Lean Basics

Friday, 15 May 2026

Terms and Types

The Lean Judgment `a : A`

term : type

A Lean judgment `a : A` says that the expression `a` has type `A`.

Lean first asks whether an expression can be assigned a type. If no type can be found, the expression is rejected before truth or proof becomes relevant.

```
#check 42
-- 42 : ℕ
```

First Typed Expressions

```
#check 42
-- 42 : ℕ
#check "hello"
-- "hello" : String
#check true
-- Bool.true : Bool
#check Nat
-- Nat : Type
#check Bool
-- Bool : Type
#check Type
-- Type : Type 1
```

Type Errors Are Not False

```
#check 42 + 1
-- 42 + 1 : ℕ

-- Expected error:
-- #check 42 + "hi"
-- failed to synthesize instance of type class
--   HAdd ℕ String ?m.5
```

A type error is not a false proposition. The expression has not passed Lean's type checker.

Definitional Equality and `rfl`

Equal by Definition

Two Lean expressions are definitionally equal when the kernel treats them as the same after unfolding definitions and reducing computation.

The first case is simply that an expression is equal to itself.

```
example : 0 = 0 :=
  rfl
-- no goals

example (n : Nat) : n = n :=
  rfl
-- no goals
```

What `rfl` Means

```
#reduce (fun n : Nat ⇒ n + 1) 3
-- 4
#reduce 2 + 3
-- 5

def onePlusOne : Nat := 1 + 1

#reduce onePlusOne
-- 2
```

Lean may recognize two written expressions as the same object after computation and unfolding.

What `rfl` Means

```
example : (fun n : Nat ⇒ n + 1) 3 = 4 :=
  rfl
-- no goals

example : 2 + 3 = 5 :=
  rfl
-- no goals

example : onePlusOne = 2 :=
  rfl
-- no goals
```

`rfl` proves an equality exactly when the two sides are definitionally equal.

Failed `rfl`

```
-- Expected error:
-- example : 2 + 2 = 5 := rfl
-- rfl tactic failed
```

A failed `rfl` attempt means computation alone does not prove the equality.

Inspection Commands: Talking to Lean

A `#` command is a Lean command, not a term in the object language.

`#check`, `#reduce`, and `#print` are diagnostic conversations with Lean's command and meta-programming layer.

```
#check Nat
-- Nat : Type
#reduce 2 + 3
-- 5
```

Three Commands on the Same Object

```
def onePlusOne : Nat := 1 + 1

#check onePlusOne
-- onePlusOne : Nat

#reduce onePlusOne
-- 2

#print onePlusOne
-- def onePlusOne : Nat := 1 + 1
```

`#check` reports the type, `#reduce` computes a normal form, and `#print` reports what the environment stores.

#reduce, #eval, and #print

```
def square (n : Nat) : Nat := n * n

#print square
-- def square : Nat → Nat := fun n ⇒ n * n

#reduce square (2 + 3)
-- 25
```

`#print square` reports the stored declaration. `#reduce square (2 + 3)` computes an applied expression.

#reduce, #eval, and #print

```
def fact : Nat → Nat
  | 0 ⇒ 1
  | n + 1 ⇒ (n + 1) * fact n

#check fact
-- fact : ℕ → ℕ
#reduce fact 5
-- 120
#eval fact 5
-- 120
#eval fact 20
-- 2432902008176640000
```

For larger computations, `#eval` is usually the better classroom command.

Overloaded Operations and Typeclass Instances

Reading the `HAdd` Error

An overloaded notation has a meaning selected from type information.

Lean resolves many overloaded operations by typeclass instance search.

```
-- Expected error:
-- #check 42 + "hi"
-- failed to synthesize instance of type class
--   HAdd ℕ String ?m.5
```

A Local `HAdd` Instance

```
namespace HAddDemo

instance instHAddNatString : HAdd Nat String String where
  hAdd n s := toString n ++ s

#synth HAdd Nat String String
-- instHAddNatString
#check 42 + "hi"
-- 42 + "hi" : String
#eval 42 + "hi"
-- "42hi"

end HAddDemo
```

What Typeclass Search Is Doing Here

`HAdd Nat String String` says that adding a `Nat` and a `String` produces a `String`.

`#synth C` asks Lean to synthesize an instance of the class expression `C`.

This is type-directed instance search. It plays a role similar to interfaces or method dispatch, but it is not class inheritance.

Types Have Types

Types as Lean Expressions

> In Lean, types are themselves expressions with types.

```
#check Nat
-- Nat : Type
#check Bool
-- Bool : Type
#check true
-- Bool.true : Bool
#check false
-- Bool.false : Bool
#check Prop
-- Prop : Type
```

Universe Levels, Named but Not Developed

```
#check Type
-- Type : Type 1
#check Type 1
-- Type 1 : Type 2
```

Lean can reason about types because types are part of the same typed language.

To avoid a single "type of all types", Lean stratifies universes.

Universe Arithmetic in Lean

```
-- Sort 0 is Prop
-- Sort 1 is Type
-- Sort 2 is Type 1

#check Prop
-- Prop : Type
#check Type
-- Type : Type 1
#check Type 1
-- Type 1 : Type 2
```

Use Lean output, not set-containment language, to explain what lives at each universe level.

Universe Arithmetic in Lean

```
#check Type → Type
-- Type → Type : Type 1
#check Type × Type
-- Type × Type : Type 1
#check Sigma (fun A : Type ⇒ A)
-- (A : Type) × A : Type 1
#check (ULift.{1,0} Nat)
-- ULift Nat : Type 1
```

Examples in `Type 1` include type constructors and types that quantify over small types.

Universe Arithmetic in Lean

```
universe u v
variable (A : Type u) (B : Type v)

#check A → B
-- A → B : Type (max u v)
#check A × B
-- A × B : Type (max u v)
#check A ⊕ B
-- A ⊕ B : Type (max u v)
#check Type → Type
-- Type → Type : Type 1
```

Function, product, and sum types obey a maximum rule; they do not automatically raise the universe level.

Type Theory Avoids Russell's Paradox

In type theory, `a : A` is a judgment, not an internal proposition `a ∈ A` inside one universal set of all objects.

Universes are stratified: `Type u` itself lives in `Type (u+1)`. There is no closed universe `U : U`.

Type Theory Avoids Russell's Paradox

```
#check Type
-- Type : Type 1
#check Type 1
-- Type 1 : Type 2
```

Russell's paradox needs a domain that contains all objects under discussion and lets objects act as collections of objects from that same domain.

Lean's `Set α` is Typed

```
#check Set
-- Set.{u} (α : Type u) : Type u

#check ({n : Nat | n ≠ n} : Set Nat)
-- {n | n ≠ n} : Set ℕ
```

> The error `Membership Type Type` is only a Lean symptom. The foundational reason is universe stratification and typed membership.

Propositions in Lean

Propositions and Proof Terms

A proposition is an expression of type `Prop`. A proof of proposition `P` is a term whose type is `P`.

```
variable (P Q : Prop)

#check P
-- P : Prop
#check P → Q
-- P → Q : Prop
#check 2 + 3 = 5
-- 2 + 3 = 5 : Prop
```

Decidable Propositions and `Bool`

```
variable (P : Prop)

#check decide (2 + 3 = 5)
-- decide (2 + 3 = 5) : Bool
#eval decide (2 + 3 = 5)
-- true
#eval decide (2 + 3 = 6)
-- false

-- Expected error:
-- #check decide P
-- failed to synthesize instance of type class
--   Decidable P
```

Classical Logic and Decidability

```
variable (P : Prop)

#check Classical.propDecidable
-- Classical.propDecidable (a : Prop) : Decidable a
#check Classical.propDecidable P
-- Classical.propDecidable P : Decidable P

noncomputable def classicalDecision (P : Prop) : Bool :=
  @decide P (Classical.propDecidable P)

#check classicalDecision P
-- classicalDecision P : Bool
```

Classical logic makes every proposition decidable, but the decision is noncomputable.

`Bool` versus `Prop`

`Bool`

Computational data with values `true` and `false`.

`Prop`

A proposition is something to be proved; its terms are proofs.

`decide P : Bool` exists only when Lean has a `Decidable P` instance.

The Smallest Proof-Term Example

```
example (P : Prop) (hp : P) : P :=
  hp
-- no goals
```

The theorem is checked by checking that the returned term has the proposition as its type.

Term Proofs and Tactic Proofs

Term proof

Write the proof term directly after `:=`.

Tactic proof

Write `by`, then use tactics to construct a proof term.

In both styles, the kernel finally checks a term whose type is the target proposition.

Term Proofs and Tactic Proofs

```
example (P Q : Prop) (hp : P) (f : P → Q) : Q :=
  f hp
-- no goals

example (P Q : Prop) (hp : P) (f : P → Q) : Q := by
  exact f hp
-- no goals
```

> `by exact t` is the smallest bridge between tactic style and term style.

Term Proofs and Tactic Proofs

```
example (P Q : Prop) (f : P → Q) : P → Q :=
  fun hp ⇒ f hp
-- no goals

example (P Q : Prop) (f : P → Q) : P → Q := by
  intro hp
  exact f hp
-- no goals
```

In tactic style, Lean maintains a goal state while the proof term is being built.

Tactic Mode: Constructing Proof Terms Step by Step

What Tactic Mode Is

A proof block `by …` enters tactic mode. Lean displays a current goal and local hypotheses; each tactic transforms the proof state.

The completed script elaborates to a proof term checked by the kernel.

The First `intro` / `exact` Proof

```
example (P Q : Prop) (f : P → Q) : P → Q := by
  -- P Q : Prop
  -- f : P → Q
  -- ⊢ P → Q
  intro hp
  -- hp : P
  -- ⊢ Q
  exact f hp
  -- no goals
```

First Reading of `intro` and `exact`

`intro hp` turns the target `P → Q` into a new local hypothesis `hp : P` and a smaller goal `Q`.

`exact t` closes the current goal when `t` is a term of the target type.

This is only the first reading. The full rule returns later with functions, implication, `Π`, and `∀`.

A List of Tactics

tactic	first classroom role
`intro x`	start a function or implication proof
`exact t`	close a goal with an explicit term
`rfl`	close definitional-equality goals
`cases h`	split products, sums, conjunctions, or disjunctions
`exfalso`	reduce a proposition goal to `False`
`rw [h]`	rewrite by an equality
`simp [ ... ]`	simplify routine expressions and goals
`decide`	close decidable propositions by computation

A List of Tactics

tactic	developed later with
`intro`, `exact`	function types, implication, `Π`, `∀`
`rfl`	definitional equality
`cases`	products, sums, conjunction, disjunction
`exfalso`	negation and `False.elim`
`rw`, `simp`	equality manipulation
`decide`	`Bool`, `Prop`, decidable propositions

Function Types are Pi Types

Ordinary Function Types

A term of type `A → B` turns any term of type `A` into a term of type `B`.

```
variable {A B C : Type}
variable (F : A → B) (G : B → C) (a : A)

#check (Nat → Bool)
-- ℕ → Bool : Type
#check F
-- F : A → B
#check F a
-- F a : B
```

Function Application and Construction

```
#check G (F a)
-- G (F a) : C
#check G ∘ F
-- G ∘ F : A → C

example : Nat → Nat :=
  fun n ⇒ n + 1
-- no goals

#eval (fun n : Nat ⇒ n + 1) 11
-- 12
```

Type Binders and Term Binders

`A → B`, `(x : A) → B x`, and `Π x : A, B x` are type expressions.

`fun x ⇒ t` is a term expression. It constructs an inhabitant of a function type.

The binders are related, but not synonyms: one appears in types, the other constructs terms.

Implication as a Function Type

In `Prop`, a function `P → Q` maps proofs of `P` to proofs of `Q`.

```
example (P : Prop) : P → P :=
  fun hp ⇒ hp
-- no goals

example (P Q : Prop) (f : P → Q) (hp : P) : Q :=
  f hp
-- no goals
```

Implication as a Function Type

```
example (P Q R : Prop) (f : P → Q) (g : Q → R) : P → R :=
  fun hp ⇒ g (f hp)
-- no goals
```

If assuming `hp : P` lets us build a term of type `Q`, then `fun hp ⇒ ... : P → Q`.

The Same Proof in Tactic Mode

```
example (P Q : Prop) (f : P → Q) : P → Q := by
  intro hp
  exact f hp
-- no goals
```

`intro hp` is function construction in tactic mode.

Negation as a Function to `False`

```
variable (P : Prop)

#check ¬ P
-- ¬P : Prop
#check P → False
-- P → False : Prop

example (P : Prop) (h : P → False) : ¬ P :=
  h
-- no goals

example (P : Prop) (h : ¬ P) : P → False :=
  h
-- no goals
```

Double Negation

```
variable (P : Prop)

#check ¬¬ P
-- ¬¬P : Prop

#check (P → ¬¬ P)
-- P → ¬¬P : Prop

example : P → ¬¬ P := by
  intro hp hnot
  exact hnot hp
-- no goals

theorem doubleNegIntro (P : Prop) : P → ¬¬ P := by
  intro hp hnot
  exact hnot hp

#print axioms doubleNegIntro
-- 'DSA5210.Session02.doubleNegIntro' does not depend on any axioms
```

Parentheses Matter

```
example : ¬ ¬ P = P := by
  intro h
  exact h rfl
-- no goals

example : ¬¬ (P = P) := by
  intro h
  exact h rfl
-- no goals

#check ((¬¬ P) = P)
-- (¬¬P) = P : Prop
```

Lean parses `¬ ¬ P = P` as `¬¬ (P = P)`, not as `(¬¬ P) = P`.

Double Negation Elimination

```
variable (P : Prop)

#check (¬¬ P → P)
-- ¬¬P → P : Prop

theorem doubleNegElim (P : Prop) : ¬¬ P → P := by
  classical
  intro hnn
  by_contra hnot
  exact hnn hnot
-- no goals

#print axioms doubleNegElim
-- 'DSA5210.Session02.doubleNegElim' depends on axioms: [propext, Classical.choice, Quot.sound]
```

The reverse direction is double-negation elimination. It needs classical logic: `¬¬ P` refutes every refutation of `P`, but it does not construct a proof of `P` by definition.

The Function `False.elim`

```
variable (P : Prop)

#check False.elim
-- False.elim.{u} {C : Sort u} (h : False) : C

#check (False.elim : False → P)
-- False.elim : False → P

#check (False.elim : False → Nat)
-- False.elim : False → ℕ
```

`False.elim` is the eliminator for `False`. Since `False` has no constructors, an impossible proof of `False` has no cases to handle.

`False.elim` and `exfalso`

```
example (P Q : Prop) (hnot : ¬ P) (hp : P) : Q :=
  False.elim (hnot hp)
-- no goals

example (P Q : Prop) (hnot : ¬ P) (hp : P) : Q := by
  exfalso
  -- ⊢ False
  exact hnot hp
-- no goals
```

Once a proof of `False` is obtained, any proposition can be proved by elimination.

The tactic `exfalso` changes the current goal to `False`; the remaining task is to produce the contradiction.

Product Types and Conjunction

Product Types as Data

> A product type `A × B` contains one term of `A` and one term of `B`.

```
def pair : Nat × Bool := (3, true)

#check pair.1
-- pair.1 : ℕ
#check pair.2
-- pair.2 : Bool
#eval pair.1
-- 3
#eval pair.2
-- true
```

Conjunction as Product-Like Proof Data

A proof of `P ∧ Q` contains a proof of `P` and a proof of `Q`.

```
example (P Q : Prop) (hp : P) (hq : Q) : P ∧ Q :=
  And.intro hp hq
-- no goals

example (P Q : Prop) (hp : P) (hq : Q) : P ∧ Q :=
  ⟨hp, hq⟩
-- no goals
```

Constructors and Projections

```
example (P Q : Prop) (h : P ∧ Q) : P :=
  h.left
-- no goals

example (P Q : Prop) (h : P ∧ Q) : Q :=
  h.right
-- no goals
```

`And.intro hp hq` shows the constructor explicitly. `⟨hp, hq⟩` lets Lean infer the constructor from the expected type.

Sum Types and Disjunction

Sum Types as Tagged Data

A sum type `A ⊕ B` is a tagged disjoint union: either `inl a` or `inr b`.

```
#check Sum
-- Sum.{u, v} (α : Type u) (β : Type v) : Type (max u v)
#check (Nat ⊕ String)
-- ℕ ⊕ String : Type

example : Nat ⊕ String := .inl 5
-- no goals

example : Nat ⊕ String := .inr "hi"
-- no goals
```

Constructor Tags Are Part of the Value

```
theorem inlOne_ne_inrOne :
    (Sum.inl 1 : Nat ⊕ Nat) ≠ Sum.inr 1 := by
  intro h
  cases h
-- no goals
```

`Sum.inl 1` and `Sum.inr 1` are different values even though the visible number is the same.

Constructor Tags Are Part of the Value

```
theorem inlOne_ne_inrOne :
    (Sum.inl 1 : Nat ⊕ Nat) ≠ Sum.inr 1 := by
  -- ⊢ Sum.inl 1 ≠ Sum.inr 1
  intro h
  -- h : Sum.inl 1 = Sum.inr 1
  -- ⊢ False
  cases h
  -- no goals
```

The proof compares the tags directly: left injection and right injection are different constructors.

Disjunction as Proof of One Side

```
example (P Q : Prop) (hp : P) : P ∨ Q :=
  Or.inl hp
-- no goals

example (P Q : Prop) (hq : Q) : P ∨ Q :=
  Or.inr hq
-- no goals
```

The data-level construction is `Sum`; the proposition-level construction is `Or`.

Sum Data versus Or Proofs

```
example (P Q : Prop) (h1 h2 : P ∨ Q) : h1 = h2 :=
  Subsingleton.elim h1 h2
-- no goals

example (P Q : Prop) (hp : P) (hq : Q) :
    (Or.inl hp : P ∨ Q) = Or.inr hq :=
  Subsingleton.elim (Or.inl hp) (Or.inr hq)
-- no goals
```

The second example says: `hp : P` gives one proof of `P ∨ Q`, and `hq : Q` gives another proof of `P ∨ Q`; Lean regards these two proof terms as equal.

This is the contrast with `Sum`: data keeps the left/right tag, but proofs in `Prop` are proof-irrelevant.

Types Depending on Values

`Fin n` as the First Dependent Type Example

A dependent type is a type expression whose result depends on a value. `Fin : Nat → Type` sends `n` to `Fin n`.

```
#check Fin
-- Fin (n : ℕ) : Type
#check Fin 0
-- Fin 0 : Type
#check Fin 1
-- Fin 1 : Type
#check Fin 3
-- Fin 3 : Type
#check (2 : Fin 3)
-- 2 : Fin 3
#eval ((2 : Fin 3).val)
-- 2
```

Changing the Index

```
#check Fin.castSucc
-- Fin.castSucc {n : ℕ} : Fin n → Fin (n + 1)

#eval (Fin.castSucc (2 : Fin 3)).val
-- 2
```

The stored natural number does not change, but the result type changes from `Fin n` to `Fin (n + 1)`.

`Fin` and Subtype

```
#check Fin.equivSubtype
-- Fin.equivSubtype {n : ℕ} : Fin n ≃ { i // i < n }

#eval (Fin.equivSubtype (2 : Fin 3)).val
-- 2

#eval (Fin.equivSubtype.symm (⟨2, by decide⟩ : {i : Nat // i < 3})).val
-- 2
```

> This is an equivalence, not definitional equality by `rfl`.

Π and ∀

Dependent Pi Types

`Π x : A, B x` is the type of functions whose output type may depend on the input value.

```
#check ((n : Nat) → Fin n)
-- (n : ℕ) → Fin n : Type
#check ((n : Nat) → Fin (n + 1))
-- (n : ℕ) → Fin (n + 1) : Type
#check (Π n : Nat, Fin (n + 1))
-- (n : ℕ) → Fin (n + 1) : Type

example : (Π n : Nat, Fin (n + 1)) :=
  fun n ⇒ ⟨0, Nat.succ_pos n⟩
-- no goals
```

Universal Quantification

```
#check (∀ n : Nat, n = n)
-- ∀ (n : ℕ), n = n : Prop

example : (∀ n : Nat, n = n) :=
  fun n ⇒ Eq.refl n
-- no goals

example : (∀ n : Nat, n = n) := by
  intro n
  rfl
-- no goals
```

Π, ∀, and `fun`

`Π` and `∀` are type-level binders. They specify the expected dependent function or proof type.

`fun` is the term-level binder. It constructs the function or proof.

When the codomain is a proposition, Lean usually writes the Pi type as a universal quantifier.

Applying a Universal Proof

```
example (P : Nat → Prop) (h : ∀ n : Nat, P n) : P 3 :=
  h 3
-- no goals
```

A proof of `∀ n, P n` is a function. Applying it to `3` gives a proof of `P 3`.

Σ, Subtype, and ∃

Three Forms of Existence-Like Data

Σ	Subtype	∃
data-level dependent pair	value plus proof of a property	proposition-level existence

They look similar, but they are not interchangeable.

Σ as Data-Level Dependent Pair

```
example : (Σ n : Nat, Fin n.succ) :=
  ⟨2, ⟨1, by decide⟩⟩
-- no goals

def twoSigma : (Σ n : Nat, Fin (n + 1)) :=
  ⟨2, ⟨1, by decide⟩⟩

def threeSigma : (Σ n : Nat, Fin (n + 1)) :=
  ⟨3, ⟨1, by decide⟩⟩
```

Where Σ Type-Checks

```
#check (Σ n : Nat, Fin (n + 1))
-- (n : ℕ) × Fin (n + 1) : Type

-- Expected error:
-- #check (Σ n : Nat, n > 5)
-- Type mismatch
--   n > 5
-- has type
--   Prop
-- but is expected to have type
--   Type ?u
```

Subtype as Value Plus Property

```
def sixSubtype : {n : Nat // n > 5} :=
  ⟨6, by decide⟩

#eval sixSubtype.val
-- 6
#check sixSubtype.property
-- sixSubtype.property : ↑sixSubtype > 5
```

The value remains computational data; the property is stored as a proof.

∃ as Proposition-Level Existence

```
def sixExists : ∃ n : Nat, n > 5 :=
  ⟨6, by decide⟩

def sevenExists : ∃ n : Nat, n > 5 :=
  ⟨7, by decide⟩

example : sixExists = sevenExists :=
  Subsingleton.elim sixExists sevenExists
-- no goals
```

Where ∃ Type-Checks

```
#check (∃ n : Nat, n > 5)
-- ∃ n, n > 5 : Prop

-- Expected error:
-- #check (∃ n : Nat, Fin (n + 1))
-- Type mismatch
--   Fin (n + 1)
-- has type
--   Type
-- but is expected to have type
--   Prop
```

Distinctions to Preserve

Σ

keeps witness data inspectable

Subtype

keeps a value with a proposition-valued property

∃

proves existence in proof-irrelevant `Prop`

Use a subtype, not `∃`, when the witness is meant to remain computational data.

Curry-Howard: The Pattern Behind the Type Formers

Name the Pattern After the Examples

Logic	Lean/type-theoretic reading
proposition	a type-like object in `Prop`
proof	term inhabiting that proposition
theorem	named proof term
implication	function from proofs to proofs
conjunction	pair of proofs
disjunction	tagged proof of one side
universal quantifier	dependent function
existential quantifier	proposition-level existence

The Careful Version for Lean

Curry-Howard explains why proofs can be terms and why theorem checking is type checking.

Lean keeps computational data in `Type` and proof-irrelevant propositions in `Prop`.

Do not say that all Lean types are propositions, or that `∃` is just `Σ`.

Parallel Constructions in `Type` and `Prop`

Product	Sum	Dependent
`A × B`	`A ⊕ B`	`Σ x, B x`
`P ∧ Q`	`P ∨ Q`	`∃ x, P x`

Definitions and Parameters

Definitions Name Terms

A Lean definition `def name : A := t` adds a named term `name` of type `A` to the environment.

```
def square (n : Nat) : Nat := n * n

#check square
-- square (n : ℕ) : ℕ
#eval square 5
-- 25
#reduce square (2 + 3)
-- 25
```

What the Environment Stores

```
#print square
-- def square : Nat → Nat := fun n ⇒ n * n
```

Once a definition is in the environment, other definitions and proofs can refer to the name.

Theorem Statements as Function Types

```
theorem my_and_intro {P Q : Prop} (hp : P) (hq : Q) : P ∧ Q :=
  And.intro hp hq

theorem my_and_intro' : ∀ {P Q : Prop}, P → Q → P ∧ Q :=
  fun hp hq ⇒ And.intro hp hq

#check my_and_intro
-- my_and_intro {P Q : Prop} (hp : P) (hq : Q) : P ∧ Q
#check (∀ {P Q : Prop}, P → Q → P ∧ Q)
-- (∀ {P Q : Prop}, P → Q → P ∧ Q) : Prop
```

A theorem is also a named term, but its type is a proposition.

Explicit and Implicit Parameters

```
def idExplicit (α : Type) (x : α) : α := x
def idImplicit {α : Type} (x : α) : α := x

#check idExplicit Nat 3
-- idExplicit ℕ 3 : ℕ
#check idImplicit 3
-- idImplicit 3 : ℕ
#check @idImplicit
-- @idImplicit : {α : Type} → α → α
#check idExplicit _ 3
-- idExplicit ℕ 3 : ℕ
#check @idImplicit _ 3
-- idImplicit 3 : ℕ
example {P Q : Prop} (hp : P) (hq : Q) : P ∧ Q :=
  @my_and_intro _ _ hp hq
```

The placeholder `_` asks Lean to infer an argument that is present syntactically.

Heterogeneous Equality and Changing Types

n + 1 and 1 + n

> These expressions are mathematically equal, but not definitionally equal.

```
#reduce (n + 1)
-- n.succ
#reduce (1 + n)
-- (Nat.rec ⟨fun x ⇒ x, PUnit.unit⟩
--   (fun n n_ih ⇒ ⟨fun x ⇒ (n_ih.1 x).succ, n_ih⟩) n).1 1

-- Expected error:
-- #check (show n + 1 = 1 + n from rfl)
-- Type mismatch
--   rfl
-- has type
--   ?m.16 = ?m.16
-- but is expected to have type
--   n + 1 = 1 + n
```

Arithmetic Equality as a Term

For this equality, Lean needs a theorem rather than definitional computation.

```
#check Nat.add_comm n 1
-- Nat.add_comm n 1 : n + 1 = 1 + n

#check congrArg Fin (Nat.add_comm n 1)
-- congrArg Fin (Nat.add_comm n 1) : Fin (n + 1) = Fin (1 + n)
```

`congrArg Fin` applies the equality of indices to the type former `Fin`.

`HEq` Compares Across Types

`HEq x y`, printed as `x ≍ y`, is heterogeneous equality: the two sides may have different types.

```
#check HEq
-- HEq.{u} {α : Sort u} : α → {β : Sort u} → β → Prop

#check HEq.rfl
-- HEq.rfl.{u} {α : Sort u} {a : α} : a ≍ a

#check HEq (2 : Fin 3) (2 : Fin 4)
-- 2 ≍ 2 : Prop
```

The `Fin` Example

```
#check (0 : Fin (n + 1))
-- 0 : Fin (n + 1)

#check (0 : Fin (1 + n))
-- 0 : Fin (1 + n)

#check HEq (0 : Fin (n + 1)) (0 : Fin (1 + n))
-- 0 ≍ 0 : Prop
```

The stored value is `0` on both sides, but the two displayed types have different indices.

Transport with `cast`

```
#check cast (congrArg Fin (Nat.add_comm n 1)) (0 : Fin (n + 1))
-- cast ⋯ 0 : Fin (1 + n)

example :
    cast (congrArg Fin (Nat.add_comm n 1)) (0 : Fin (n + 1)) =
      (0 : Fin (1 + n)) := by
  rw [← Fin.cast_eq_cast]
  exact Fin.cast_mk (Nat.add_comm n 1) 0 (by omega)
-- no goals
```

Prefer ordinary equality after explicit transport when the types can be lined up.

Proof of the HEq Statement

```
example : HEq (0 : Fin (n + 1)) (0 : Fin (1 + n)) := by
  apply (cast_eq_iff_heq (e := congrArg Fin (Nat.add_comm n 1))).mp
  rw [← Fin.cast_eq_cast]
  exact Fin.cast_mk (Nat.add_comm n 1) 0 (by omega)
-- no goals
```

> Convert HEq to equality after transport; then use `Fin.cast_mk`.

Local Lean Demo Files

How to Check the Local Lean Files

```
cd /Users/hoxide/mydoc/ai4mathnus
./scripts/check-nus-session02-lean.sh
```

> The slide deck should show short excerpts; the Lean demo files are the checked source of the classroom examples.

Files Used During Lecture

```
nus-dsa/lean/Session02/Demo.lean
nus-dsa/lean/Session02/Scratch.lean
nus-dsa/lean/Session02/README.md
```

> Use `Demo.lean` for the complete path through the lecture and `Scratch.lean` for live reconstruction.