

AAA616: Program Analysis

Lecture 14 — Introduction to Program Verification in Lean

Hakjoo Oh
2026 Fall

Lean

- Lean is a functional programming language *and* an interactive theorem prover.
- Programs and proofs are written in the same language and checked by the same type checker.
- Pure: evaluating an expression has no side effects.
- Statically typed, with a very expressive type system (*dependent types*).



Example (1)

Here is a list-reversal function that uses an accumulator:

```
def revAcc (xs acc : List  $\alpha$ ) : List  $\alpha$  :=  
  match xs with  
  | [] => acc  
  | x :: xs' => revAcc xs' (x :: acc)  
  
def reverseFast (xs : List  $\alpha$ ) : List  $\alpha$  := revAcc xs []  
  
#eval reverseFast [1, 2, 3]      -- [3, 2, 1]
```

- The test passes. Is reverseFast correct for *all* lists?
- Correct with respect to what? We need a *specification*: a description of the intended behavior that is obviously right.

Example (1): Specification and Proof

The specification is the simplest reverse (slow, but obviously right), and the claim is that the two agree on every input:

```
def reverse : List  $\alpha$   $\rightarrow$  List  $\alpha$ 
| [] => []
| x :: xs => reverse xs ++ [x]

theorem reverseFast_correct :
   $\forall$  xs : List  $\alpha$ , reverseFast xs = reverse xs := by
  intro xs
  simp [reverseFast, revAcc_eq]
```

- revAcc_eq is a four-line lemma proved by induction (shown at the end of the lecture).
- Lean accepts the proof: reverseFast agrees with the specification on every list, of every length.
- A human needs to read only the specification. The code and the proof are checked by Lean.

Example (1): A Wrong reverse

Suppose the code appended at the wrong end:

```
def revAcc (xs acc : List  $\alpha$ ) : List  $\alpha$  :=  
  match xs with  
  | [] => acc  
  | x :: xs' => revAcc xs' (acc ++ [x])      -- bug  
  
#eval reverseFast [1, 2, 3]      -- [1, 2, 3]
```

The same proof attempt fails, and Lean shows what it could not prove:

```
unsolved goals  
case cons  
...  
⊢ acc ++ [x] = x :: acc
```

Example (2): Integer Overflow

The midpoint computation of a binary search:

```
def mid (lo hi : UInt32) : UInt32 := (lo + hi) / 2
#eval mid 10 20      -- 15
#eval mid 0 100      -- 50
```

Binary search relies on one property: if $lo \leq hi$, the midpoint lies between them. Lean checks all 2^{64} input pairs:

```
import Std.Tactic.BVDecide      -- at the top of the file
theorem mid_ok :
  ∀ lo hi : UInt32, lo ≤ hi → lo ≤ mid lo hi ∧ mid lo hi ≤ hi := by
  intro lo hi h
  unfold mid
  bv_decide
```

The prover found a counterexample, consider the following assignment:

```
hi = 4294967295
```

```
lo = 4294967295
```

Example (2): Integer Overflow (Continued)

- $lo + hi$ overflows 32 bits: mid 4294967295 4294967295 is 2147483647, far below lo .
- This code was in Java's standard library for nine years. Tests never triggered it: it needs arrays of more than a billion elements.¹

The fix, and the same theorem now accepted:

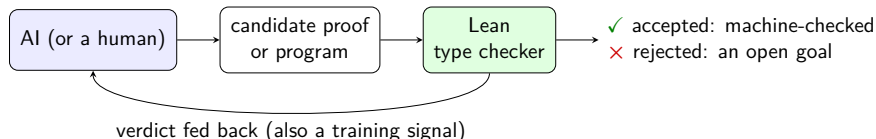
```
def mid (lo hi : UInt32) : UInt32 := lo + (hi - lo) / 2

theorem mid_ok :
  ∀ lo hi : UInt32, lo ≤ hi → lo ≤ mid lo hi ∧ mid lo hi ≤ hi := by
  intro lo hi h
  unfold mid
  bv_decide -- accepted
```

¹J. Bloch, "Nearly All Binary Searches and Mergesorts are Broken," Google Research Blog, 2006. <https://research.google/blog/extra-extra-read-all-about-it-nearly-all-binary-searches-and-mergesorts-are-broken/>

Why Lean, Why Now?

AI now produces code and mathematical proofs at scale. The bottleneck has moved from *producing* to *checking*: is the proof correct? does the program do what it claims?



- Nobody can referee thousands of lines of AI-generated proof by hand, whether it proves a theorem or the correctness of a program. A Lean proof is checked by a small trusted kernel: if Lean accepts it, it is correct (for the stated claim).
- So Lean has become a trust layer between AI and mathematics, and increasingly software. The verdict is also a training signal for the AI.

Lean in Mathematics and AI

- 2023** Tao et al. formalize the Polynomial Freiman–Ruzsa theorem in Lean, three weeks after the paper.
- 2024** DeepMind’s AlphaProof: silver-medal level at the IMO, with proofs written in Lean (Nature, 2025).
- 2025** Tao’s Equational Theories Project: 22 million implications between algebraic laws, all in Lean, by humans and AI.
- 2025** AI reaches gold-medal level at the IMO; Harmonic’s Aristotle delivers its solutions as Lean-verified proofs.
- 2026** AI resolves open Erdős problems (e.g., #728, with a Lean proof); Axiom’s prover reports 42/42 at IMO 2026 with public Lean proofs.
- Mathlib** 290,000 theorems, 138,000 definitions, 770+ contributors; Fermat’s Last Theorem is being formalized.

Getting Started

- Install Lean with elan: <https://lean-lang.org/install/>
- Use VS Code with the Lean 4 extension.
- Create a file `hello.lean`. Lean checks it as you type and shows messages in the *Infoview* panel.
- `#eval` evaluates an expression and shows its value:

```
#eval 1 + 2      -- 3
```

- In the slides, Lean's answers are written as comments (`-- 3`), and longer messages in gray boxes.

Roadmap

① **Functional programming in Lean**

expressions, types, functions, structures, inductive types, pattern matching, recursion, polymorphism, lists

② **Dependent types: from programs to proofs**

types that depend on values, propositions as types, proofs as programs

③ **Theorem proving basics**

logical connectives, tactics, induction, verifying a program

Part I

Functional Programming in Lean

Programs Are Expressions

A statement does something. An expression evaluates to a value.

- Imperative languages are statement-oriented: a program is a sequence of statements that modify memory.
- Lean is expression-oriented: a program is an expression, and running it means evaluating it, as in arithmetic:
$$2 \times (3 + 1) \implies 2 \times 4 \implies 8.$$
- In Lean, a variable never changes its value, and evaluation has no side effects.

Evaluating Expressions

`#eval` asks Lean for the value of an expression:

```
#eval 1 + 2      -- 3
#eval 1 + 2 * 5   -- 11
#eval 7 / 2       -- 3
#eval 7 % 2       -- 1
```

`#eval` is a command for asking Lean. It is not part of a program.

Function Application

A function is applied by writing it next to its arguments, separated by spaces:

```
#eval String.append "Hello, " "Lean!"      -- "Hello, Lean!"
```

Parentheses group a nested application:

```
#eval String.append "great " (String.append "oak " "tree")  
-- "great oak tree"
```

Note that $f (g x)$ in Lean is $f(g(x))$ in C. $f g x$ means f applied to the two arguments g and x .

Conditionals

`if` is an expression, not a statement. The `else` branch is mandatory.

```
#eval if 1 > 2 then "yes" else "no"      -- "no"
```

Evaluation (`==>` is our notation, not Lean code):

```
String.append "it is " (if 1 > 2 then "yes" else "no")
==> String.append "it is " (if false then "yes" else "no")
==> String.append "it is " "no"
==> "it is no"
```

cf) Python: `"yes" if 1 > 2 else "no"`

Static Typing

Every expression has a type, and Lean checks types before evaluating.

`#check` shows the type without evaluating:

```
#check 1 + 2      -- 1 + 2 : Nat
#check "Hello"    -- "Hello" : String
```

An ill-typed expression is rejected:

```
#check String.append ["hello", " "] "world"
```

```
Application type mismatch: The argument
  ["hello", " "]
has type
  List String
but is expected to have type
  String
```

Basic Types

```
#check 42           -- 42 : Nat
#check (-3 : Int)    -- -3 : Int
#check 1.5           -- 1.5 : Float
#check 'a'           -- 'a' : Char
#check "Lean"        -- "Lean" : String
#check true          -- Bool.true : Bool
#check ()            -- () : Unit
```

- Nat: natural numbers **0, 1, 2, ...** of arbitrary size, the default type of number literals. (e : T) is a type annotation.
- Note that subtraction on Nat stops at zero:

```
#eval (1 - 2 : Nat)    -- 0
#eval (1 - 2 : Int)    -- -1
```

Definitions

`def` gives a name to a value:

```
def hello := "Hello"
def lean : String := "Lean"

#eval String.append hello (String.append " " lean)
-- "Hello Lean"
```

- `:=` defines a name. (`=` is equality.)
- Lean can infer the type of `hello`, but it is good practice to write types.

Functions

Arguments are listed before the colon, the result type after it:

```
def add1 (n : Nat) : Nat := n + 1

def maximum (n : Nat) (k : Nat) : Nat :=
  if n < k then k else n

#eval add1 7          -- 8
#eval maximum 3 4     -- 4
```

Application substitutes the arguments into the body:

```
    maximum (5 + 8) (2 * 7)
==> maximum 13 14
==> if 13 < 14 then 14 else 13
==> 14
```

$(n\ k : \text{Nat})$ abbreviates $(n : \text{Nat})\ (k : \text{Nat})$.

Function Types and Currying

```
#check (add1)           -- add1 : Nat → Nat
#check (maximum)        -- maximum : Nat → Nat → Nat
#check maximum 3        -- maximum 3 : Nat → Nat
```

- Every function takes exactly one argument. `maximum` takes `n` and returns a function that takes `k` (*currying*).
- `Nat → Nat → Nat` means `Nat → (Nat → Nat)`, and `maximum 3 4` means `(maximum 3) 4`.
- `maximum 3` is a function (*partial application*). A function cannot be displayed:

```
#eval maximum 3
```

```
Could not synthesize a `ToExpr`, `Repr`, or `ToString` instance
for type
  Nat → Nat
```

Anonymous Functions

`fun x => e` is a function without a name:

```
#check fun x => x + 1          -- fun x => x + 1 : Nat → Nat
#check fun (x : Int) => x + 1  -- fun x => x + 1 : Int → Int

#eval (fun x => x + x) 5        -- 10
#eval (· * 2) 5                -- 10
```

- `(· * 2)` is shorthand for `fun x => x * 2`. Each `·` becomes an argument.
- Note that `·` (typed as `\.` in VS Code) is not the period.

Higher-Order Functions

Functions are values: they can be passed as arguments and returned.

```
def twice (f : Nat → Nat) (x : Nat) : Nat := f (f x)
```

```
#eval twice add1 5          -- 7
```

```
#eval twice (· * 2) 5       -- 20
```

```
def addN (n : Nat) : Nat → Nat := fun x => x + n
```

```
#eval addN 3 4              -- 7
```

Quiz) What Is the Type of the Function?

```
def f (g : Nat → Nat) (x : Nat) := g (g x) + x
```

Quiz) What Is the Type of the Function?

```
def f (g : Nat → Nat) (x : Nat) := g (g x) + x
```

```
#check (f)      -- f : (Nat → Nat) → Nat → Nat
```

Local Definitions

`let` names an intermediate value:

```
def hyp (a b : Float) : Float :=  
  let s := a * a + b * b  
  Float.sqrt s  
  
#eval hyp 3 4      -- 5.000000
```

The body follows on the next line, or after a semicolon:

```
let s := ...; Float.sqrt s.
```

Structures

A structure groups named fields into a new type (cf) struct in C):

```
structure Point where
  x : Float
  y : Float
deriving Repr

def origin : Point := { x := 0.0, y := 0.0 }

#eval origin          -- { x := 0.000000, y := 0.000000 }
#eval origin.x        -- 0.000000
```

`deriving Repr` lets `#eval` display values of the new type.

Functions on Structures

```
def addPoints (p1 p2 : Point) : Point :=  
  { x := p1.x + p2.x, y := p1.y + p2.y }  
  
#eval addPoints { x := 1.5, y := 32 } { x := -8, y := 0.2 }  
-- { x := -6.500000, y := 32.200000 }
```

```
def distance (p1 p2 : Point) : Float :=  
  Float.sqrt (((p2.x - p1.x) ^ 2.0) + ((p2.y - p1.y) ^ 2.0))  
  
#eval distance { x := 1.0, y := 2.0 } { x := 5.0, y := -1.0 }  
-- 5.000000
```

Updating Structures

`{ p with x := 0 }` is a copy of `p` with `x` replaced:

```
def zeroX (p : Point) : Point :=  
  { p with x := 0 }  
  
def fourAndThree : Point := { x := 4.3, y := 3.4 }  
  
#eval zeroX fourAndThree      -- { x := 0.000000, y := 3.400000 }  
#eval fourAndThree            -- { x := 4.300000, y := 3.400000 }
```

Note that the original value is unchanged. Nothing is mutated in Lean.

Constructors and Dot Notation

A structure `S` comes with a constructor `S.mk` and an accessor `S.f` for each field `f`:

```
#check (Point.mk)      -- Point.mk : Float → Float → Point
#check (Point.x)       -- Point.x  : Point → Float

#eval Point.x origin   -- 0.000000   (the same as origin.x)
```

In general, `e.f a` calls `T.f e a` when `e` has type `T`:

```
#eval "one string".append " and another"
-- "one string and another"
```

Inductive Types

An inductive type is defined by its *constructors*. In the standard library, `Bool` and `Nat` are defined as follows:

```
inductive Bool where
  | false : Bool
  | true  : Bool

inductive Nat where
  | zero : Nat
  | succ (n : Nat) : Nat
```

- Every `Nat` is built from `zero` and `succ`: 2 is `Nat.succ (Nat.succ Nat.zero)`.
- A constructor just holds its arguments. There is no initialization code.

Pattern Matching

`match` checks which constructor built a value, and names its arguments:

```
def isZero (n : Nat) : Bool :=  
  match n with  
  | Nat.zero => true  
  | Nat.succ k => false
```

```
def pred (n : Nat) : Nat :=  
  match n with  
  | Nat.zero => Nat.zero  
  | Nat.succ k => k
```

```
#eval pred 5      -- 4
```

```
#eval pred 0      -- 0
```

```
pred 5  
==> match Nat.succ 4 with | Nat.zero => Nat.zero | Nat.succ k => k  
==> 4
```

Defining Your Own Types

```
inductive Shape where
  | circle (radius : Float)
  | rect (width height : Float)
deriving Repr

def area (s : Shape) : Float :=
  match s with
  | Shape.circle r => 3.14 * r * r
  | Shape.rect w h => w * h

#eval Shape.circle 1.0           -- Shape.circle 1.000000
#eval area (Shape.rect 2 3)      -- 6.000000
```

Knowing how to construct a value tells you how to take it apart.

Recursive Functions

A recursive function follows the shape of the data: a base case for zero, a recursive case for succ.

```
def even (n : Nat) : Bool :=  
  match n with  
  | Nat.zero => true  
  | Nat.succ k => not (even k)  
  
def plus (n : Nat) (k : Nat) : Nat :=  
  match k with  
  | Nat.zero => n  
  | Nat.succ k' => Nat.succ (plus n k')  
  
#eval even 7      -- false  
#eval plus 3 2    -- 5
```

We will prove properties of plus later.

Natural Number Patterns

Patterns can use number literals and $n + 1$:

```
def even : Nat → Bool           -- replaces the earlier even
| 0 => true
| n + 1 => not (even n)

def halve : Nat → Nat
| 0 => 0
| 1 => 0
| n + 2 => halve n + 1

#eval halve 7           -- 3
```

- `def f : A → B | p => ...` matches directly on the argument.
- `n + 2` is `Nat.succ (Nat.succ n)`. The literal must be on the right.

Termination

Lean requires every recursive function to terminate:

```
def evenLoops (n : Nat) : Bool :=  
  match n with  
  | Nat.zero => true  
  | Nat.succ k => not (evenLoops n)
```

```
fail to show termination for  
  evenLoops  
with errors  
failed to infer structural recursion:  
Not considering parameter n of evenLoops:  
  it is unchanged in the recursive calls
```

- Recursion on a constructor argument (*structural recursion*) is accepted automatically. Other recursion needs a termination proof.
- Why so strict? Part II gives the reason.

Polymorphism

Types are values of type `Type`, so a type can be an argument:

```
structure PPoint ( $\alpha$  : Type) where
  x :  $\alpha$ 
  y :  $\alpha$ 
deriving Repr

def natOrigin : PPoint Nat := { x := Nat.zero, y := Nat.zero }

def replaceX ( $\alpha$  : Type) (point : PPoint  $\alpha$ ) (newX :  $\alpha$ ) : PPoint  $\alpha$  :=
  { point with x := newX }

#eval replaceX Nat natOrigin 5      -- { x := 5, y := 0 }
```

`PPoint Nat`, `PPoint String`, ... are all types.

Implicit Arguments

With curly braces, Lean fills in the argument from the other arguments:

```
def replaceX { $\alpha$  : Type} (point : PPoint  $\alpha$ ) (newX :  $\alpha$ ) : PPoint  $\alpha$  :=  
  { point with x := newX }  
  
#eval replaceX natOrigin 5           -- { x := 5, y := 0 }
```

In fact, α need not be declared at all. Lean adds $\{\alpha : \text{Type}\}$ automatically:

```
def replaceX (point : PPoint  $\alpha$ ) (newX :  $\alpha$ ) : PPoint  $\alpha$  :=  
  { point with x := newX }
```

(Each version replaces the previous one.)

Lists

```
def primesUnder10 : List Nat := [2, 3, 5, 7]
```

List is an inductive type:

```
inductive List ( $\alpha$  : Type) where
  | nil : List  $\alpha$ 
  | cons :  $\alpha \rightarrow$  List  $\alpha \rightarrow$  List  $\alpha$ 
```

[2, 3] is List.cons 2 (List.cons 3 List.nil), also written
2 :: 3 :: []. Functions on lists follow the two constructors:

```
def length (xs : List  $\alpha$ ) : Nat :=
  match xs with
  | [] => 0
  | _ :: ys => Nat.succ (length ys)

#eval length primesUnder10      -- 4
```

List Functions

The standard library provides many list functions:

```
#eval primesUnder10.length           -- 4
#eval [1, 2, 3].map (· * 2)           -- [2, 4, 6]
#eval [1, 2, 3, 4].filter (· % 2 == 0) -- [2, 4]
#eval [1, 2, 3].foldl (· + ·) 0       -- 6
#eval [1, 2] ++ [3]                  -- [1, 2, 3]
#eval [1, 2, 3].reverse              -- [3, 2, 1]
```

`xs.map f` is `List.map f xs` (dot notation).

Option

Option α is a value of type α that may be missing:

```
inductive Option ( $\alpha$  : Type) : Type where
  | none : Option  $\alpha$ 
  | some (val :  $\alpha$ ) : Option  $\alpha$ 
```

```
def first? (xs : List  $\alpha$ ) : Option  $\alpha$  :=
  match xs with
  | [] => none
  | y :: _ => some y

#eval first? primesUnder10      -- some 2
#eval first? ([] : List Nat)    -- none
```

There is no null. A missing value must be handled by pattern matching.

Pairs and Sums

$\alpha \times \beta$ is the type of pairs, $\alpha \oplus \beta$ is “either α or β ”:

```
def fives : String × Int := ("five", 5)
```

```
#eval fives.fst      -- "five"
```

```
#eval fives.snd      -- 5
```

```
def animals : List (String ⊕ String) :=  
  [Sum.inl "Spot", Sum.inr "Tiger", Sum.inl "Rex"]
```

```
def howManyDogs : List (String ⊕ String) → Nat  
| [] => 0  
| Sum.inl _ :: rest => howManyDogs rest + 1  
| Sum.inr _ :: rest => howManyDogs rest
```

```
#eval howManyDogs animals      -- 2
```

Quiz) What Is the Result?

```
#eval [1, 2, 3].map (fun x => (x, x * x))
```

Quiz) What Is the Result?

```
#eval [1, 2, 3].map (fun x => (x, x * x))
```

```
-- [(1, 1), (2, 4), (3, 9)]
```

Summary of Part I

- Programs are expressions, and every expression has a type.
- Functions are values; every function takes one argument.
- Data: **structure** (a record), **inductive** (constructors); pattern matching takes it apart.
- Recursion follows the data and must terminate.
- Types are values: polymorphism is just type arguments.

Next: types that depend on values.

Part II

Dependent Types: From Programs to Proofs

Dependent Function Types

The type of `replaceX` (the explicit version) names its first argument, and later types refer to it:

```
#check (replaceX)
-- replaceX : ( $\alpha$  : Type)  $\rightarrow$  PPoint  $\alpha \rightarrow \alpha \rightarrow$  PPoint  $\alpha$ 
```

- $(x : A) \rightarrow B$ is a *dependent function type*: the result type B may depend on the argument x .
- If B does not mention x , it is the ordinary $A \rightarrow B$.
- The type of `replaceX Nat` is obtained by substituting `Nat` for α :
`PPoint Nat  $\rightarrow$  Nat  $\rightarrow$  PPoint Nat`.

A Type That Depends on a Value

Nothing restricts this to type arguments. A result type can depend on an ordinary value:

```
inductive Sign where
  | pos
  | neg

def posOrNegThree (s : Sign) :
  match s with | Sign.pos => Nat | Sign.neg => Int :=
  match s with
  | Sign.pos => (3 : Nat)
  | Sign.neg => (-3 : Int)

#eval posOrNegThree Sign.pos      -- 3
#eval posOrNegThree Sign.neg     -- -3
```

The type of `posOrNegThree s` is computed from the value `s`: `Nat` for `Sign.pos`, `Int` for `Sign.neg`.

Example: Lists That Know Their Length

A `Vect  $\alpha$  n` is a list of exactly n elements. The length is part of the type:

```
inductive Vect ( $\alpha$  : Type) : Nat  $\rightarrow$  Type where
  | nil : Vect  $\alpha$  0
  | cons :  $\alpha \rightarrow$  Vect  $\alpha$  n  $\rightarrow$  Vect  $\alpha$  (n + 1)

def v2 : Vect Nat 2 := Vect.cons 1 (Vect.cons 2 Vect.nil)
```

Compare the first-element functions for `List` and `Vect`:

```
def List.head' : List  $\alpha \rightarrow \alpha$ 
  | x :: _ => x           -- error: Missing cases: []

def Vect.head : Vect  $\alpha$  (n + 1)  $\rightarrow \alpha$ 
  | Vect.cons x _ => x    -- OK: the empty case cannot happen

#eval v2.head             -- 1
```

Lists That Know Their Length (Continued)

Taking the head of an empty `Vect` is a *type error*, caught before the program runs:

```
#eval (Vect.nil : Vect Nat 0).head
```

```
Application type mismatch: The argument
  Vect.nil
has type
  Vect Nat 0
but is expected to have type
  Vect ?m.5 (?m.6 + 1)
```

- The type `Vect  $\alpha$  (n + 1)` states a fact: “this list is not empty”.
- Types can carry *claims* about values, and type checking *verifies* them.

Propositions Are Types

In Lean, a proposition is a type, and a proof is a value of that type.

```
#check 1 + 1 = 2      -- 1 + 1 = 2 : Prop
#check Nat            -- Nat : Type
```

- `Prop` is the type of propositions, as `Type` is the type of data types.
- To prove P , construct a value of type P . `rfl` proves $a = b$ when both sides evaluate to the same value:

```
theorem two : 1 + 1 = 2 := rfl
```

```
example : plus 3 2 = 5 := rfl
```

```
example : even 4 = true := rfl
```

- `theorem` is `def` for propositions. `example` is an anonymous `theorem`.
- The type checker is the proof checker. There is no separate logic.

The Correspondence

Programming	Logic
type A	proposition A
value of type A	proof of A
function $A \rightarrow B$	implication $A \rightarrow B$
pair $A \times B$	conjunction $A \wedge B$
sum $A \oplus B$	disjunction $A \vee B$
Unit (one value)	True
Empty (no values)	False
dependent function $(x : A) \rightarrow B\ x$	$\forall x : A, B\ x$

This is the *Curry–Howard correspondence*: propositions as types, proofs as programs.

Same Code, Two Readings

Swapping a pair, and swapping a conjunction:

```
def swap (x :  $\alpha \times \beta$ ) :  $\beta \times \alpha$  := (x.2, x.1)
```

```
theorem and_swap (p q : Prop) (h : p  $\wedge$  q) : q  $\wedge$  p := ⟨h.2, h.1⟩
```

- $h : p \wedge q$ is a proof of $p \wedge q$. It is a pair: $h.1$ proves p and $h.2$ proves q .
- $\langle a, b \rangle$ builds a pair (or any structure) from its fields.

```
#check (and_swap)
```

```
-- and_swap :  $\forall$  (p q : Prop), p  $\wedge$  q  $\rightarrow$  q  $\wedge$  p
```

A theorem with variables is a function: $\forall$ is a dependent function type, and $\rightarrow$ is a function type.

Theorems About Programs

```
theorem plus_zero (n : Nat) : plus n 0 = n := rfl

#check plus_zero      -- plus_zero (n : Nat) : plus n 0 = n
#check plus_zero 3    -- plus_zero 3 : plus 3 0 = 3
```

- `plus_zero` is a function from a number `n` to a proof about `n`.
- `rfl` works because `plus n 0` evaluates to `n` by the first case of `plus`.
- `plus 0 n = n` is also true, but `rfl` fails: `plus` matches on its second argument, and `n` is unknown. It needs induction (Part III).

Proofs as Arguments

Since proofs are values, a function can take a proof as an argument. `List.head` requires a proof that the list is not empty:

```
#check @List.head
-- @List.head : {α : Type u_1} → (as : List α) → as ≠ [] → α

#eval [1, 2, 3].head (by decide)      -- 1
#eval ([] : List Nat).head (by decide)
-- error: `decide` proved that the proposition [] ≠ [] is false
```

- `by decide` asks Lean to decide the proposition by computation.
- The type $(as : \text{List } \alpha) \rightarrow as \neq [] \rightarrow \alpha$ depends on the value `as`: an unsafe operation becomes safe, checked at compile time.

Why Termination Matters

Suppose Lean accepted this definition:

```
def loop (n : Nat) : Nat := loop n + 1
```

- Then $\text{loop } n = \text{loop } n + 1$ would hold by definition, and cancelling $\text{loop } n$ would prove $0 = 1$.
- Every proposition would be provable, and proofs would mean nothing.
- This is why Lean insists that recursion terminates: a language whose type checker is a proof checker must be *total*.

Part III

Theorem Proving Basics

Propositions and Connectives

```
variable (p q : Prop)

#check p ∧ q      -- p ∧ q : Prop
#check p → q      -- p → q : Prop
#check ¬p         -- ¬p : Prop
```

	Prove it with	Use a proof h with
$p \rightarrow q$	<code>fun hp => ...</code>	<code>h hp</code>
$p \wedge q$	<code><hp, hq></code>	<code>h.1, h.2</code>
$p \vee q$	<code>Or.inl hp, Or.inr hq</code>	<code>match h with ...</code>
$\neg p (= p \rightarrow \text{False})$	<code>fun hp => ...</code>	<code>absurd hp h</code>

`variable (p q : Prop)` makes `p` and `q` available in the following declarations.

Implication and Conjunction

```
example (hp : p) : q → p := fun hq => hp
```

```
example (hp : p) (hq : q) : p ∧ q := ⟨hp, hq⟩
```

```
example (h : p ∧ q) : q ∧ p := ⟨h.right, h.left⟩
```

Hypotheses are arguments: `hp : p` reads “hp is a proof of p”. `h.left` and `h.right` are `h.1` and `h.2`.

Disjunction and Negation

A proof of $p \vee q$ is either `Or.inl hp` or `Or.inr hq`, so it is consumed by pattern matching:

```
example (hp : p) : p  $\vee$  q := Or.inl hp
```

```
example (h : p  $\vee$  q) : q  $\vee$  p :=  
  match h with  
  | Or.inl hp => Or.inr hp  
  | Or.inr hq => Or.inl hq
```

$\neg p$ is $p \rightarrow \text{False}$. From p and $\neg p$, anything follows:

```
example (hp : p) (hnp :  $\neg p$ ) : q := absurd hp hnp
```

Tactics

Tactics build a proof term step by step, and Lean shows the remaining *goal* at every step:

```
theorem test (p q : Prop) (hp : p) (hq : q) : p ∧ q ∧ p := by
  apply And.intro
  exact hp
  apply And.intro
  exact hq
  exact hp
```

After `apply And.intro`, the Infoview shows two goals:

```
case left
p q : Prop
hp : p
hq : q
⊢ p
```

```
case right
p q : Prop
hp : p
hq : q
⊢ q ∧ p
```

Reading a Goal

```
p q : Prop
hp : p
hq : q
⊢ p
```

- Above $\vdash$: the hypotheses you may use.
- Below $\vdash$: what you must prove.
- A proof is complete when no goals remain.

Put the cursor after a tactic in VS Code and watch the goal change.

Basic Tactics

Tactic	Effect
intro h	goal $p \rightarrow q$: assume $h : p$, prove q
exact e	close the goal with the proof e
apply f	use $f : a \rightarrow b$ to reduce goal b to a
constructor	split goal $p \wedge q$ into p and q
cases h	case analysis on $h : p \vee q$
rfl	both sides are equal by computation

```
example : p → q → p := by
  intro hp hq
  exact hp
```

After intro hp hq:

```
p q : Prop
hp : p
hq : q
⊢ p
```

constructor and cases

```
example (h : p ∧ q) : q ∧ p := by
  constructor
  · exact h.right
  · exact h.left
```

```
example (h : p ∨ q) : q ∨ p := by
  cases h with
  | inl hp => exact Or.inr hp
  | inr hq => exact Or.inl hq
```

- `·` focuses on one goal at a time.
- `cases` on $p \vee q$ gives two goals, one per constructor, just like `match`.

Equality: rfl and rw

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

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

```
example (a b : Nat) : a + b = b + a := by  
  rw [Nat.add_comm]
```

`rw [h]` rewrites the goal with the equation `h` (left to right), and closes it if both sides become identical. Library theorems such as `Nat.add_comm` work the same way.

Quantifiers

$\forall$ is a function type, so it is proved by a function. $\exists$ is a pair of a witness and a proof:

```
example :  $\forall$  n : Nat, plus n 0 = n := fun n => rfl
```

```
example :  $\exists$  n : Nat, n + n = 4 := ⟨2, rfl⟩
```

In tactic mode, `intro n` handles $\forall$, and `exact ⟨2, rfl⟩` handles $\exists$.

Induction

$\text{plus } 0 \ n = n$ is not true by computation. Prove it by induction on n :

```
theorem zero_plus (n : Nat) : plus 0 n = n := by
  induction n with
  | zero => rfl
  | succ k ih => rw [plus, ih]
```

The two goals:

```
case zero
├ plus 0 0 = 0

case succ
k : Nat
ih : plus 0 k = k
├ plus 0 (k + 1) = k + 1
```

ih is the induction hypothesis. $\text{rw } [\text{plus}]$ unfolds one step of plus , giving $(\text{plus } 0 \ k).\text{succ} = k + 1$; then $\text{rw } [\text{ih}]$ finishes.

Induction on Lists

Recursive functions on lists are proved correct by induction on lists:

```
theorem length_append (xs ys : List  $\alpha$ ) :  
  length (xs ++ ys) = length xs + length ys := by  
  induction xs with  
  | nil => simp [length]  
  | cons x xs ih =>  
    simp [length, ih]  
    omega
```

- `simp [length, ih]` simplifies with the definition of `length` and the induction hypothesis; `omega` finishes the arithmetic.
- Note that the proof mirrors the definition of `length`: induction follows recursion.

Automation

```
example : 10 < 20 := by decide
```

```
example (a b : Nat) : a + b = b + a := by omega
```

```
example (xs : List Nat) : (xs ++ []).length = xs.length := by  
  simp
```

Tactic	Proves
decide	decidable propositions, by evaluation
omega	linear arithmetic over Nat and Int
simp	goals that simplification rules reduce to True
bv_decide	machine integers and bit vectors, via SAT

Proofs Are Programs, Really

`#print` shows the term that the tactics built:

```
theorem t1 (hp : p) (hq : q) : p ∧ q := by
  constructor
  · exact hp
  · exact hq

#print t1
-- theorem t1 : ∀ (p q : Prop), p → q → p ∧ q :=
-- fun p q hp hq => ⟨hp, hq⟩
```

Tactics are a convenient way to write the same functional program. The kernel checks only the final term, with the same rules used for `plus` and `length`.

Example (1), Revisited

Every line of Example (1) is now readable. **Specification:** the obviously correct way to reverse a list (slow: ++ copies the list, $O(n^2)$ in total):

```
def reverse : List  $\alpha$   $\rightarrow$  List  $\alpha$ 
| [] => []
| x :: xs => reverse xs ++ [x]
```

Implementation: a fast version with an accumulator, i.e., a loop that moves elements one by one onto acc:

```
def revAcc (xs acc : List  $\alpha$ ) : List  $\alpha$  :=
  match xs with
  | [] => acc
  | x :: xs' => revAcc xs' (x :: acc)

def reverseFast (xs : List  $\alpha$ ) : List  $\alpha$  := revAcc xs []

#eval reverseFast [1, 2, 3]      -- [3, 2, 1]
```

Example (1), Revisited (Continued)

First a lemma about the loop, general in `acc`; then the theorem:

```
theorem revAcc_eq (xs acc : List  $\alpha$ ) :  
  revAcc xs acc = reverse xs ++ acc := by  
  induction xs generalizing acc with  
  | nil => rfl  
  | cons x xs ih => simp [revAcc, reverse, ih]
```

```
theorem reverseFast_correct :  
   $\forall$  xs : List  $\alpha$ , reverseFast xs = reverse xs := by  
  intro xs  
  simp [reverseFast, revAcc_eq]
```

- `generalizing acc`: the recursive call uses a different accumulator (`x :: acc`), so the induction hypothesis must hold for every `acc`. Without it, the proof gets stuck.
- A buggy `revAcc` (say, `acc ++ [x]` instead of `x :: acc`) leaves `simp` stuck at `acc ++ [x] = x :: acc`: the claim is false.

Specification in the Type

The specification can also live in the function's *type*:

```
def reverseFast' (xs : List  $\alpha$ ) : { ys : List  $\alpha$  // ys = reverse xs } :=  
  ⟨revAcc xs [], by simp [revAcc_eq]⟩  
  
#eval (reverseFast' [1, 2, 3]).val      -- [3, 2, 1]
```

- { ys : List α // ys = reverse xs } is the type of lists ys together with a proof that ys = reverse xs. A value of this type is a pair ⟨value, proof⟩.
- Implementing the function now *is* proving the theorem: the type is the specification, and the type checker is the verifier.
- Both styles are the same under propositions-as-types. A separate theorem (previous slide) is the common style: the code stays plain and fast, and one function can have many theorems.

Example (2), Revisited

```
theorem mid_ok :  
   $\forall lo\ hi : \text{UInt32}, lo \leq hi \rightarrow lo \leq \text{mid } lo\ hi \wedge \text{mid } lo\ hi \leq hi :=$  by  
  intro lo hi h  
  unfold mid  
  bv_decide
```

Every line is now readable:

- $\forall lo\ hi : \text{UInt32}, \dots$: a dependent function type; $\rightarrow$ is implication, $\wedge$ is a pair of proofs.
- `intro lo hi h` takes the arguments; `unfold mid` replaces `mid` by its definition.
- `bv_decide` is automation like `decide` and `omega`: it reduces the goal to SAT, and the solver's answer is re-checked in Lean. A counterexample means the claim is false; a proof means it holds for every input.

Summary

- Programs: pure expressions, inductive data, pattern matching, terminating recursion.
- Types can depend on values ($\text{Vect } \alpha \ n$), so the type checker can verify claims about values.
- Propositions are types, proofs are programs: $\rightarrow$ function, $\wedge$ pair, $\vee$ sum, $\forall$ dependent function.
- Tactics build proof terms: `intro`, `exact`, `apply`, `cases`, `rw`, `induction`, `simp`, `omega`.
- Program verification: specification, implementation, and a theorem relating them, proved by induction following the recursion or by automation.

References

Some examples in these slides are adapted from the following books. Code and messages follow Lean 4.33.0.

- D. T. Christiansen, *Functional Programming in Lean*.
https://lean-lang.org/functional_programming_in_lean/
- J. Avigad, L. de Moura, S. Kong, S. Ullrich, *Theorem Proving in Lean 4*.
https://lean-lang.org/theorem_proving_in_lean4/