Inductive Types, Structures, Typeclasses, and Termination in Lean

Lean deep dive: data, proof, interfaces, and recursive definitions

AI for Mathematics - Week 7

Ma Jiajun - May 19, 2026

Structures Package Data and Proof Fields

A `structure` Packages Data

`structure`

A structure declares a new type with named fields, a constructor, projections, and printable values when `Repr` is derived.

- `Point2D.mk` constructs points.
- `⟨1, 1⟩` uses anonymous constructor syntax.
- `p.x` projects a field.

```
structure Point2D where
  x : Int
  y : Int
  deriving Repr

#check Point2D.mk
-- Point2D.mk (x y : Int) : Point2D

#eval Point2D.mk 1 2
-- { x := 1, y := 2 }

def p : Point2D where
  x := 3
  y := -2

def q : Point2D := ⟨1, 1⟩

#print q
-- def q : Point2D := { x := 1, y := 1 }

#eval q
-- { x := 1, y := 1 }

#eval p.x + p.y
-- 1
```

Dot Notation Gives Method-Style Calls

```
def Point2D.sumxy (p : Point2D) : Int :=
  p.x + p.y

def Point2D.add (p q : Point2D) : Point2D :=
  (p.x + q.x, p.y + q.y)

#eval p.sumxy
-- 1

#eval p.add q
-- { x := 4, y := -1 }
```

OOP analogy

`p.sumxy` looks like a method call.

Lean meaning

`Point2D.sumxy` is an ordinary function; dot notation inserts `p` as the first argument.

`ClosedInterval.valid` Is a Proof Field

A structure can mix computational fields and proof fields.

- `left` and `right` are data.
- `valid` is proof that `left ≤ right`.
- The invariant is checked when the value is constructed.

```
import Mathlib

structure ClosedInterval where
  left : Nat
  right : Nat
  valid : left ≤ right

variable (n : Nat)

def I1 : ClosedInterval where
  left := 3
  right := 5
  valid := by decide

def intervalFrom (n : Nat) : ClosedInterval where
  left := n
  right := n + 2
  valid := by simp only [le_add_iff_nonneg_right, zero_le]

#check intervalFrom
-- intervalFrom (n : Nat) : ClosedInterval

#check (intervalFrom n).valid
-- (intervalFrom n).valid :
--   (intervalFrom n).left ≤ (intervalFrom n).right
```

Stored Proof Fields Can Be Used Later

```
example (I : ClosedInterval) :
    ∃ i, I.left ≤ i ∧ i ≤ I.right := by
  use I.left
  constructor
  · simp only [Std.le_refl]
  · exact I.valid
```

Witness

Choose `i = I.left`.

Stored proof

The upper bound proof is exactly `I.valid`.

`Fin n` Is Value Plus Bound Proof

```
import Mathlib

#check Fin

#print Fin
-- structure Fin (n : Nat) : Type
-- fields: val : Nat, isLt : val < n

def secondOfFive : Fin 5 :=
  (2, by decide)

-- Expected error:
-- #check (2 : Fin 5) = (2 : Fin 6)

#check (2 : Fin 5) = (2 : Nat)
-- ↑2 = 2 : Prop

#eval (Finset.univ : Finset (Fin 5))
-- {0, 1, 2, 3, 4}

#check (2 : Fin 5)
-- 2 : Fin 5

#synth OfNat (Fin 5) 2
-- Fin.instOfNat
```

Interpretation

A value of `Fin 5` is a natural number plus a proof that it is less than `5`.

Type boundary

`Fin 5` and `Fin 6` are different types; comparison with `Nat` uses coercion.

Classes and Instance Search

Classes are structures Lean can find automatically.

The Problem After Overloaded Notation

Lean sees an overloaded symbol

`42 + "hi"` asks for an addition interface between `Nat` and `String`.

Lean searches for an instance

If no matching instance exists, elaboration fails.

```
 ordinary OOP class: object carries methods and state

Lean class: type has an interface, supplied separately by an instance
```

Notation Elaborates to Terms: Addition

```
#eval 42 + 2
-- 44

#check HAdd
-- HAdd (alpha : Type u) (beta : Type v)
--   (gamma : outParam (Type w)) : Type _

#check HAdd.hAdd
-- [HAdd alpha beta gamma] → alpha → beta → gamma

#synth HAdd Nat Nat Nat
-- instHAdd
```

`+`

Uses an `HAdd` instance.

Point

The same symbol can mean different operations on different types.

Custom HAdd Instance

```
/-
instance haddnatstring1 : HAdd Nat String Nat where
  hAdd := fun a b ⇒ a + b.length
-/

instance haddnatstring2 : HAdd Nat String String where
  hAdd := fun a b ⇒ (toString a) ++ b

-- #synth HAdd Nat String Nat
#synth HAdd Nat String String
-- haddnatstring2

#eval 42 + "hi"
-- "42hi"
```

The result type is part of the `HAdd` interface: `HAdd Nat String String` is different from `HAdd Nat String Nat`.

Coercion and Expected Type

```
instance stringToNat : Coe String Nat where
  coe s := s.length

#eval ("hi" : Nat)
-- 2

#eval (42 + "hi" : Nat)
-- 44

#eval (42 + "hi" : String)
-- "42hi"
```

Expected type

`String` asks for the custom `HAdd Nat String String` instance.

`Coe String Nat`

`Nat` lets Lean coerce `"hi"` to its length, then use ordinary `Nat + Nat`.

Numeric Literals Use `OfNat`

```
#check OfNat
-- OfNat (α : Type u) : Nat → Type u

#synth OfNat (Fin 5) 2
-- Fin.instOfNat

#check (2 : Fin 5)
-- 2 : Fin 5
```

`OfNat (Fin 5) 2`

The expected type `Fin 5` tells Lean how to interpret the numeral `2`.

Not `Coe Nat (Fin 5)`

`OfNat` handles numeric literals. `Coe A B` handles automatic conversion from values of type `A` to values of type `B`.

Notation Elaborates to Terms: Order

```
#check LE
-- LE (alpha : Type u) : Type u

#check LE.le
-- [LE alpha] → alpha → alpha → Prop

#synth HAdd Nat Nat Nat
-- instHAdd

#synth LE Nat
-- instLENat
```

`≤`

Uses an `LE` instance.

Instance search

`#synth LE Nat` asks Lean which order interface is available for `Nat`.

Mathlib's `Preorder` Extends `LE`

```
#check Preorder
-- Preorder (alpha : Type u) : Type u

#check le_rfl
-- le_rfl [Preorder alpha] : a ≤ a

#check le_trans
-- le_trans [Preorder alpha] :
--   a ≤ b → b ≤ c → a ≤ c
```

A preorder is a relation `≤` that is reflexive and transitive.

- `LE alpha` supplies the relation.
- `LT alpha` supplies strict order notation.
- `le_refl` supplies reflexivity.
- `le_trans` supplies transitivity.

Preorder Examples

Natural numbers

Ordinary numerical `≤`.

Sets

Subset inclusion.

Tasks

`a ≤ b` means `a` can reach `b` in a dependency graph.

Conversions

`a ≤ b` means `a` can be converted into `b`.

If the graph has cycles, two different tasks may reach each other: preorder, not necessarily partial order.

Generic Theorems Ask for a Class

```
example [Preorder alpha] (a : alpha) : a ≤ a :=
  le_rfl

example [Preorder alpha] {a b c : alpha}
    (hab : a ≤ b) (hbc : b ≤ c) : a ≤ c :=
  le_trans hab hbc
```

- Parentheses `(x : A)` introduce ordinary explicit data.
- Square brackets `[C alpha]` introduce typeclass assumptions.
- Lean tries to fill square-bracket arguments by instance search.

Diamond Inheritance Affects Instance Search

Shape

One class may inherit two parent interfaces that both inherit the same lower interface.

Risk

If the two paths carry different inherited data, instance search may depend on which path Lean follows.

```
Child / \ Parent₁ Parent₂ \ / Shared
```

Mathlib hierarchies are designed so shared inherited structures stay coherent.

Inductive Types

Constructors are generation rules.

Constructors Generate Values

An inductive type is generated by its constructors.

- Constructors are the canonical ways to build values.
- Every value is built from finitely many constructor applications.
- To consume an inductive value, split into the constructor cases.

```
inductive MyBool where
  | false : MyBool
  | true : MyBool

#check MyBool.false
-- MyBool.false : MyBool

#check MyBool.true
-- MyBool.true : MyBool
```

Binary Trees Use `nil` and `node`

Generation rules

- `nil : MyTree alpha`.
- If `a : alpha` and `l r : MyTree alpha`, then `node a l r : MyTree alpha`.
- There are no other basic ways to make a `MyTree alpha`.

```
inductive MyTree (alpha : Type) where
  | nil : MyTree alpha
  | node : alpha → MyTree alpha → MyTree alpha → MyTree alpha
  deriving Repr
```

Concrete Binary Trees

`t1`

`leftOnly`

`rightOnly`


```
namespace MyTree

def t1 : MyTree Nat :=
  node 3 nil nil

def leftOnly : MyTree Nat :=
  node 2 (node 1 nil nil) nil

def rightOnly : MyTree Nat :=
  node 2 nil (node 3 nil nil)

def t3 : MyTree Nat :=
  node 5 leftOnly rightOnly

end MyTree
```

Bad Tree Definition: `leaf` and `node`

```
inductive BadTree (alpha : Type) where
  | leaf : alpha → BadTree alpha
  | node : BadTree alpha → BadTree alpha → BadTree alpha
  deriving Repr
```

What it defines

A full binary tree: every internal node has both left and right subtrees.

What it cannot express

A node with only a left child, or only a right child.

The code typechecks, but the datatype does not match the intended ordinary binary tree.

Constructors Also Determine Proof Shape

Computation side

Pattern matching follows the constructor list.

One branch for `nil`, one branch for `node`.

Proof side

Induction follows the same constructor list.

Recursive constructor arguments produce induction hypotheses.

Induction is not only for natural numbers. Every inductive type has its own induction principle.

Natural-Number Induction

`Nat` Is Generated by `zero` and `succ`

```
-- Conceptual shape:
-- inductive Nat where
--   | zero : Nat
--   | succ : Nat → Nat
```

```
 0 : Nat succ 0 : Nat succ (succ 0)
: Nat ...
```

Natural-number induction is the proof principle generated by these two constructors.

Natural-Number Induction Principle

To prove `P n` for every `n : Nat`:

1. prove `P 0`,
2. prove `P (n + 1)`, assuming `P n`.

```
-- Conceptual type:
P 0
→ (forall n, P n → P (n + 1))
→ forall n, P n
```

The induction hypothesis comes from the recursive argument of the successor constructor.

The Proposition Family Is the Motive

For the theorem below, the proposition family is:

```
P i := i + 1 = 1 + i
```

Induction proves `P i` for an arbitrary `i : Nat`.

```
theorem add_one_eq_one_add (i : Nat) :
    i + 1 = 1 + i := by
  induction i with
  -- two proof branches
```

`induction i with` Creates Two Branches

`zero` branch

```
goal: P 0
```

```
goal: 0 + 1 = 1 + 0
```

`succ` branch

```
i : Nat
ih : P i
goal: P (i + 1)
```

```
ih : i + 1 = 1 + i
goal: (i + 1) + 1 = 1 + (i + 1)
```

Base Case: Computation Closes the Goal

```
theorem add_one_eq_one_add (i : Nat) :
    i + 1 = 1 + i := by
  induction i with
  | zero ⇒
      rfl
```

Goal

`0 + 1 = 1 + 0`

Why `rfl` works

Both sides reduce by computation to the same natural number.

Successor Case: Use the Induction Hypothesis

```
  | succ i ih ⇒
      calc
        (i + 1) + 1 = (1 + i) + 1 := by rw [ih]
        _ = 1 + (i + 1) := by rw [Nat.add_assoc]
```

`ih`

`i + 1 = 1 + i`

Successor goal

`(i + 1) + 1 = 1 + (i + 1)`

The induction hypothesis proves the predecessor statement; the successor proof still needs rewriting and associativity.

What Can Go Wrong in Induction

No `ih` in base case

The `zero` branch has no predecessor.

`ih` is not the goal

It proves the predecessor statement, not the successor statement.

Wrong motive

A weak proposition family gives a weak induction hypothesis.

Later generalization

Strong and well-founded induction give larger predecessor sets.

Recursion from Constructors

`Nat` Recursion: Odd Part

```
namespace Sharkovskii

def oddPart : Nat → Nat
  | 0 ⇒ 0
  | n + 1 ⇒
      if (n + 1) % 2 = 0 then
        oddPart ((n + 1) / 2)
      else
        n + 1
termination_by k ⇒ k
decreasing_by
  exact Nat.div_lt_self (Nat.succ_pos n) (by decide)
```

Mathematical idea

For a positive period `n`, write `n = 2^e * o`, where `o` is odd.

`termination_by`

The measure is the input `k`. The recursive call must make this measure smaller.

`Nat` Recursion: Two-Adic Exponent

```
namespace Sharkovskii

def twoAdic : Nat → Nat
  | 0 ⇒ 0
  | n + 1 ⇒
      if (n + 1) % 2 = 0 then
        twoAdic ((n + 1) / 2) + 1
      else
        0
termination_by k ⇒ k
decreasing_by
  exact Nat.div_lt_self (Nat.succ_pos n) (by decide)
```

What it returns

`twoAdic n` counts how many factors of `2` have been stripped.

`decreasing_by`

`Nat.div_lt_self` proves that dividing a positive natural number by `2` gives a smaller input.

Evaluating the Odd Part

- `oddPart 12 = 3`, because $12 = 2^2 \cdot 3$.
- `twoAdic 12 = 2`, the exponent of `2`.
- `oddPart 40 = 5`, because $40 = 2^3 \cdot 5$.

> Parentheses matter: `oddPart ((n + 1) / 2)` recurses on the input after division.

```
#eval oddPart 0
-- 0
#eval oddPart 12
-- 3
#eval twoAdic 12
-- 2
#eval oddPart 40
-- 5
#eval twoAdic 40
-- 3

end Sharkovskii
```

AI Can Easily Produce Bad Definitions

```
namespace Sharkovskii

def stripTwosAux : Nat → Nat → Nat
  | 0, n ⇒ n
  | fuel + 1, n ⇒
      if n % 2 = 0 && n ≠ 0 then
        stripTwosAux fuel (n / 2)
      else
        n

def oddPart (n : Nat) : Nat :=
  stripTwosAux n n
```

Problem

The helper `fuel` forces structural termination but hides the mathematical operation.

Lesson

Generated Lean must be read for mathematical intent, not only for local typechecking.

Tree Recursion: Depth and Size

```
namespace MyTree

def depth : MyTree alpha → Nat
  | nil ⇒ 0
  | node _ l r ⇒ Nat.max (depth l) (depth r) + 1

def size : MyTree alpha → Nat
  | nil ⇒ 0
  | node _ l r ⇒ 1 + size l + size r
```

`depth`

Combines two recursive results with maximum.

`size`

Combines two recursive results with addition.

Tree Recursion: Mirror

```
namespace MyTree

def mirror : MyTree alpha → MyTree alpha
  | nil ⇒ nil
  | node a l r ⇒ node a (mirror r) (mirror l)

#eval depth t3
-- 3
```

`mirror`

Recurses on both children and swaps them.

Constructor shape

The `node` branch stores a value, a left subtree, and a right subtree.

Tree Structural Induction

```
namespace MyTree

theorem size_mirror (t : MyTree alpha) :
    size (mirror t) = size t := by
  induction t with
  | nil ⇒
      rfl
  | node a l r ihl ihr ⇒
      simp [mirror, size, ihl, ihr]
      omega

end MyTree
```

Structural induction comes from the constructors. Later, well-founded induction is stronger: it can use any relation with no infinite descending chain.

Termination

Recursive definitions become part of Lean's logic.

Structural Recursion Is the Easy Case

Lean accepts recursive definitions when recursive calls are visibly made on structurally smaller data.

The recursive call is on `n`, the smaller part of `n + 1`.

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

#eval factorial 5
-- 120
```

Nonterminating Recursion Is Rejected

Expected failure

No recursive call moves to a smaller argument.

Logical reason

If arbitrary nonterminating definitions were accepted, the logic would become unsound.

```
-- Expected failure:
-- def bad : Nat → Nat
--   | n ⇒ bad n
```

Well-Founded Induction

General recursion needs a decreasing relation.

Accessibility Means Every Descent Is Handled

```
#check Acc
-- Acc (r : alpha → alpha → Prop) : alpha → Prop

#check Acc.intro
-- constructor for accessibility

#check WellFounded
-- WellFounded (r : alpha → alpha → Prop) : Prop
```

Relation orientation

`r y x` means that `y` is a predecessor of `x`.

Well-founded

Every element is accessible: there is no infinite descending chain.

Well-Founded Induction

To prove P x for every x: fix x assume P y for every predecessor y of x prove P x

(forall x, (forall y, r y x -> P y) -> P x) -> forall x, P x

```
example {alpha : Type} {r : alpha → alpha → Prop}
    (wf : WellFounded r)
    (P : alpha → Prop)
    (step : forall x, (forall y, r y x → P y) → P x)
    (x : alpha) : P x :=
  wf.induction x step
```

Mathlib `Tree`: Induction by Node Count

```
#check Tree
-- Tree (α : Type u) : Type u

#check Tree.node
-- Tree.node : α → Tree α → Tree α → Tree α

#check Tree.numNodes
-- Tree.numNodes : Tree α → Nat

example (P : Tree α → Prop)
    (step : ∀ t : Tree α,
      (∀ s : Tree α,
        Tree.numNodes s < Tree.numNodes t → P s) → P t) :
    ∀ t : Tree α, P t := by
  intro t
  exact (InvImage.wf Tree.numNodes wellFounded_lt).induction t step
```

Measure

`Tree.numNodes : Tree α -> Nat` sends a tree to a natural number.

`InvImage.wf`

Pulls the well-founded relation `<` on `Nat` back along `Tree.numNodes`.

Stronger hypothesis

The step may use every smaller-node tree, not only the immediate children.

Euclid's Algorithm Decreases in the Second Argument

```
def euclid : Nat → Nat → Nat
  | a, 0 ⇒ a
  | a, b + 1 ⇒ euclid (b + 1) (a % (b + 1))
termination_by _ b ⇒ b
decreasing_by
  exact Nat.mod_lt a (Nat.succ_pos b)

#eval euclid 30 12
-- 6
```

`termination_by`

Tells Lean to watch the second argument.

`decreasing_by`

Proves the watched argument decreases.

Important

This proof is about termination, not correctness of gcd.

Strong Induction Is the Natural-Number Case

```
#check Nat.strong_induction_on
-- (n : Nat) →
--   (forall n, (forall m < n, p m) → p n) → p n

example (P : Nat → Prop)
    (step : forall n, (forall m, m < n → P m) → P n) :
    forall n, P n := by
  intro n
  exact Nat.strong_induction_on n step
```

- Ordinary induction gives the immediately previous case.
- Strong induction gives all smaller natural numbers.
- Well-founded induction generalizes this to any no-infinite-descent relation.

Notation and Sharkovskii Periods

Overloaded notation is an interface question.

Dynamics: Iteration

```
namespace Dynamics

def iterate {X : Type} (f : X → X) : Nat → X → X
  | 0, x ⇒ x
  | n + 1, x ⇒ f (iterate f n x)
```

Discrete dynamical system

A state space `X` with a self-map `f : X -> X`.

Orbit

`x, f x, f (f x), ...` is represented by `iterate f n x`.

Dynamics: Exact Period

```
namespace Dynamics

def ReturnsAfter {X : Type} (f : X → X)
    (n : Nat) (x : X) : Prop :=
  iterate f n x = x

def IsPeriodicPoint {X : Type} (f : X → X)
    (n : Nat) (x : X) : Prop :=
  0 < n ∧ ReturnsAfter f n x

def HasExactPeriod {X : Type} (f : X → X)
    (n : Nat) (x : X) : Prop :=
  IsPeriodicPoint f n x ∧
    ∀ m : Nat, 0 < m → m < n → ¬ ReturnsAfter f m x

def HasPointOfExactPeriod {X : Type} (f : X → X)
    (n : Nat) : Prop :=
  ∃ x : X, HasExactPeriod f n x

end Dynamics
```

Return after `n` steps

`ReturnsAfter f 0 x` is always true.

Exact period

Positive return after `n` steps and no earlier positive return.

Point of exact period exists

`HasPointOfExactPeriod f n` means some point has exact period `n`.

Sharkovskii's Order Is Not Numerical Order

3, 5, 7, 9, ...

2*3, 2*5, 2*7, ...

2^2*3, 2^2*5, 2^2*7, ...

...

..., 2^3, 2^2, 2, 1, 0

Mathematical role

The order describes forcing between possible exact periods of interval maps.

Lean role

The theorem is hard; the period relation is a good object for modeling notation and order interfaces.

References in the notes: Sharkovskii (1964); Li and Yorke, "Period Three Implies Chaos" (1975).

Sharkovskii's Theorem

Theorem

Let I be an interval in the real line, let $f : I \to I$ be continuous, and let m,n be positive integers.

If there exists a point x of exact period m for f, and m comes before n in the Sharkovskii order, then there exists a point of exact period n for f.

Mathematical content

The order records which periods force which other periods for continuous interval maps.

Lean content

We model the period order, not the theorem about real intervals and continuous maps.

Li-Yorke: Period Three Implies Chaos

Period-forcing part

For a continuous interval map, the existence of a point of exact period 3 implies that for every positive integer k, there exists a point of exact period k.

Why Sharkovskii explains this

The number 3 is the first element in Sharkovskii's order.

What we are not formalizing here

Li-Yorke also proves a stronger chaotic-behavior statement. This lecture uses only the period-forcing part.

Li and Yorke, "Period Three Implies Chaos" (1975).

A Decidable Sharkovskii Relation

```
namespace Sharkovskii

abbrev sharkLE (m n : Nat) : Prop :=
  n = 0 ∨
    (1 < oddPart m ∧ oddPart n = 1) ∨
    (1 < oddPart m ∧ 1 < oddPart n ∧
      (twoAdic m < twoAdic n ∨
        (twoAdic m = twoAdic n ∧ oddPart m ≤ oddPart n))) ∨
    (oddPart m = 1 ∧ oddPart n = 1 ∧ n ≤ m)
```

Proposition, not Bool

`sharkLE m n` is a proposition built from decidable arithmetic statements.

Non-powers of two

Compare `twoAdic` first, then `oddPart` only when the exponents match.

`0`

`n = 0` makes `0` largest; powers of two use ordinary `n ≤ m`.

Computed Examples Through `decide`

```
infix:50 " ⩽ₛ " ⇒ sharkLE

#check sharkLE
-- Sharkovskii.sharkLE (m n : Nat) : Prop

#check (3 ⩽ₛ 5)
-- 3 ⩽ₛ 5 : Prop

#eval decide (3 ⩽ₛ 5)
-- true
#eval decide (5 ⩽ₛ 6)
-- true
#eval decide (6 ⩽ₛ 5)
-- false
#eval decide (8 ⩽ₛ 4)
-- true
#eval decide (4 ⩽ₛ 8)
-- false
#eval decide (1 ⩽ₛ 0)
-- true
#eval decide (0 ⩽ₛ 1)
-- false
```

Natural decidability

No custom `Decidable` instance is needed; Lean can decide the arithmetic proposition.

`1 ⩽ₛ 0`

This is true because `0` is the largest element in the augmented order.

Formal Theorem Statement Using the Order

```
def SharkovskiiTheoremStatement
    {I : Type}
    (IsContinuousIntervalMap : (I → I) → Prop) : Prop :=
  ∀ f : I → I, IsContinuousIntervalMap f →
    ∀ m n : Nat, 0 < n → m ≼ₛ n →
      Dynamics.HasPointOfExactPeriod f m →
      Dynamics.HasPointOfExactPeriod f n

#check SharkovskiiTheoremStatement
-- Sharkovskii.SharkovskiiTheoremStatement
--   {I : Type} →
--   ((I → I) → Prop) → Prop

end Sharkovskii
```

Formal shape

Exact period `m` plus `m ≼ₛ n` forces exact period `n`.

Positive target period

`0 < m` is already inside `HasPointOfExactPeriod f m`; keep `0 < n` to exclude the trivial zero-step return.

Inheritance Through Order Structures

Stronger interfaces extend weaker ones.

`PartialOrder` Adds Antisymmetry

```
#check PartialOrder
-- PartialOrder (alpha : Type u) : Type u

#check le_antisymm
-- le_antisymm [PartialOrder alpha] :
--   a ≤ b → b ≤ a → a = b
```

A partial order is a preorder satisfying antisymmetry.

If `a ≤ b` and `b ≤ a`, then `a = b`.

Graph Reachability Separates the Concepts

Reachability graph

Reflexive-transitive closure gives a preorder.

DAG reachability

No cycles, so mutual reachability forces equality.

```
preorder: a reaches b b reaches c therefore a reaches c

partial order: preorder plus no two distinct elements reach each other both ways
```

Lean’s Built-In Order Hierarchy

```
#check Preorder
-- Preorder (alpha : Type u) : Type u

#check PartialOrder
-- PartialOrder (alpha : Type u) : Type u

#check Nat.instPreorder
-- Nat.instPreorder : Preorder Nat

#check Nat.instPartialOrder
-- Nat.instPartialOrder : PartialOrder Nat

#check LinearOrder
-- LinearOrder (alpha : Type u) : Type u
```

`Preorder` fields

`Preorder` already includes `LE`, `LT`, reflexivity, transitivity, and compatibility between `<` and `≤`.

Ask for the Weakest Interface

```
example [PartialOrder alpha] {a b : alpha}
    (hab : a ≤ b) (hba : b ≤ a) : a = b :=
  le_antisymm hab hba
```

Only reflexivity or transitivity?

Ask for `[Preorder alpha]`.

Need antisymmetry?

Ask for `[PartialOrder alpha]`.

The Pattern to Remember

Inductive type

Constructors determine values, recursive functions, and induction principles.

Structure

Fields package data and proofs into one object.

Class

An interface dictionary that Lean can find by instance search.

Termination

Recursive definitions must decrease structurally or by a well-founded relation.

Source adapted from `/Users/hoxide/mydoc/ai4mathnus/nus-dsa/session03-notes.md`.