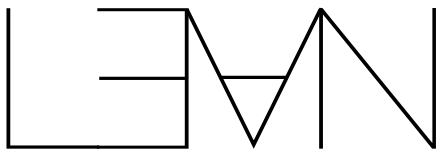
COSE212: Programming Languages

Lecture 21 — Dependent Types (Introduction to 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 α) : List α :=
  match xs with
  | [] => acc
  | x :: xs' => revAcc xs' (x :: acc)

def reverseFast (xs : List α) : List α := 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 α → List α
  | [] => []
  | x :: xs => reverse xs ++ [x]

theorem revAcc_eq (xs acc : List α) :
    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 :
    ∀ xs : List α, reverseFast xs = reverse xs := by
  intro xs
  simp [reverseFast, revAcc_eq]
```

Example (1): A Wrong `reverse`

Suppose the code appended at the wrong end:

```
def revAcc (xs acc : List α) : List α :=
  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 $\mathbf{2^{64}}$ 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:

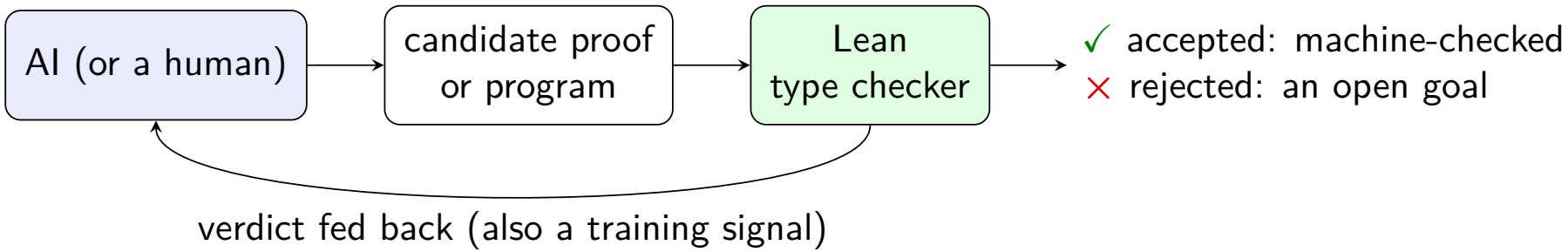
```
def mid (lo hi : UInt32) : UInt32 := lo + (hi - lo) / 2

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
```

[1] J. 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/

Theorem Proving in the Age of AI

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 Software Verification

Mathematics

Aug 2026	Claude raises the proven fraction of Riemann zeta zeros on the critical line from 41.6% (1974) to 67.2%, with a Lean proof.
Sep 2026	Fermat's Last Theorem fully formalized in Lean by Claude agents: 11 days, 13M lines, 29,500 lemmas.
Oct 2026	OpenAI claims a disproof of the Exponential Time Hypothesis, released with a Lean proof one day later; now under expert review.

Software

Sep 2026	Microsoft SymCrypt: post-quantum crypto (ML-KEM, SHA-3) in Rust proved correct in Lean via Aeneas; AI agents write the proofs.
Oct 2026	Anthropic & Theorem: memory safety proved for 123 open-source libraries from OSS-Fuzz, down to RISC-V binaries (in Rocq).

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.

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:
  $\mathbf{2} \times (\mathbf{3} + \mathbf{1}) \implies \mathbf{2} \times \mathbf{4} \implies \mathbf{8}$.
- 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 $\mathbf{0}, \mathbf{1}, \mathbf{2}, \ldots$ 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 : Nat)` abbreviates `(n : Nat) (k : 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?

Polymorphism

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

```
structure PPoint (α : Type) where
  x : α
  y : α
deriving Repr

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

def replaceX (α : Type) (point : PPoint α) (newX : α) : PPoint α :=
  { 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 {α : Type} (point : PPoint α) (newX : α) : PPoint α :=
  { point with x := newX }

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

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

```
def replaceX (point : PPoint α) (newX : α) : PPoint α :=
  { 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 (α : Type) where
  | nil : List α
  | cons : α → List α → List α
```

`[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 α) : 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 (α : Type) : Type where
  | none : Option α
  | some (val : α) : Option α
```

```
def first? (xs : List α) : Option α :=
  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)]
```

Dependent Function Types

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

```
#check (replaceX)
-- replaceX : (α : Type) → PPoint α → α → PPoint α
```

- `(x : A) → 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 → B`.
- The type of `replaceX Nat` is obtained by substituting `Nat` for α: `PPoint Nat → Nat → 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 α n` is a list of exactly `n` elements. The length is part of the type:

```
inductive Vect (α : Type) : Nat → Type where
  | nil : Vect α 0
  | cons : α → Vect α n → Vect α (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 α → α
  | x :: _ => x                -- error: Missing cases: []

def Vect.head : Vect α (n + 1) → α
  | 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` α `(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 → B`	implication `A → B`
pair `A × B`	conjunction `A ∧ B`
sum `A ⊕ B`	disjunction `A ∨ B`
`Unit` (one value)	`True`
`Empty` (no values)	`False`
dependent function `(x : A) → B x`	`∀ x : A, B x`

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

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 Are Programs

`#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`.

Why Termination Matters

Suppose Lean accepted this definition:

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

- Then `loop n = loop n + 1` would hold by definition, and cancelling `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*.

Summary

- Programs: pure expressions, inductive data, pattern matching, terminating recursion.
- Types can depend on values (`Vect` α `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/`