Recap:

- Lean checks that a given formal proof is a valid proof of the formal statement it is attached to
- LLMs are very good at writing formal proofs given a formal statement. They are good (but not perfect) at writing formal statements of natural language statements.
- Lean is its own programming language and can execute arbitrary code on your computer.

How Lean checks a proof

1. What does Lean check?
2. Why does that establish a proof?
3. How does the kernel check it?

Our running example

If f : A → B and g : B → C are injective then g ∘ f is injective.

`g (f x) = g (f y)`	assume
`f x = f y`	use injectivity of g
`x = y`	use injectivity of f

Terms and types

A term is an expression, such as 3, x, or f x. We write t : A to mean that the term t has type A

`3 : Nat`	3 is a term whose type is Nat.
`x : A`	x is an object of type A.
`f : A → B`	f is a function.
`f x : B`	Apply f to x.

```
(fun x : A => x) : A → A
```

`fun` builds a function type

A proof of a theorem is a term whose type is the same as the theorem statement

`T : Prop` T is a proposition.

`t : T` t is a proof of T.

For example (if h is a proof that x = y):

`x = y : Prop` `h : x = y`

Lean checks that the proposed term t has type T

Implication as a function type

```
P : Prop    Q : Prop
```

```
u : P → Q    p : P
```

Applying u to p gives `u p : Q`.

This is modus ponens: from P and P → Q, conclude Q.

To prove P → Q, assume `p : P` and build a proof of Q.

Example: To prove P → P, return the original proof

```
(fun p : P => p) : P → P
```

Universal quantification as a function type

```
P : A → Prop    a : A    P a : Prop
```

A predicate P assigns a proposition to each element of A (note this proposition depends on a).

These are two notations for the same type:

```
∀ x : A, P x    (x : A) → P x
```

Proofs:

```
p : ∀ x : A, P x    p a : P a
```

A proof p of `∀ x : A, P x` takes an element `a : A` and returns a proof `p a of P a`

A proof term records the steps of a proof

Building the proof term	Logical step
Use `p : P` from the context.	Use the hypothesis P.
Apply `u : P → Q` to p.	Deduce Q from P and P → Q.
Build a function from proofs of P to proofs of Q.	Prove an implication.
Build a function from arbitrary x to proofs of P x.	Prove a universal statement.

Checking these steps checks the correctness of the logical deduction

The elaborator and the kernel

Lean source	Statements, notation, proof instructions
↓	
Elaborator	Fills in omitted information and builds core expressions
↓	
`statement T + proof term t`	Explicit terms for checking
↓	
Kernel	Independently checks `t : T`

The kernel applies a fixed set of rules to the supplied terms.

Checking an expression recursively

The context records the available names and their types.

Expression	What the kernel does
A variable such as `x`	If the context contains `x : A`, report that x has type A.
Function construction `fun x : A => body`	Check A is a type. Add `x : A` to the context, then check `body`.
Function application `f a`	Find f's type. If it is `A → B`, check `a : A`, then conclude `f a : B`.

For function construction, if `body : R x`, the result has type `(x : A) → R x`.

Comparing types for definitional equality

The kernel can also unfold definitions and reduce expressions, though the elaborator does most of this.

Expression	After computation
`(fun z : A => z) a`	`a`
`(g ∘ f) x`	`g (f x)`

This lets differently written types match.

Finally, compare the inferred type of t with T.

Our running example

If $f : A \to B$ and $g : B \to C$ are injective then $g \circ f$ is injective.

`g (f x) = g (f y)`	assume
`f x = f y`	use injectivity of g
`x = y`	use injectivity of f

The hypotheses and the goal

```
A : Type    B : Type    C : Type
f : A → B    g : B → C
```

```
hf : Function.Injective f
hg : Function.Injective g
```

Unfolding their types, with all point inputs shown:

```
hf : ∀ x y : A, f x = f y → x = y
hg : ∀ u v : B, g u = g v → u = v
```

```
T : Prop := Function.Injective (g ∘ f)
```

We need a term `t : T`. This will be a function that takes in `x : A`, `y : A`, and a proof that `g (f x) = g (f y)`, and outputs a proof that `x = y`.

The complete proof term

The elaborator supplies T and a complete proof term t.

```
t : T :=
  fun (x : A) (y : A)
      (h0 : (g ∘ f) x = (g ∘ f) y) =>
    let h1 : f x = f y := @hg (f x) (f y) h0
    let h2 : x = y := @hf x y h1
    h2
```

We now follow the kernel’s checks of this proof term.

Unfolding the statement

```
T := Function.Injective (g ∘ f)
```

Unfold injectivity.

```
∀ x y : A, (g ∘ f) x = (g ∘ f) y → x = y
```

Unfolding the statement

```
T := Function.Injective (g ∘ f)
```

Unfold injectivity.

```
∀ x y : A, (g ∘ f) x = (g ∘ f) y → x = y
```

Unfold composition.

```
∀ x y : A, g (f x) = g (f y) → x = y
```

Unfolding the statement

```
T := Function.Injective (g ∘ f)
```

Unfold injectivity.

```
∀ x y : A, (g ∘ f) x = (g ∘ f) y → x = y
```

Unfold composition.

```
∀ x y : A, g (f x) = g (f y) → x = y
```

Read the same type as a function type.

```
(x : A) → (y : A) →
  (h0 : g (f x) = g (f y)) → x = y
```

Each displayed form of the theorem statement T has type Prop

Inputs/context, and a wrong step

Before checking the function body, the kernel records its inputs:

```
x : A    y : A
h0 : (g ∘ f) x = (g ∘ f) y
```

After unfolding: `h0 : g (f x) = g (f y)`.

The body must return a proof of `x = y`.

Inputs/context, and a wrong step

Before checking the function body, the kernel records its inputs:

```
x : A    y : A
h0 : (g ∘ f) x = (g ∘ f) y
```

After unfolding: `h0 : g (f x) = g (f y)`.

The body must return a proof of `x = y`.

Using `@hf x y h0` as the body would fail:

```
@hf x y : f x = f y → x = y
```

Required input type: `f x = f y`

Type of h0: `g (f x) = g (f y)`

Function application rule: these types do not match.

Checking h1 using injectivity of g

```
hg : ∀ u v : B, g u = g v → u = v
```
(g is injective)

```
h0 : g (f x) = g (f y)
```

```
f x : B   f y : B   g (f x) : C   g (f y) : C
```

Checking h1 using injectivity of g

```
hg : ∀ u v : B, g u = g v → u = v
```
(g is injective)

```
h0 : g (f x) = g (f y)
```

```
f x : B   f y : B   g (f x) : C   g (f y) : C
```

Function application rule: substitute each point into the type.

```
@hg (f x) (f y) :
  g (f x) = g (f y) → f x = f y
```

Checking h1 using injectivity of g

```
hg : ∀ u v : B, g u = g v → u = v
```
(g is injective)

```
h0 : g (f x) = g (f y)
```

```
f x : B   f y : B   g (f x) : C   g (f y) : C
```

Function application rule: substitute each point into the type.

```
@hg (f x) (f y) :
  g (f x) = g (f y) → f x = f y
```

h0 has the required input type, after unfolding.

```
h1 := @hg (f x) (f y) h0
h1 : f x = f y
```

Checking h2 using injectivity of f

`hf : ∀ u v : A, f u = f v → u = v` (f is injective)
`x : A  y : A  h1 : f x = f y`

Function application rule: supply x and y

`@hf x y : f x = f y → x = y`

Checking h2 using injectivity of f

`hf : ∀ u v : A, f u = f v → u = v` (f is injective)
`x : A  y : A  h1 : f x = f y`

Function application rule: supply x and y

`@hf x y : f x = f y → x = y`

h1 has exactly the required input type.

```
h2 := @hf x y h1
h2 : x = y
```

The body returns h2, a proof of the required equality.

Checking the complete proof

```
t : T := fun x y h0 =>
  let h1 : f x = f y := @hg (f x) (f y) h0
  let h2 : x = y := @hf x y h1
  h2
```

Checking the complete proof

```
t : T := fun x y h0 =>
  let h1 : f x = f y := @hg (f x) (f y) h0
  let h2 : x = y := @hf x y h1
  h2
```

Function rule: bind the inputs to obtain

```
t : (x : A) → (y : A) →
  (h0 : (g ∘ f) x = (g ∘ f) y) → x = y
```

Type comparison: this matches
`T := Function.Injective (g ∘ f)` after unfolding.

The same proof in Lean

`LeanDemo/LeanDemo/KernelDemo.lean`

1. Follow h0, h1, and h2 in `inj_comp`.
2. Inspect the elaborated statement and proof.
3. Replay the kernel’s type checks and unfolding.