) (step1 : t > 0) (initial : x 0 = x0)
```

- $x : \mathbb{N} \rightarrow E$ records the iterates. $t : \mathbb{R}$ records the fixed step size.
- `update` and `initial` state the recurrence and initial condition.
- `hl` and `step1` assert positivity of the smoothness bound and step size.

Gradient Descent: Convergence Rate

For convex L -smooth f and a minimizer x_* , with $L > 0$ and $0 < t < 1/L$,

$$f(x_k) - f(x_*) \leq \frac{\|x_0 - x_*\|^2}{2tk}, \quad k \geq 1.$$

```
lemma gradient_method_convergence {alg : Gradient_Descent pro x₀}
  (diff : Differentiable ℝ f) (smooth : LipschitzWith alg.l (gradient f))
  (hfun : ConvexOn ℝ Set.univ f) (step2 : alg.t ≤ 1 / alg.l) (xm : E) :
  ∀ k : ℕ, f (alg.x (k + 1)) - f xm ≤ 1 / (2 * (k + 1) * alg.t) * ‖x₀ - xm‖ ^ 2 := by
```

- `diff`, `smooth`, and `hfun` supply the analytic assumptions. `step2` gives the required upper bound on the step size.
- The formal index is $k + 1$. The comparison point `xm` can be chosen as x_* .

Formalizing Constrained Optimization Problems

$$\min_{x \in \mathbb{R}^n} f(x) \quad \text{subject to} \quad \begin{cases} c_i(x) = 0, & i \in \mathcal{E}, \\ c_i(x) \geq 0, & i \in \mathcal{I}. \end{cases}$$

```
structure Constrained_OptimizationProblem (E : Type _) (τ σ : Finset N) :=  
  (domain : Set E)  
  (equality_constraints : (i : N) → E → R)  
  (inequality_constraints : (j : N) → E → R)  
  (eq_ine_not_intersect : τ ∩ σ = ∅)  
  (objective : E → R)
```

- **Problem structure:** domain, objective, and indexed constraints. It also defines feasible points and local minimizers.
- **Local geometry:** active constraints, tangent cones, and linearized feasible directions for the KKT proof.

KKT Necessary Conditions

$$\mathcal{L}(x, \lambda, \mu) = f(x) + \sum_{i \in \mathcal{E}} \lambda_i c_i(x) - \sum_{i \in \mathcal{I}} \mu_i c_i(x).$$

For a C^1 problem, LICQ gives KKT multipliers at a local minimizer x_* :

$$\nabla_x \mathcal{L}(x_*, \lambda, \mu) = 0 \quad (\text{stationarity}),$$

$$c_i(x_*) = 0 \ (i \in \mathcal{E}), \quad c_i(x_*) \geq 0 \ (i \in \mathcal{I}) \quad (\text{feasibility}),$$

$$\mu_i \geq 0, \quad \mu_i c_i(x_*) = 0 \ (i \in \mathcal{I}) \quad (\text{dual feasibility and complementarity}).$$

Equality multipliers are unrestricted. LICQ means linear independence of the equality and active inequality gradients.

Formalizing the KKT Necessary Conditions

```
theorem first_order_necessary (p1 : Constrained_OptimizationProblem (EuclideanSpace R (Fin n))  $\tau$   $\sigma$ )
  (loc : EuclideanSpace R (Fin n)) (h1 : p1.Local_Minima loc)
  (hf : Differentiable R p1.objective)
  (conte :  $\forall i \in \tau$ , ContDiffAt R (1 : N) (equality_constraints p1 i) loc)
  (conti :  $\forall i \in \sigma$ , ContDiffAt R (1 : N) (inequality_constraints p1 i) loc)
  (hLICQ : p1.LICQ loc) (hdomain : p1.domain = univ) :
  p1.FeasPoint loc  $\wedge$  ( $\exists$  (lambda1 :  $\tau \rightarrow R$ ) (lambda2 :  $\sigma \rightarrow R$ ),
    (gradient (fun m  $\mapsto$  (p1.Lagrange_function m lambda1 lambda2)) loc = 0)  $\wedge$  ( $\forall j : \sigma$ , lambda2 j  $\geq$  0)  $\wedge$ 
    ( $\forall j : \sigma$ , (lambda2 j) * (p1.inequality_constraints j loc) = 0)) := by
```

- $h1$ states local minimality. $hLICQ$ states the constraint qualification.
- The conclusion gives feasibility and multipliers satisfying stationarity, nonnegativity, and complementarity.
- The proof uses a formalized Farkas lemma to obtain the multipliers.

Outline

1. Mathematical Formalization
2. Lean Statements and Proofs
3. Formalizing Mathematical Optimization
- 4. Theorem Search and LLM-Based Proving**

Theorem Search with `exact?` and `apply?`

```
example (a b : Nat) (h : a ≤ b) :  
  a + 1 ≤ b + 1 := by  
  exact?
```

```
example (a b : Nat) (h : a ≤ b) :  
  a + 1 ≤ b + 1 := by  
  apply?
```

- `exact?` searches for a proof that closes the current goal.
- `apply?` searches for an applicable theorem and tries to discharge its premises. It may leave subgoals.
- For this example, both suggest `exact Nat.add_le_add_right h 1`.

Guessing Theorem Names

A theorem name often reflects its objects and conclusion.

Mathematical fact	Candidate name
A positive $p \leq n$ divides $n!$	<code>Nat.dvd_factorial</code>
The least factor of $N \neq 1$ is prime	<code>Nat.minFac_prime</code>
The least factor divides N	<code>Nat.minFac_dvd</code>
A prime does not divide 1	<code>Nat.Prime.not_dvd_one</code>

- Guess a namespace, such as `Nat`, and keywords, such as `dvd` and `factorial`.
- Then read the declaration to check its hypotheses and conclusion.

Searching for a Divisibility Lemma

Formal statement to prove

$\forall p, n : \mathbb{N},$
 $p.\text{Prime} \rightarrow p \leq n \rightarrow p \mid n.\text{factorial}$

Natural-language query

A prime number at most n divides $n!$.

- An **embedding model** maps the query and theorem descriptions to vectors.
- LeanSearch (leansearch.net) retrieves and reranks candidate theorems.
- Check that the retrieved theorem has the right conclusion and that its premises follow from the stated assumptions.

The Statement of a Library Theorem

```
Nat.dvd_factorial {m n : ℕ} :  
  0 < m → m ≤ n → m ∣ n.factorial
```

$$0 < m, \quad m \leq n \quad \Rightarrow \quad m \mid n!.$$

- m and n are implicit parameters of type $\mathbb{N}$.
- $0 < m$ and $m \leq n$ are the premises. $m \mid n.factorial$ is the conclusion.
- For our goal, Lean infers $m = p$. Primality supplies the positivity premise.

Applying a Retrieved Theorem

```
have h_p_dvd_fac : p | n.factorial :=  
  Nat.dvd_factorial hp.pos (Nat.not_lt.mp hle)
```

- `hp.pos` supplies the first premise, $0 < p$.
- `Nat.not_lt.mp hle` supplies the second premise, $p \leq n$.
- Applying `Nat.dvd_factorial` to these proofs produces a proof of $p \mid n!$.

LLM-Assisted Theorem Proving

LLMs construct candidates. Lean checks them.

- The model proposes a formal statement, a complete proof, or the next tactic.
- Lean returns acceptance, the resulting proof state, or an error message.
- Search explores alternative candidates and uses this feedback to revise them.
- Successful proofs and failed attempts can provide training signals.

A finished proof must pass Lean's kernel check for the specified statement.

Three Proof Paradigms

	Whole proof	Stepwise	Agentic
Model output	Complete proof	Next tactic	Auxiliary lemmas and formal proof code
Lean feedback	Per proof candidate	Per tactic	Per proof attempt
Search / control	Proof sampling	Proof-state search	Lemma generation and revision
Main challenge	Long proof coherence	Branch selection	Finding useful intermediate lemmas

- Agentic systems can use whole-proof generation or tactic-level search as components for proving their intermediate lemmas.
- All approaches must produce a complete proof checked by Lean for the target statement under its declared assumptions.