An Introduction to Lean

Functional Programming and Interactive Theorem Proving

Weinan Wang

Department of Mathematics, University of Oklahoma

ww@ou.edu

April 29, 2025

Preface

These notes are an introduction to Lean, a functional programming language and interactive theorem prover. They are aimed at undergraduates who have written a few programs before and have the mathematical maturity of a first course in discrete mathematics or proof. No prior exposure to type theory, formal logic, or functional programming is assumed.

The examples target Lean 4 and its standard library, with occasional use of Mathlib, the community mathematics library. Lean code is set in a monospaced font. Lean uses many Unicode symbols—$\to$, $\forall$, λ, $\mathbb{N}$, $\vdash$, and more—each entered in the editor by a short backslash abbreviation and displayed as one types; the correspondence between abbreviation and symbol is part of the editor extension.

The surest way through this material is with Lean open. Every example is meant to be typed and run, its computed values, inferred types, and proof goals inspected in the editor's information panel. Reading alone conveys the shape of the subject; typing conveys the feel of it, which is where the understanding lives.

Two conventions run throughout. Displayed program text appears in framed, line-numbered blocks. The state of a proof in progress—the hypotheses and the goal—appears in a lightly shaded panel that mirrors what the editor shows.

The material is arranged in nine chapters. The first three concern Lean as a programming language: meeting the tools and the read–check loop; the type system that classifies every expression; and functions, definitions, and the recursion that computes with them. The next four turn to Lean as a logic: inductive types, the sole source of data; the correspondence, due to Curry and Howard, that reads a proposition as a type and a proof as a program; the tactics that build proofs interactively; and induction, the principle shared between recursive definitions and inductive proofs. The eighth chapter treats structures and type classes, the mechanism of reuse that organises both programs and the mathematical library. The ninth is a sustained case study, building and verifying several sorting algorithms and a small compiler, in which every earlier idea is put to work at once.

Each chapter closes with exercises, ordered roughly by difficulty and marked with a star when they demand more. They are meant to be attempted in the editor, where a wrong

answer announces itself as a type error or an unproved goal rather than sitting silently on the page. The starred exercises reach a little beyond the text and reward the reader willing to consult the standard library or to experiment; the unstarred ones consolidate what a chapter has presented, and working them is the difference between recognising the material and commanding it.

A handful of examples recur across chapters, deliberately. The natural numbers, lists, and binary trees appear first as data, then as objects of proof; the arithmetic expressions introduced as an inductive type are later evaluated, measured, and finally compiled. Following these threads, watching one construction serve computation in one chapter and proof in the next, is the surest way to see that programming and proving are, in this language, a single activity.

Contents

Chapter 1

Meeting Lean

Lean is a programming language and a proof assistant built on one foundation: a form of dependent type theory in which every expression has a type and every proof is itself an expression. The tool that runs a program can also check a proof, because to Lean a proof and a program are the same kind of object.

1.1 Two tools in one

Seen as a programming language, Lean is pure, statically typed, and compiled. A program is an expression to be evaluated; functions have no side effects, and the type of every expression is known before it runs. Seen as a proof assistant, Lean is a system in which mathematical statements are written precisely and their proofs are checked to the last step by the machine, with no gaps taken on trust. The two views are the same system because of a single correspondence: a proposition is a type, and a proof of it is a term of that type. Constructing a proof and writing a program of a given type are the same activity.

The payoff of the single foundation is trust. A proof accepted by Lean has been reduced to the primitive rules of its type theory and rechecked by a small kernel, so "Lean accepts it" means the argument is correct in a way no informal proof can guarantee. The same mechanism that guarantees a program has the type it claims guarantees a theorem has a genuine proof.

1.2 Setting up

A working Lean setup has three parts, shown in Figure 1.2. The version manager `elan` installs and selects Lean toolchains, so different projects can use different versions without

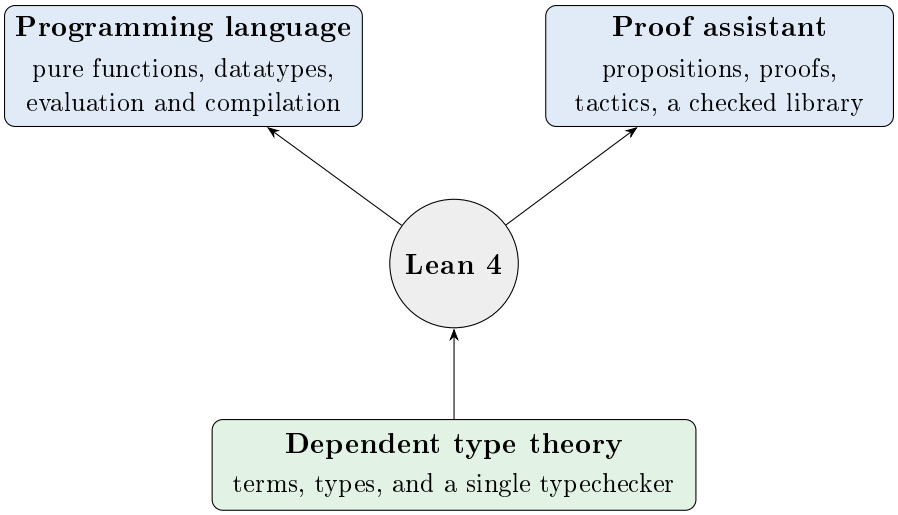


Figure 1.1: Lean presents two faces—a programming language and a proof assistant—resting on one foundation. Because proofs and programs are the same kind of object, a single typechecker serves both.

conflict. The build tool `lake` manages a project: its dependencies, its library structure, and the command that compiles it. The editor—most commonly Visual Studio Code with the Lean extension—is where code is written; behind it runs the *language server*, a Lean process that checks the file continuously and reports back.

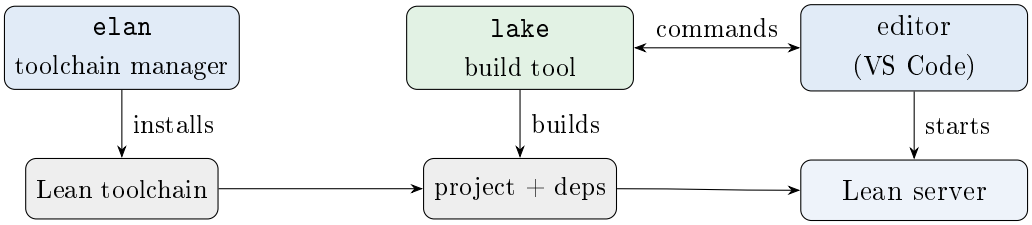


Figure 1.2: The three parts of a Lean setup. `elan` provides the toolchain, `lake` builds the project against it, and the editor runs a Lean server that checks the open file.

Installation follows the instructions on the Lean website: install `elan`, install the editor extension, and let it fetch a toolchain. A new project is created from the command line with `lake new my_project`, which lays out a folder with a configuration file and a first source file ending in `.lean`. Opening that folder in the editor starts the server, and the setup is ready.

1.3 The read–check loop

Lean is *interactive*: rather than compiling a whole program and reading output afterward, one edits a file and the server rechecks it after every change, placing its findings—inferred types, computed values, unproved goals, and errors—in a side panel called the *Infoview*. Placing the cursor at a point in the file shows the state of the proof or program exactly there. The working rhythm is the short loop of Figure 1.3: type a little, read the response, adjust.

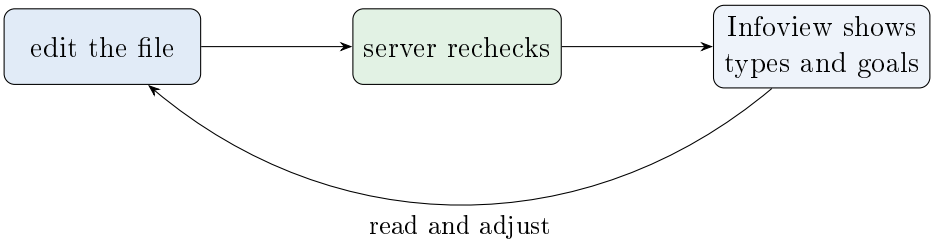


Figure 1.3: The interactive loop. Each edit triggers a recheck, whose results appear in the Infoview; reading them guides the next edit.

Three commands drive the earliest experiments, each beginning with `#` because it is a request to the system rather than part of a program. The command `#eval` evaluates an expression and prints its value; `#check` reports the type of an expression without evaluating it; and `#print` shows how a name is defined. A line beginning with one of these produces a message in the Infoview at that line.

A schematic Infoview, after checking a single line, looks like the panel below: the expression, its type, and any message sit together where the cursor rests.

```
▶ Messages (line 1)
  2 + 3 : ℕ
```

1.4 Evaluating expressions

The command `#eval` turns Lean into a calculator for pure expressions. Arithmetic on the natural numbers, strings, and lists all evaluate to a printed value.

```
#eval 2 + 3                 -- 5
#eval (2 + 3) * 4           -- 20
#eval 100 / 7               -- 14   (natural-number division rounds down)
#eval 100 % 7               -- 2    (remainder)
```

```
#eval "Hello, " ++ "Lean" -- "Hello, Lean"
#eval "formalization".length   -- 13
#eval [3, 1, 2].reverse   -- [2, 1, 3]
#eval [1, 2, 3, 4].map (fun x => x * x)   -- [1, 4, 9, 16]
```

Two features already stand out. Division on $\mathbb{N}$ returns a natural number, discarding any remainder, because the result must again be of type $\mathbb{N}$; genuine fractions require a different type. And a function such as `map` is an ordinary value that takes another function as an argument—here the squaring function `fun x => x * x`—which is characteristic of a functional language.

1.5 Inspecting types

Every expression in Lean has a type, and `#check` reports it without running anything. The response is written `e : T`, read "`e` has type `T`".

```
#check 2 + 3        -- 2 + 3 : ℕ
#check "Hello"      -- "Hello" : String
#check true         -- true : Bool
#check [1, 2, 3]    -- [1, 2, 3] : List ℕ
#check (2 : Int)    -- 2 : ℤ
#check Nat          -- ℕ : Type
#check String       -- String : Type
```

The last two lines are worth pausing over: a *type* is itself an expression, and its type is `Type`. Types thus have types of their own, and `Type` is where the types of ordinary data live. The line `(2 : Int)` shows a *type ascription*: the annotation `: Int` asks Lean to read the literal `2` as an integer rather than a natural number, and the reported type changes accordingly.

1.6 Definitions

A definition names an expression with the keyword `def`. The simplest form names a value; adding parameters names a function.

```
def greeting : String := "Hello, Lean!"
#eval greeting            -- "Hello, Lean!"

```

```
def double (n : ℕ) : ℕ := 2 * n
#eval double 21           -- 42

def area (width height : ℕ) : ℕ := width * height
#eval area 3 4            -- 12
```

The general shape is `def name (parameters) : Type := body`. The parameters are named with their types, the type after the colon is the type of the result, and the body is the expression the name stands for. A function is applied by writing it next to its arguments, with spaces and no parentheses: `double 21`, `area 3 4`. Parentheses group subexpressions, not argument lists.

A function need not be named. The expression `fun x => ...` is an *anonymous function*, and it is exactly what a named function abbreviates.

```
#check fun (x : ℕ) => x + 1     -- fun x => x + 1 : ℕ → ℕ
#eval (fun x => x * x) 5        -- 25
```

The reported type $\mathbb{N} \to \mathbb{N}$ is a *function type*, read "natural numbers to natural numbers". The arrow $\to$ is Lean's notation for a function type.

Example 1.1. A short definition and its evaluation form a complete, checkable experiment. For the polynomial $p(x) = x^2 - 3x + 2$,

```
def p (x : Int) : Int := x * x - 3 * x + 2
#eval p 5     -- 12
#eval p 1     -- 0
#eval p 2     -- 0
```

The results $p(1) = p(2) = 0$ display the roots of p, and every value has been computed by Lean rather than asserted.

1.7 Comments and the shape of a file

Text between `/-` and `-/` is a block comment, and text after `--` to the end of a line is a line comment; Lean ignores both. A source file is a sequence of definitions and commands, checked from top to bottom, so a name must be defined before it is used.

```
/- A first Lean file.
   Comments like this are ignored by the checker. -/
```

```
3
4 def factorialInput : ℕ := 5   -- an ordinary definition
5
6 #eval factorialInput          -- 5   (a command, reporting a value)
7 #check factorialInput         -- factorialInput : ℕ
```

With evaluation, type inspection, and definitions in hand, the raw materials of both programming and proving are in place. Types, which `#check` has already brought to the surface, are the central organizing idea of the language.

1.8 Typing the notation

Lean's source is full of symbols that no keyboard carries directly: the arrow $\to$, the quantifiers $\forall$ and $\exists$, the Greek letters that name types. The editor accepts each of these through a short backslash abbreviation, expanded as soon as it is recognised. Typing `\to` and then a space produces $\to$; typing `\alpha` produces α. The abbreviations follow the names one would say aloud, so they are quickly learned rather than memorised.

Symbol	Abbreviation	Meaning
$\to$	`\to` or `\r`	function arrow, implication
$\forall$	`\forall` or `\all`	universal quantifier
$\exists$	`\exists` or `\ex`	existential quantifier
$\wedge$	`\and`	conjunction
$\vee$	`\or`	disjunction
$\neg$	`\not` or `\neg`	negation
$\leq$	`\le`	less than or equal
$\times$	`\times` or `\x`	product type
$\mathbb{N}$	`\N` or `\nat`	the naturals
α	`\alpha` or `\a`	a type variable
λ	`\lambda` or `\fun`	anonymous function
$\vdash$	`\vdash` or `\\|-`	the turnstile in goals

Table 1.1: Common symbols and the backslash sequences that produce them. The editor expands each sequence on the following keystroke; hovering over a symbol reports the sequence that types it.

The mechanism runs both ways. When a symbol is already on screen, hovering over it shows the abbreviation that would reproduce it, so an unfamiliar character encountered while reading can always be identified. This removes the single most common obstacle a newcomer meets: the sense that the notation is unreachable from an ordinary keyboard.

1.9 The command family

Four commands beginning with a hash mark inspect a Lean file without being part of any program. Each reports something about an expression and prints its answer in the Infoview. Knowing which one to reach for is part of reading and writing Lean fluently.

Command	Question it answers	Example output
`#check`	what is the type of this?	`5 : Nat`
`#eval`	what does this compute to?	`120`
`#reduce`	what is its normal form?	`Nat.succ Nat.zero`
`#print`	how is this name defined?	the definition body

Table 1.2: The inspection commands. `#check` reports a type, `#eval` runs a computation, `#reduce` unfolds to a normal form, and `#print` recovers a definition.

The distinction between `#eval` and `#reduce` is worth a moment. Both compute, but `#eval` uses an efficient compiled evaluator and prints the result the way a program would, while `#reduce` performs reduction inside the kernel and exposes the raw term, successors and all. For a numeral the compiled answer is far more readable; for seeing how a term is actually built, the raw form is more honest.

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

#eval   fact 5        -- 120
#reduce fact 3        -- shows Nat.succ (Nat.succ ( ... ))  for 6
#check  fact          -- fact : Nat → Nat
#print  fact          -- prints the definition above
```

The command `#print` is a window into the library. Applied to a name from the standard library it recovers the definition, letting one read how a familiar operation is actually implemented rather than guessing. Applied to a theorem it shows the proof term, which is often smaller and stranger than the tactic proof that produced it.

1.10 Reading an error

A file that does not compile is the normal state of a file being written, and the error report is the tool for fixing it, not a rebuke. Lean's errors name a location, state what was expected, and state what was found. Consider applying the successor function to a boolean.

```
#check Nat.succ true
```

```
error: application type mismatch
  Nat.succ true
argument
  true
has type
  Bool : Type
but is expected to have type
  Nat : Type
```

The message is a small essay with a fixed structure. It names the offending application, singles out the argument `true`, reports its type `Bool`, and states the type `Nat` that the position required. The fix is read directly off the report: supply a `Nat` where a `Nat` is wanted. Almost every type error follows this template, and learning to read the four lines turns a wall of red into a precise instruction.

Remark 1.2. Two phrases recur and are worth recognising. "Expected type" names what the surrounding context demands; "has type" or "inferred type" names what the offending expression actually is. A mismatch between the two is the commonest error, and the two named types are exactly the information needed to resolve it.

1.11 A first session

A short worked session shows the commands cooperating. The task is to define a function, inspect its type, run it, and confirm a small fact about it, in the order one would actually type.

```
-- 1. define a function
def triple (n : Nat) : Nat := 3 * n

-- 2. confirm its type
#check triple            -- triple : Nat → Nat

-- 3. run it on an input
#eval triple 7           -- 21

-- 4. state and check a fact about it
example : triple 0 = 0 := rfl
```

```
example : triple 4 = 12 := rfl
```

The keyword `example` states a proposition and proves it without attaching a name, which is exactly what is wanted for a throwaway check. Both facts hold by `rfl` because each side reduces to the same numeral. A session like this, define, inspect, run, verify, is the rhythm of working in Lean, and every later chapter is an elaboration of these four moves.

Example 1.3 (When `rfl` is not enough)**.** Not every true equation is closed by `rfl`. The claim that tripling distributes over addition is true, but its two sides do not reduce to a common numeral, since they contain variables. Attempting `rfl` fails, and the fact instead needs the reasoning developed later.

```
-- example : triple (a + b) = triple a + triple b := rfl   -- fails
example (a b : Nat) : triple (a + b) = triple a + triple b := by
  simp [triple, Nat.mul_add]
```

The failure is informative: `rfl` settles equations that hold by computation, and a statement quantified over all `a` and `b` holds by a law, not a computation. Recognising which kind of equation is at hand is the first diagnostic skill a proof requires.

1.12 How a project is organised

A single file suffices for experiment, but larger work is arranged as a *package* managed by the build tool `lake`. A package has a configuration file naming it and its dependencies, a root source directory, and any number of modules that import one another. The command `lake build` compiles the package, checking every proof it contains.

```
-- lakefile.lean  (schematic)
import Lake
open Lake DSL

package my_project

lean_lib MyProject

@[default_target]
lean_exe my_project where
  root := `Main
```

Dependencies, including Mathlib, are listed in the same file and fetched by `lake`. Within

the source, one module makes another's definitions available with `import`, exactly as the standard library is brought in. The mechanics matter little for learning the language, but knowing that a proof lives inside a package, checked in full by a single build command, explains how large formal developments are kept honest as they grow.

1.13 A tour of the built-in types

Before writing programs of any size it helps to know the basic types the language provides and the literals that produce their values. Five carry most everyday data: the naturals, the integers, floating-point numbers, booleans, and text. Each has its own literal syntax, and `#check` reports which type a given literal receives.

```
#check (42 : Nat)      -- unbounded non-negative whole numbers
#check (-42 : Int)     -- signed whole numbers
#check (3.14 : Float)  -- floating-point numbers
#check true            -- Bool: true or false
#check 'a'             -- Char: a single character
#check "hello"         -- String: text
```

The naturals are unbounded: `Nat` has no largest value, and subtraction that would go below zero is truncated to zero rather than wrapping or erroring. The integers add negatives. Floating-point numbers follow the usual hardware conventions and are kept distinct from the exact naturals and integers, since their arithmetic is approximate.

Type	Sample literals	Typical operations
`Nat`	`0`, `7`, `100`	`+`, `*`, `-` (truncated), `/`, `%`
`Int`	`-3`, `0`, `15`	`+`, `*`, `-`, `/`, `%`
`Float`	`3.14`, `-0.5`	`+`, `*`, `/`, `Float.sqrt`
`Bool`	`true`, `false`	`&&`, `||`, `not`
`Char`	`'a'`, `'Z'`, `'9'`	`Char.toNat`, `Char.isDigit`
`String`	`"hi"`, `""`	`++`, `.length`, `.toList`

Table 1.3: The common built-in types with representative literals and operations. Naturals truncate subtraction at zero; floats follow hardware arithmetic and stay separate from the exact numeric types.

1.14 Strings and characters

Text is a first-class value. Strings concatenate with `++`, report their length, and convert to and from lists of characters, so the full apparatus of list processing is available to them. A character is a single code point, itself convertible to a natural for arithmetic or comparison.

```
#eval "Hello, " ++ "world"        -- "Hello, world"
#eval "abc".length                -- 3
#eval "abc".toList                -- ['a', 'b', 'c']
#eval 'A'.toNat                   -- 65
#eval s!"1 + 1 = {1 + 1}"         -- "1 + 1 = 2"
```

The last line uses *string interpolation*: the prefix `s!` marks a string in which any expression inside braces is evaluated and inserted. This is the readable way to assemble a message from computed parts, and it appears throughout in error reports and program output.

1.15 Booleans and conditions

The boolean operations are the connectives `&&` for conjunction, `||` for disjunction, and `not` for negation. Comparisons produce booleans, and a conditional chooses between two values on a boolean test. Because a conditional is an expression, its two branches must share a type, and the whole `if` has that type.

```
#eval true && false               -- false
#eval 3 < 5 && 5 < 10             -- true
#eval not (2 == 3)                -- true
#eval if 2 < 3 then "yes" else "no"   -- "yes"
```

A subtlety distinguishes the boolean `==` from the propositional `=`. The former computes a `Bool` and can be evaluated; the latter states a `Prop` and is proved. For the built-in types the two agree in meaning, and the boolean form is what appears inside a running computation, the propositional form inside a proof.

1.16 Namespaces

Related definitions are grouped in a *namespace*, which prefixes their names and keeps them from colliding with others. A namespace is opened when its short names are wanted directly,

and closed otherwise, so the same brief name may be reused in different namespaces without ambiguity.

```
namespace Geometry
  def area (w h : Nat) : Nat := w * h
  def perimeter (w h : Nat) : Nat := 2 * (w + h)
end Geometry

#eval Geometry.area 3 4           -- 12

open Geometry
#eval area 3 4                    -- 12,  short name after open
```

Inside the block the definitions are written with short names; outside, they are reached through the prefix `Geometry.` until the namespace is opened. The constructors of every inductive type live in a namespace named for the type, which is why they are written `Color.red` and `List.cons`, and why opening the namespace makes the bare names available.

1.17 A first program

An evaluated expression prints its value, but a *program* performs actions: it reads input, prints output, and returns nothing of interest. Such an action has type `IO Unit`, and the entry point of a compiled program is a definition named `main` of that type. The smallest one prints a line.

```
def main : IO Unit := IO.println "Hello, world!"

#eval main            -- prints: Hello, world!
```

The type `IO Unit` says that `main` performs input-output and yields the uninformative value `()`. Actions are sequenced in a `do` block, where each line runs in turn, and values read from the world are bound with an arrow.

```
def greet : IO Unit := do
  IO.println "What is your name?"
  let name ← (return "world")   -- a real program would read input
  IO.println s!"Hello, {name}!"
```

The `do` block orders the actions top to bottom, and the bound `name` is used in the

interpolated greeting. Programs and proofs sit in the same language: the proofs of later chapters and this printing program are both terms, differing only in whether their type is a proposition or an action.

Remark 1.4. Running a real compiled program uses `lake exe`, which builds the package and executes its `main`. Within the editor, `#eval main` runs the action immediately and shows its output, which is enough for experiment. The two routes execute the same definition; only the surrounding ceremony differs.

1.18 Sections and shared variables

When several definitions share the same parameters, declaring those parameters once is cleaner than repeating them. The `variable` command introduces names that any following definition may use, and a `section` bounds their scope, so that outside the section the names are gone. Only the variables a definition actually mentions become its parameters.

```
section Arithmetic
  variable (n m : Nat)

  def addThem : Nat := n + m
  def mulThem : Nat := n * m
end Arithmetic

#check addThem      -- addThem : Nat → Nat → Nat
#check mulThem      -- mulThem : Nat → Nat → Nat
```

Although `addThem` is written with no parameter list, it acquires `n` and `m` from the enclosing `variable` declaration, so its type is a two-argument function. The section keeps this convenience local: after `end Arithmetic` the names `n` and `m` are no longer in scope, and a later definition is unaffected. Grouping related definitions under shared variables is the ordinary way to keep a file free of repeated parameter lists.

Remark 1.5. A definition takes only the variables it uses, not every one in scope. A definition inside the section mentioning only `n` would have type `Nat → Nat`, not `Nat → Nat → Nat`. This keeps signatures honest: the parameters of a definition reflect what it depends on, whatever was declared around it.

1.19 Arrays

Lists are built from links and traversed from the front, which makes indexing into them slow. When random access matters, an `Array` is the better structure: its elements sit in a contiguous block, so reading the element at a position is immediate. Array literals are written with a hash before the brackets.

```
def a : Array Nat := #[10, 20, 30]

#eval a.size          -- 3
#eval a[0]!           -- 10,  immediate access by index
#eval a.push 40       -- #[10, 20, 30, 40]
#eval a.map (· * 2)   -- #[20, 40, 60]
```

The operations mirror those on lists, `map`, `push`, `size`, but the representation differs, and with it the cost. Indexing an array with `a[i]!` is a constant-time operation, whereas reaching the same position in a list requires walking past every earlier element. The two structures coexist: lists suit recursive definitions and proofs, arrays suit indexed computation, and a program uses whichever fits the task.

	List	**Array**
literal	`[1, 2, 3]`	`#[1, 2, 3]`
index access	slow (walk the links)	fast (direct)
add to front	fast (`::`)	slow (shift all)
add to end	slow (walk to end)	fast (`push`)
suits	recursion, proofs	indexed computation

Table 1.4: Lists versus arrays. Both hold a sequence of elements, but their representations make different operations cheap; the choice follows the access pattern a task requires.

1.20 Documentation comments

Three kinds of comment appear in Lean source. A line comment begins with two dashes and runs to the end of the line; a block comment is enclosed in `/-` and `-/` and may span many lines; and a *documentation* comment, written with `/-` and `-/` before a declaration, attaches text that the editor displays when the declaration is hovered.

```
-- an ordinary line comment

```

```
/- a block comment,
   possibly spanning
   several lines -/

/-- Doubles a natural number. This text appears on hover. -/
def double (n : Nat) : Nat := n + n
```

The documentation comment is more than decoration: it becomes part of the declaration's recorded interface, visible wherever `double` is used and collected into generated documentation for a library. Writing a short doc comment for each definition is the habit that keeps a growing body of code navigable, since a reader can learn what a function does without finding and reading its body.

Example 1.6 (A small documented file)**.** Assembling the pieces, a self-describing fragment pairs each definition with its purpose and a check confirming its behaviour.

```
/-- The square of a natural number. -/
def square (n : Nat) : Nat := n * n

/-- The sum of the squares of two naturals. -/
def sumOfSquares (a b : Nat) : Nat := square a + square b

example : sumOfSquares 3 4 = 25 := by decide
```

Each definition states its intent in a doc comment, and the `example` records a fact that would break if either definition were changed carelessly. A file written this way documents and tests itself, and both the documentation and the tests are checked every time it compiles.

1.21 Arithmetic in depth

Beyond the four operations, the numeric types support division, remainder, powers, and the comparisons and extrema, and knowing their exact behaviour avoids surprises. Division on the naturals and integers is integer division, discarding any fractional part, and its companion remainder recovers what division drops.

```
#eval 17 / 5        -- 3,  integer division truncates
#eval 17 % 5        -- 2,  the remainder
#eval 2 ^ 10        -- 1024
#eval min 3 8       -- 3
#eval max 3 8       -- 8
#eval Nat.gcd 12 18 -- 6
```

The result of `17 / 5` is `3`, not a fraction, and `17 % 5` is the leftover `2`, so that `5 * 3 + 2` recovers `17`. On the naturals, division by zero is defined to be zero rather than an error, and subtraction that would fall below zero is truncated, both choices made so that the operations are total functions returning a genuine natural in every case.

Operation	Meaning	Note
`a / b`	integer quotient	truncates; `a / 0 = 0`
`a % b`	remainder	`b * (a / b) + a % b = a`
`a ^ b`	power	`a` raised to `b`
`min a b`	smaller of two	
`max a b`	larger of two	
`a - b`	subtraction on `Nat`	truncated at `0`

Table 1.5: Numeric operations beyond the basic four. On the naturals, division and subtraction are made total by defining the awkward cases, division by zero and negative differences, to yield zero.

Remark 1.7. The totality of these operations is deliberate. A mathematical function returns a value for every input, so `a / 0` must denote something, and zero is the chosen convention. This differs from languages that raise an exception, and it means a proof about division must account for the zero case explicitly rather than assuming it cannot arise. The convention trades a run-time error for a proof obligation.

1.22 More with text

Text processing draws on a vocabulary paralleling that for lists. A string can be split on a separator, searched for a character, trimmed of surrounding space, and rejoined from pieces; individual characters can be classified and transformed. These operations make short work of the common tasks of reading and shaping input.

```
#eval "a,b,c".splitOn ","       -- ["a", "b", "c"]
#eval "hello".contains 'e'      -- true
#eval " hi ".trim               -- "hi"
#eval String.join ["a", "b"]    -- "ab"
#eval 'A'.isAlpha               -- true
#eval 'A'.toLower               -- 'a'
```

Splitting turns a string into a list of substrings, at which point the list operations of later chapters apply; joining reverses the process. Character classification, `isAlpha`, `isDigit`, and the case conversions, lets text be filtered or normalised element by element. The correspondence with lists is close because a string is, in effect, a sequence of characters with a specialised representation.

1.23 A small program

Combining arithmetic and text, a short program counts the words in a string by splitting on spaces and measuring the resulting list. It reads as a pipeline of the operations just introduced.

```
def wordCount (s : String) : Nat :=
  (s.splitOn " ").filter (· ≠ "") |>.length

def averageWordLength (s : String) : Nat :=
  let words := (s.splitOn " ").filter (· ≠ "")
  let total := words.foldl (fun acc w => acc + w.length) 0
  if words.isEmpty then 0 else total / words.length

#eval wordCount "the quick brown fox"          -- 4
#eval averageWordLength "the quick brown fox"  -- 4
```

The word count splits on spaces, discards any empty pieces from repeated spaces, and counts what remains; the average length sums the word lengths and divides. Each is a few lines built from splitting, filtering, folding, and the arithmetic of the previous section. A program of real use is assembled the same way, from the small operations the standard library provides, composed into the transformation one needs.

Example 1.8 (Guarding against the empty case)**.** The average deliberately checks for an empty list of words before dividing, returning zero rather than dividing by zero. Although the naturals define `n / 0` as zero, the explicit guard states the intent and would be essential were the division on a type without that convention.

```
#eval averageWordLength ""     -- 0,  no words, no division
```

Handling the boundary case explicitly is a habit worth keeping. A program that relies on a truncation convention to avoid an error is correct but obscure; one that names the empty case says what it means.

Exercises

Exercise 1.1. Predict the value of each expression, then check with `#eval`: `7 * 6`, `20 / 3`, `20 % 3`, `"ab" ++ "cd"`, and `[5, 4, 3].reverse`.

Exercise 1.2. Predict the type reported by `#check` for each of `3 + 4`, `"lean"`, `false`, `[true, false]`, and `(5 : Int)`.

Exercise 1.3. Define `square : ℕ → ℕ` returning n^2, and evaluate it at 0, 7, and 12.

Exercise 1.4. Define `poly : Int → Int` for $q(x) = x^2 - 5x + 6$ and evaluate it at 2 and 3. What do the results say about q?

Exercise 1.5. Write an anonymous function that adds 10 to its input and apply it to 32 in a single `#eval`.

Exercise 1.6. Define `repeatTwice : String → String` that concatenates its argument with itself, and evaluate it on `"ab"`.

Exercise 1.7. Explain, in terms of the type of the result, why `#eval 7 / 2` prints `3` rather than `3.5`.

Exercise 1.8. *Using `#check`, determine and explain the type Lean assigns to `fun (f : ℕ → ℕ) => f 0`. What does this type say about the argument `f`?

Exercise 1.9. Using the abbreviations of Table 1.1, type the symbols $\forall$, $\to$, $\times$, and $\mathbb{N}$ in the editor. Report the backslash sequence that produced each.

Exercise 1.10. Apply both `#eval` and `#reduce` to `fact 4`. Describe how the two outputs differ and explain why `#eval` is more readable for a numeral.

Exercise 1.11. Use `#print` on a definition of your own and on `Nat.add`. What does each report, and how does the library definition differ from what you expected?

Exercise 1.12. Deliberately apply a function to an argument of the wrong type and read the resulting error. Identify the "expected type" and the "has type" lines, and state the one-line fix.

Exercise 1.13. Carry out a first session for a squaring function: define it, check its type, evaluate it on `6`, and verify `square 0 = 0` and `square 5 = 25` with `example`.

Exercise 1.14. * Explain why `square (a + b) = square a + square b` is *false*, whereas `triple (a + b) = triple a + triple b` is true but not provable by `rfl`. What distinguishes the two situations?

Exercise 1.15. Report the type Lean assigns to each literal: `0`, `-5`, `2.5`, `'x'`, `"x"`, `true`. Which two are whole numbers, and how do they differ?

Exercise 1.16. Using string operations, concatenate two strings, take the length of the result, and build an interpolated message reporting that length with `s!`.

Exercise 1.17. Evaluate `(3 < 5) && (5 < 4)` and `not (2 == 2)`. Explain the difference between the boolean `==` and the propositional `=`.

Exercise 1.18. Define two functions inside a namespace `Stats`, then call each once through its full name and once after `open Stats`.

Exercise 1.19. Write a `main : IO Unit` that prints two separate lines, and run it with `#eval`. What value does `main` return?

Exercise 1.20. * Evaluate `3 - 5` at type `Nat` and explain the result. Then evaluate it at type `Int` and account for the difference.

Exercise 1.21. Using `variable` inside a `section`, share two natural parameters across two definitions, then check the type each definition receives outside the section.

Exercise 1.22. Build an `Array Nat`, read an element by index, `push` a new element, and `map` a function over it. Contrast each operation's cost with the same on a `List`.

Exercise 1.23. Attach a documentation comment to a function of your own, and describe where its text becomes visible.

Exercise 1.24. Explain why a definition placed inside a `variable` block takes only the variables it mentions, and give an example whose type has fewer arguments than the number of variables in scope.

Exercise 1.25. Write a small self-documenting file with two definitions, each carrying a doc comment, and one `example` testing them.

Exercise 1.26. * Give one task for which an `Array` is the better structure and one for which a `List` is, justifying each choice by the access pattern involved.

Exercise 1.27. Evaluate `23 / 4` and `23 % 4`, and verify by hand that `4 * (23 / 4) + 23 % 4` equals 23.

Exercise 1.28. Evaluate `min 12 7`, `max 12 7`, `Nat.gcd 24 36`, and `3 ^ 4`, predicting each before checking.

Exercise 1.29. Using string operations, split `"a,b,c,d"` on the comma, test whether it contains a given character, and rejoin the pieces with `String.join`.

Exercise 1.30. Write a program `vowelCount` that counts the vowels in a string, and test it on a short sentence.

Exercise 1.31. Explain why `7 / 0` evaluates to `0` on the naturals, and what obligation this convention creates in a proof about division.

Exercise 1.32. ⋆ Write a program that reverses the order of the words in a sentence, using `splitOn`, a list reversal, and `String.join`.

Chapter 2

Types and Terms

Lean is built on a single organizing idea: every expression carries a type, and the type of an expression is itself an expression with a type. A number, a function, a proof, and the collection of all natural numbers are all *terms*, and each one stands in a typing relationship with some other term. Understanding this relationship is the whole of the foundation; the rest of the language decorates it with convenient notation.

2.1 Everything has a type

Write a colon between a term and a type to assert that the term inhabits the type. Lean reads the assertion, checks it, and either accepts it or reports where the check failed.

```
#check 5            -- 5 : Nat
#check true         -- true : Bool
#check "hello"      -- "hello" : String
#check Nat          -- Nat : Type
#check (2 : Int)    -- 2 : Int
```

The first four lines report the type that Lean assigns automatically. The last line uses an explicit ascription: the annotation `(2 : Int)` forces the literal 2 to be read as an integer rather than a natural number. Ascription does not convert a value; it selects, among the possible readings of a piece of syntax, the one the author intends.

A literal like 2 is overloaded. On its own Lean gives it the type `Nat`, but the same three characters can denote an integer, a rational, or a real, depending on the surrounding demand. The colon lets the author supply that demand directly.

Definition 2.1 (Typing judgment)**.** The assertion that a term t has type T is written $t : T$

and is called a *typing judgment*. It is the atomic statement of the theory: to know that $t : T$ holds is to know that t is a legitimate inhabitant of T.

Judgments are not propositions to be proved with tactics. They are decided mechanically by the type checker as it reads the file. When the checker succeeds, the judgment holds; when it fails, the file does not compile and the offending expression is underlined.

2.2 The layered picture

Because the type of a term is again a term, the colon relation stacks into layers. The value 5 sits in `Nat`; the type `Nat` sits in `Type`; and `Type` itself sits one level higher.

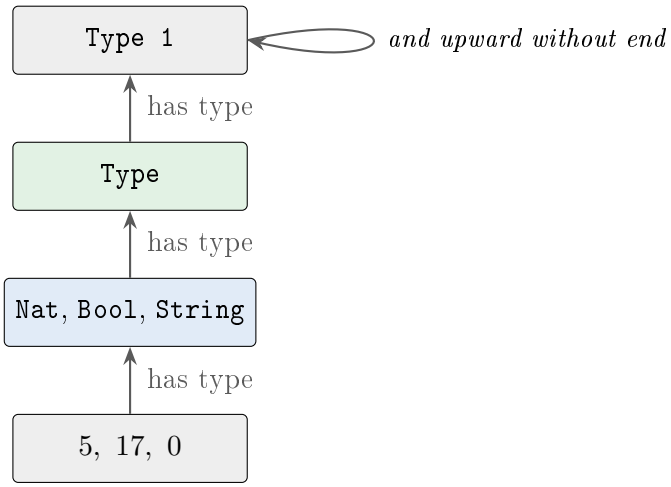


Figure 2.1: Terms and their types form an unbounded tower. Values sit in ordinary types, ordinary types sit in `Type`, and `Type` sits in a still larger universe.

The tower never terminates in a top element. If there were a single type of all types, containing itself, the system would admit a version of Russell's paradox and every proposition would become provable. The layered universes of Figure 2.1 are exactly the device that keeps the theory consistent.

2.3 Universes

The types that classify data live in a hierarchy of *universes*. The smallest is written `Type`, or equivalently `Type 0`. Above it sit `Type 1`, `Type 2`, and so on.

```
#check Nat          -- Nat  : Type
```

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

Alongside this tower sits a separate universe reserved for *propositions*, written `Prop`. A term of type `Prop` is a statement; an inhabitant of that statement is a proof of it. Lean treats `Prop` as the bottom of the same hierarchy under the umbrella name `Sort`: the identifications `Prop` $=$ `Sort 0` and `Type` $n =$ `Sort` $(n+1)$ hold definitionally.

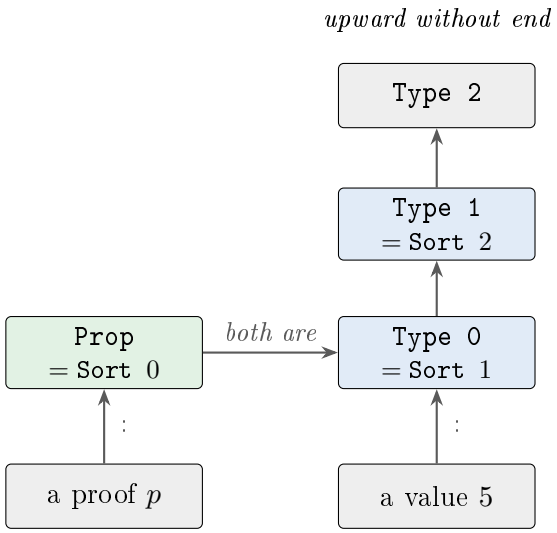


Figure 2.2: The universe `Prop` of propositions sits beside the tower of data universes. A proof inhabits a proposition, a value inhabits a data type, and both propositions and data types are themselves inhabitants of higher universes.

The separation of `Prop` from `Type` is not bureaucratic. It records a deliberate design choice: two proofs of the same proposition are treated as interchangeable, whereas two values of the same data type need not be. This *proof irrelevance* is what lets a proof be erased before a program runs, since the identity of a proof can never affect a computed answer.

2.4 Function types

Given types A and B, the type $A \to B$ classifies functions that accept an input of type A and produce an output of type B. The arrow associates to the right, so $A \to B \to C$ means $A \to (B \to C)$: a function of one argument that returns another function.

```
#check Nat.succ          -- Nat.succ : Nat → Nat
#check Nat.add           -- Nat.add  : Nat → Nat → Nat
```

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

The middle example is the crucial one. Addition on the naturals does not take a pair of numbers; it takes one number and returns a function awaiting the second. Supplying arguments one at a time is called *currying*, and it is the default in Lean.

Example 2.2 (Reading an arrow type)**.** The type `Nat → Nat → Bool` describes a test that compares two naturals. Applying it to a single argument is legitimate and yields a value of type `Nat → Bool`: a test with the first argument already fixed.

```
def geq (m n : Nat) : Bool := n ≤ m
#check geq             -- geq   : Nat → Nat → Bool
#check geq 10          -- geq 10 : Nat → Bool
#eval  geq 10 3        -- true
#eval  geq 10 25       -- false
```

The partial application `geq 10` is a genuine value that can be named, passed to another function, or stored in a list.

2.5 The typing judgment as a rule

The type checker does not guess. Each way of forming an expression comes with a rule stating which judgment about the whole follows from judgments about the parts. The rule for application is representative: if f has a function type and a has the matching input type, then the application $f\,a$ has the output type.

$$\frac{f : A \to B \qquad a : A}{f\,a : B}$$

Reading such a figure downward is exactly what the checker does. To accept `Nat.succ 4` it confirms `Nat.succ : Nat → Nat` and `4 : Nat`, matches the input type `Nat` against the argument, and concludes `Nat.succ 4 : Nat`. A mismatch at the middle step is precisely a type error.

```
#check Nat.succ 4          -- Nat.succ 4 : Nat        ✓ accepted
-- #check Nat.succ true    -- error: expected Nat, got Bool
```

The commented line, if uncommented, halts compilation. Lean reports that the argument `true` has type `Bool` where a `Nat` was required, and points at the offending word.

2.6 Inference and checking

Two questions face the elaborator whenever it meets an expression. Given a term, what is its type? And given a term together with an expected type, does it fit? The first is *type inference*; the second is *type checking*. Lean performs both, and the interplay is what makes the surface syntax so light.

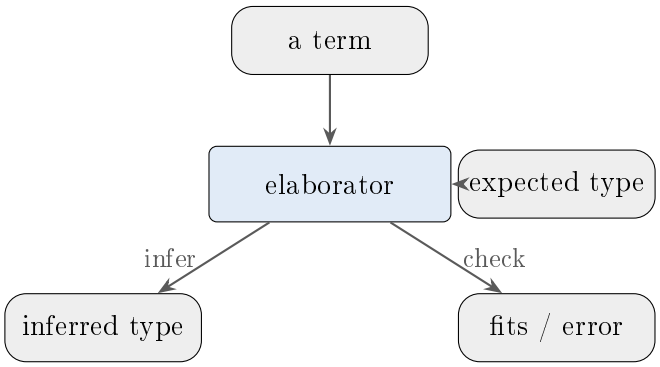


Figure 2.3: The elaborator runs in two directions. With only a term it infers a type; with a term and an expected type it checks compatibility, using the expectation to resolve overloaded literals and implicit arguments.

Ascription feeds the checking direction. When `(2 : Int)` is elaborated, the expected type `Int` flows inward and selects the integer reading of the literal. Without the annotation the elaborator infers `Nat` instead. The same mechanism resolves the type of an empty list, which is ambiguous until a demand arrives from outside.

```
#check ([] : List Nat)      -- [] : List Nat
#check ([] : List Bool)     -- [] : List Bool
```

2.7 Implicit arguments

Some arguments are determined by others and would be tedious to write. The identity function is the standard illustration: once its input is supplied, the type of that input is visible, so stating the type separately is redundant.

```
def id' (α : Type) (x : α) : α := x
#eval id' Nat 7            -- 7,  but the Nat is noise

def idImp {α : Type} (x : α) : α := x
```

```
#eval idImp 7              -- 7,  Lean infers α = Nat
#eval idImp "hi"           -- "hi", Lean infers α = String
```

Braces mark an argument as *implicit*. Lean fills it in from the types of the explicit arguments, and the caller writes only what carries information. When the implicit value cannot be inferred, an at-sign switches every argument back to explicit form so it can be supplied by hand.

```
#check @idImp              -- @idImp : (α : Type) → α → α
#eval  @idImp Nat 7         -- 7
```

Remark 2.3. The pattern generalizes: a great deal of Lean's convenience comes from information that is present in one argument being reused to fill another. The type checker never invents a value; it only propagates one that the surrounding expression already determines.

2.8 Dependent function types

The output type of a function may itself mention the input value, not merely the input type. Such a function has a *dependent* type, written with a named binder in place of the plain arrow.

```
#check @List.replicate
-- @List.replicate : (n : Nat) → α → List α
```

Here the result type `List` α does not depend on the value n, but the general shape $(n : A) \to B\,n$ permits it to. A function whose return type genuinely varies with the argument is what distinguishes a dependent type theory from an ordinary typed language, and it is the feature that will later let a single arrow encode a universally quantified statement.

Definition 2.4 (Dependent function type)**.** Given a type A and, for each $x : A$, a type $B\,x$, the dependent function type $(x : A) \to B\,x$ classifies functions f such that $f\,a : B\,a$ for every $a : A$. When B does not depend on x this collapses to the ordinary arrow $A \to B$.

The notation $(x : A) \to B\,x$ is Lean's spelling of the product $\prod_{x:A} B\,x$ familiar from type theory. Ordinary functions are the special case where the family B is constant, so nothing is lost by using one notation for both.

2.9 Pairs and products

Two values may be carried together as a *pair*, whose type is the product of the two component types. The product of A and B is written $A \times B$, and a pair is written with a comma inside parentheses. The components are recovered by the projections `.1` and `.2`.

```
def pt : Nat × Nat := (3, 4)

#check pt          -- pt : Nat × Nat
#eval  pt.1        -- 3
#eval  pt.2        -- 4
#check (1, "a", true)   -- Nat × String × Bool
```

The product associates to the right, so $A \times B \times C$ means $A \times (B \times C)$: a triple is a value paired with a pair. This is the data-level twin of the currying seen in function types, and the two mirror each other throughout the theory. A function of a pair and a curried function of two arguments carry the same information, related by a standard back-and-forth.

```
def curry   (f : A × B → C) : A → B → C := fun a b => f (a, b)
def uncurry (f : A → B → C) : A × B → C := fun p => f p.1 p.2
```

The pair of conversions witnesses that $A \times B \to C$ and $A \to B \to C$ are interchangeable ways of describing a function of two inputs. Which to use is a matter of convenience; Lean leans on the curried form by default and reaches for products when two results, rather than two arguments, must travel together.

2.10 Sums and choice

Where a product carries both components, a *sum* carries exactly one, tagged with which. The sum of A and B is written $A \oplus B$, and its two constructors, `Sum.inl` and `Sum.inr`, inject a value of the left or right type. Reading a sum means matching on the tag.

```
def tagged : Sum Nat String := Sum.inl 5

def describe : Sum Nat String → String
  | Sum.inl n => "a number"
  | Sum.inr s => "a string"

#eval describe (Sum.inl 7)      -- "a number"
```

```
#eval describe (Sum.inr "hi")   -- "a string"
```

The contrast between product and sum is the contrast between *and* and *or*, and it is no accident that the logical connectives of Chapter 5 reuse exactly these type formers. A value of $A \times B$ supplies an A and a B; a value of $A \oplus B$ supplies an A or a B. The whole of the propositional connectives is prefigured in these two ways of combining data.

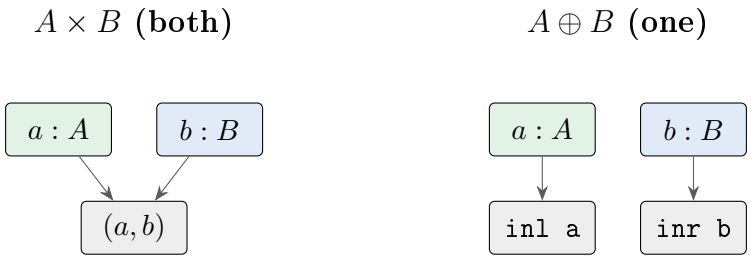


Figure 2.4: Product versus sum. A product needs both components to build a value; a sum needs exactly one, tagged by which side it came from. These are the data-level forms of conjunction and disjunction.

2.11 Optional values

A recurring need is a value that may be absent: the result of a lookup that might fail, the head of a list that might be empty. Rather than a null that lurks inside every type, Lean makes absence explicit with the type `Option` α, whose two constructors are `none` for absence and `some a` for a present value.

```
def safeHead : List α → Option α
  | []      => none
  | x :: _  => some x

#eval safeHead [1, 2, 3]           -- some 1
#eval safeHead ([] : List Nat)     -- none
```

Because absence is a constructor rather than a hidden state, the type checker forces every use of an optional value to account for the `none` case. A function that reads a `some` must say what it does when handed a `none`, and forgetting to is a type error, not a crash at run time. The possibility of failure is lifted into the type, where the checker can see it.

Example 2.5 (Chaining optional computations)**.** When one optional result feeds another, matching at every step becomes tedious. The operation `bind`, written with a specialized

notation, threads the `none` case through automatically: if any step yields `none`, the whole chain does.

```
def secondHead (xss : List (List α)) : Option α :=
  match xss with
  | []       => none
  | xs :: _  => safeHead xs

#eval secondHead [[1,2],[3]]        -- some 1
#eval secondHead ([[]] : List (List Nat))   -- none
```

The absent case propagates without a cascade of matches. Making failure a value, rather than an exception, is what allows it to be composed as smoothly as success.

2.12 Inference at work

The elaborator's two directions, inferring a type from a term and checking a term against a type, cooperate on every line. A short table of expressions and their outcomes shows the interplay, and in particular where an ascription is doing real work.

Expression	Inferred type	Note
`5`	`Nat`	default for a numeral
`(5 : Int)`	`Int`	ascription selects Int
`[1, 2, 3]`	`List Nat`	from the elements
`[]`	*ambiguous*	needs an expected type
`([] : List Bool)`	`List Bool`	ascription resolves it
`(3, "a")`	`Nat × String`	componentwise
`fun x => x + 1`	`Nat → Nat`	from the use of `+ 1`
`none`	*ambiguous*	the element type is open

Table 2.1: Type inference on sample expressions. Where a value is inherently ambiguous, an empty list or a bare `none`, an expected type supplied by ascription or by context resolves it.

The pattern in Table 2.1 is uniform. When a term determines its own type, the elaborator reads it off and no annotation is needed; when a term is polymorphic and underdetermined, a demand from outside must pin it down. Fluency is largely a matter of sensing which case one is in, and reaching for an ascription only in the second.

Remark 2.6. The same demand-driven resolution fills implicit arguments and selects type-class instances. All three, overloaded literals, implicit arguments, and instances, are underdetermined pieces that the surrounding expression completes. Seeing them as one mechanism, rather than three, is what makes the elaborator predictable.

2.13 Writing polymorphic functions

A function that works for every element type is written by taking the type as an argument, usually implicit. The body may not assume anything about the type beyond what its arguments provide, which is precisely what makes the function reusable. Selecting the first element of a list, or a default when it is empty, needs nothing of the element type at all.

```
def firstOr {α : Type} (default : α) : List α → α
  | []      => default
  | x :: _  => x

#eval firstOr 0 [10, 20, 30]         -- 10
#eval firstOr 0 ([] : List Nat)      -- 0
#eval firstOr "?" ["a", "b"]         -- "a"
```

The single definition serves lists of numbers, strings, or anything else, because it treats the elements opaquely. Genuine polymorphism is exactly this discipline of not inspecting what one has abstracted over; a function that needed to compare elements, or add them, would have to demand that capability through a class assumption, as Chapter 8 describes.

Universe polymorphism

A function may be generic not only over a type but over the universe that type inhabits. Writing the type argument as `Sort u`, with a universe variable `u`, lets the same definition apply to data and to propositions alike. The identity function is the purest example.

```
def ident {α : Sort u} (x : α) : α := x

#check @ident        -- @ident : {α : Sort u_1} → α → α
#eval  ident 5       -- 5,      α is Nat, a Type
#check ident trivial -- α is True, a Prop
```

Because `ident` is stated at `Sort u`, it accepts an argument of any universe, a number in `Type` or a proof in `Prop`. Universe polymorphism is what allows a handful of core operations to be written once and reused at every level of the hierarchy, and it is generated automatically for most definitions without the variable ever being written by hand.

2.14 Dependent pairs and subtypes

A product pairs two values of fixed types. A *dependent* pair lets the type of the second depend on the first, packaging a value together with data or a proof about it. The general form is written with Σ, and a common special case is the *subtype*, a value bundled with a proof that it satisfies a property.

```
def Positive := {n : Nat // n > 0}

def five : Positive := ⟨5, by omega⟩

#eval five.val          -- 5,   the underlying number
#check five.property    -- five.property : five.val > 0
```

An element of `Positive` carries a natural in its `.val` field and, in `.property`, a proof that the natural is positive. The proof travels with the value and constrains it, so a function expecting a `Positive` is guaranteed a nonzero input without checking. The proof is erased at run time, leaving just the number, but the type checker has already enforced the constraint.

```
#check (⟨Nat, 5⟩ : Σ α : Type, α)   -- a type paired with an element of it
```

The genuinely dependent pair `⟨Nat, 5⟩` has a first component that is a type and a second component of that very type. This is the data-level counterpart of the existential quantifier: a witness together with something depending on it. Products, subtypes, and existentials are three uses of one construction, the dependent pair, differing only in whether the second component is data or a proof.

2.15 Type abbreviations

A long or meaningful type may be given a short name with `abbrev`. Unlike a structure, an abbreviation introduces no new type: it is transparent, and a value of the abbreviation is interchangeable with a value of the type it names. This documents intent without erecting a barrier the type checker must be told to cross.

```
abbrev Name := String
abbrev Assoc := List (Name × Nat)

def lookup (a : Assoc) (key : Name) : Option Nat :=
```

```
  match a with
  | []            => none
  | (k, v) :: rest => if k == key then some v else lookup rest key
```

The names `Name` and `Assoc` make the signature of `lookup` read as its purpose, an association list mapping names to numbers, while remaining ordinary strings and lists underneath. Because the abbreviation is transparent, a plain `String` may be passed where a `Name` is wanted, with no conversion. The clarity is free.

2.16 Where propositions and data meet

The universe `Prop` of propositions and the universe `Type` of data are kept apart, but they interact at definite points, and `#check` makes the boundary visible. A proposition is a term of `Prop`; a predicate is a function into `Prop`; and the decidability of a proposition is itself a piece of data in `Type`.

```
#check (2 = 2)                  -- Prop
#check (fun n : Nat => n = n)   -- Nat → Prop,  a predicate
#check Decidable (2 = 2)        -- Type,  decidability is data
#check Nat                      -- Type
```

Reading these types locates each expression on the correct side of the boundary. An equation is a proposition; a family of equations indexed by a variable is a predicate valued in `Prop`; and the evidence that lets a program branch on an equation is data. Keeping the three straight, the statement, the predicate, and the decision procedure, is what makes the interplay of proof and computation legible rather than confusing.

Remark 2.7. The separation is not a wall but a membrane. Data determines propositions, as a number determines the equation stating it is even; propositions guard data, as a subtype's proof constrains its value; and decidability carries a proposition into the computational world. The universes are distinct so that proofs may be erased, yet the traffic between them is constant and orderly.

2.17 Three kinds of argument

An argument to a function is supplied in one of three ways, marked by the brackets around its binder. Round brackets make it *explicit*, given by the caller; curly brackets make it *implicit*,

inferred by the elaborator; square brackets make it an *instance* argument, resolved from the registered instances. A single function may use all three.

```
def combine {α : Type} [ToString α] (label : String) (x : α) : String :=
  s!"{label}: {x}"

#eval combine "value" 42       -- "value: 42"
#eval combine "flag" true      -- "flag: true"
```

The caller writes only the explicit arguments, `label` and `x`. The type α is implicit, inferred from `x`; the `ToString` α instance is found automatically so that the interpolation can display `x`. The three bracket kinds are the whole of how information reaches a function, and Table 2.2 summarises which party supplies each.

Brackets	Kind	Supplied by
`(x : T)`	explicit	the caller, positionally
`{x : T}`	implicit	the elaborator, from other arguments
`[C x]`	instance	instance resolution, from the class database

Table 2.2: The three binder kinds. Every argument is provided by the caller, inferred from context, or resolved as an instance, according to its brackets.

The at-sign prefix seen earlier makes every argument explicit at once, so that even implicit and instance arguments must be written by hand. This is occasionally needed when the elaborator cannot infer an implicit, or when one wishes to see the full form a term takes.

2.18 Named and optional arguments

Explicit arguments are ordinarily given by position, but they may also be given by name, which is clearer when a function has several and helps when one wishes to skip an optional one. An argument with a default value may be omitted entirely, the default standing in.

```
def greet (name : String) (greeting : String := "Hello") : String :=
  s!"{greeting}, {name}!"

#eval greet "Ada"                       -- "Hello, Ada!"
#eval greet "Ada" "Hi"                  -- "Hi, Ada!"
#eval greet (name := "Ada")             -- "Hello, Ada!"
#eval greet "Grace" (greeting := "Hi") -- "Hi, Grace!"
```

The parameter `greeting` has the default `"Hello"`, so a single argument suffices; supplying it, by position or by the named form `(greeting := "Hi")`, overrides the default. Named arguments make a call self-documenting and let optional parameters be set selectively. Defaults and names together keep a flexible function convenient at its common uses while remaining fully adjustable.

2.19 When two functions are equal

Two functions are equal when they agree on every input. This principle, *function extensionality*, is provided by `funext`: from a proof that `f x = g x` for all `x`, it concludes `f = g`. It is what lets one prove an equation between functions rather than merely between their values.

```
theorem double_is_add (n : Nat) : 2 * n = n + n := by ring

theorem funs_equal :
    (fun n => 2 * n) = (fun n => n + n) := by
  funext n
  ring
```

The first theorem is a fact about values, holding for each `n`. The second is a fact about the functions themselves, and `funext` bridges the two: after `funext n` the goal becomes the pointwise equation, closed by `ring`. Without extensionality one could prove the values equal at every point yet not conclude the functions equal; `funext` is precisely the permission to make that step.

Remark 2.8. Function extensionality is not automatic in a bare type theory; it is a consequence of the axioms Lean adopts, holding uniformly for all function types. Its effect is that functions may be compared by behaviour, which is usually what one means by their equality. The analogous principle for propositions, that logically equivalent propositions are equal, is likewise available, so equality throughout the system tracks the intended notion rather than mere syntactic identity.

2.20 Terser polymorphic signatures

Writing `{α : Type}` before every polymorphic definition is repetitive, and Lean offers two abbreviations. An identifier left unbound in a signature is automatically inserted as an implicit argument, and the notation `Type*` stands for a type in some universe without naming the level. Together these let a generic signature omit the ceremony.

```
def pair (x : α) (y : β) : α × β := (x, y)

#check @pair      -- @pair : {α β} → α → β → α × β
```

Although α and β are never declared, they are treated as implicit type arguments, inserted automatically because they appear unbound. The definition is exactly as general as one written with explicit {α β : Type} binders, but shorter. This autobinding is why so many library signatures mention type variables that seem to come from nowhere: they are implicit arguments the elaborator has supplied on the author's behalf.

2.21 Two kinds of equality

A distinction runs through the whole system: some equations hold by computation, and some require a proof. An equation whose two sides reduce to a common form is *definitionally* equal, and `rfl` proves it. An equation that is true but whose sides do not reduce to a common form is only *propositionally* equal, and it needs an argument. Knowing which is which explains exactly when `rfl` succeeds.

```
example : 2 + 2 = 4 := rfl              -- definitional: both reduce to 4
example (n : Nat) : n + 0 = n := rfl    -- definitional: add recurses on 0

-- example (n : Nat) : 0 + n = n := rfl  -- FAILS: 0 + n is stuck
example (n : Nat) : 0 + n = n := by
  induction n <;> simp_all              -- propositional: needs a proof
```

The equation `n + 0 = n` holds definitionally because addition recurses on its second argument: with `0` there, the definition reduces `n + 0` straight to `n`. The equation `0 + n = n` does not, because `n` is a variable and the recursion on the second argument is stuck at the head; the two sides never meet by reduction, so `rfl` fails and an induction is required. The symmetry of the two equations is only apparent; computationally they are quite different.

Remark 2.9. Definitional equality is what the type checker decides automatically; propositional equality is what theorems establish. The line between them is drawn by reduction: whatever the checker can compute to a common form, it treats as identical, and everything else must be proved. This is why `rfl` sometimes closes a goal instantly and sometimes fails on an equation one knows to be true, and the failure is always a sign that reduction alone does not reach it.

2.22 Functions that compute types

Because types are terms, a function may return a type, computed from its input. Such a function lets the type of a later value depend on an earlier one in an arbitrary way, and it is the mechanism behind the interpreter of Chapter 4 and behind length-indexed structures. A tuple type of a chosen arity is a clear illustration.

```
def Tuple : Nat → Type
  | 0     => Unit
  | n + 1 => Nat × Tuple n

#check (() : Tuple 0)              -- the empty tuple
#check ((1, ()) : Tuple 1)         -- a one-tuple
#check ((1, 2, ()) : Tuple 2)      -- a two-tuple
```

The function `Tuple` recurses on a number to build a type: zero yields `Unit`, and a successor prepends a `Nat` factor. The type `Tuple 2` computes to `Nat × (Nat × Unit)`, so a value of it is a pair of numbers ending in the unit. A function over such a type may then produce a result whose type is `Tuple n` for the `n` it was given, a dependency no fixed type could express. Type-level computation is what makes the type of an expression respond to its value.

2.23 How far definitions unfold

The checker unfolds definitions to decide definitional equality, but not all definitions unfold equally readily. An attribute governs each definition's *transparency*: a `reducible` definition unfolds freely, an ordinary one unfolds when needed, and an `irreducible` one does not unfold at all. This controls both performance and how a definition behaves in proofs.

```
@[reducible] def NatId := Nat      -- unfolds freely, like an abbreviation
def NatDef := Nat                  -- unfolds when the checker needs to
@[irreducible] def NatOpaque := Nat -- never unfolds

example : NatId = Nat := rfl       -- succeeds: NatId unfolds
```

A `reducible` definition is transparent to the elaborator and to instance resolution, so `abbrev`, which is `reducible` by default, is interchangeable with the type it names. An `irreducible` definition is opaque: it hides its unfolding, which can speed up checking and

can prevent a proof from depending on an implementation detail that might later change. The default lies between, unfolding when a computation demands it but not otherwise.

Remark 2.10. Transparency is a tuning knob rarely touched at first but occasionally decisive. Marking a definition `irreducible` fixes an abstraction boundary, forcing later proofs to go through its stated properties rather than its internals, which keeps them robust to changes in the definition. Marking one `reducible` does the opposite, exposing it fully. The default suits most code, and the attributes are reached for only when a proof or a performance problem calls for controlling how much the checker sees.

Exercises

Exercise 2.1. Predict the type Lean reports for each of the following, then check your answers in the editor: `#check 3`, `#check (3 : Int)`, `#check (3.0)`, `#check "3"`, `#check Bool`.

Exercise 2.2. Explain, in terms of the layered tower of Figure 2.1, why `#check Type 5` succeeds and reports `Type 6`, whereas there is no expression whose type is reported as the top of the hierarchy.

Exercise 2.3. The function `Nat.add` has type `Nat → Nat → Nat`. Write down the type of `Nat.add 3`, and then the type of `Nat.add 3 4`. Verify both with `#check`.

Exercise 2.4. Define `const : Nat → Nat → Nat` returning its first argument and ignoring its second. What is the type of `const 5`, and what does `const 5 99` evaluate to?

Exercise 2.5. Using an explicit type ascription, write two expressions built from the literal `0` that Lean assigns the types `Nat` and `Int` respectively.

Exercise 2.6. Give the typing rule, in the style of the application figure, for forming a pair `(a, b)` from terms `a : A` and `b : B`. What is the type of the result?

Exercise 2.7. The identity function was written twice, once with an explicit type argument and once with an implicit one. Write a third version whose type is `Bool → Bool` and takes no type argument at all. Which of the three is least reusable, and why?

Exercise 2.8. Uncomment a line that applies a function to an argument of the wrong type and read the resulting error. Which two types does Lean report, and which word does it underline?

Exercise 2.9. * The type of `@idImp` is $(\alpha : \texttt{Type}) \to \alpha \to \alpha$. Describe in words the difference between this dependent type and the non-dependent type `Type → Type → Type`, and give a term inhabiting each.

Exercise 2.10. ⋆ Lean identifies `Type` n with `Sort` $(n+1)$ and `Prop` with `Sort` 0. Using `#check`, determine what type Lean assigns to `Sort 0`, to `Sort 1`, and to `Sort 2`, and reconcile the answers with Figure 2.2.

Exercise 2.11. Build a triple of a `Nat`, a `Bool`, and a `String`, report its type with `#check`, and extract its middle component with the projections.

Exercise 2.12. Define `curry` and `uncurry` for the specific types `Nat` × `Nat` → `Nat` and confirm that converting a function one way and back leaves its behaviour unchanged on a sample input.

Exercise 2.13. Write `describe : Sum Bool Nat → String` distinguishing the two sides, and evaluate it on a value built with each constructor.

Exercise 2.14. Define `safeTail : List` α → `Option (List` α`)` returning `none` on the empty list. Test it on `[1,2,3]` and on the empty list.

Exercise 2.15. Add a row to Table 2.1 for the expression `(fun x => (x, x))`. What type does Lean infer, and is any ascription needed?

Exercise 2.16. ⋆ The type `Option` α carries either nothing or a value of α. Exhibit a pair of functions converting between `Option` α and `Sum Unit` α, and argue that they are mutually inverse.

Exercise 2.17. Write a polymorphic function `lastOr` α `: Type (d :` α`) : List` α → α returning the last element or a default. Test it on a list of numbers and a list of strings.

Exercise 2.18. Apply `ident` to the number `7` and to the proof `trivial`. Using `#check`, report the universe of the type argument in each case.

Exercise 2.19. Define the subtype `Positive := {n : Nat // n > 0}` and construct an element. Extract its `.val` and state the type of its `.property`.

Exercise 2.20. Exhibit a genuine dependent pair of type Σ α `: Type,` α, choosing a type and an element of it. What are the two components?

Exercise 2.21. Use `abbrev` to name a type of your choosing and write one function whose signature reads more clearly for the name. Confirm a plain value of the underlying type is still accepted.

Exercise 2.22. ⋆ Describe the relationship between the subtype `{x // p x}`, the dependent pair Σ `x, p x`, and the existential ∃ `x, p x`. Which live in `Type` and which in `Prop`, and why?

Exercise 2.23. Write a function using an explicit, an implicit, and an instance argument, and state which party supplies each when it is called.

Exercise 2.24. Define a function with one required and one defaulted argument, and call it three ways: with the default, with the optional argument by position, and with it by name.

Exercise 2.25. Prove that `(fun n => n + n)` equals `(fun n => 2 * n)` using `funext` and `ring`.

Exercise 2.26. Using autobound implicits, write a polymorphic `swap` : $\alpha \times \beta \to \beta \times \alpha$ without declaring the type variables, and confirm its inferred type with `#check`.

Exercise 2.27. Explain what the at-sign prefix does to a function's arguments, and use it to apply a function while supplying an implicit argument by hand.

Exercise 2.28. ⋆ Prove `(fun x : Nat => x + 0) = (fun x => x)` with `funext`. Which lemma closes the pointwise goal, and why is extensionality needed to finish?

Exercise 2.29. Prove `n + 0 = n` with `rfl`. Then attempt `0 + n = n` with `rfl`, observe the failure, and explain it in terms of reduction.

Exercise 2.30. For each of `2 * 3 = 6`, `n * 1 = n`, and `1 * n = n`, decide whether it holds definitionally, and confirm by trying `rfl`.

Exercise 2.31. Define `Tuple` as in the text and construct a value of `Tuple 3`. What concrete type does `Tuple 3` compute to?

Exercise 2.32. Write a type-level function mapping a boolean to a type, sending `true` to `Nat` and `false` to `Bool`, and construct a value at each.

Exercise 2.33. Mark a definition `@[reducible]` and confirm that `rfl` proves it equal to the type it names. What changes if you make it `@[irreducible]`?

Exercise 2.34. ⋆ Explain a situation in which marking a definition `irreducible` improves a development, and describe the effect on proofs that previously relied on its unfolding.

Chapter 3

Functions, Definitions, and Recursion

A functional language earns its name by making functions the primary way to build everything else. In Lean a program is a collection of definitions, most of them functions, and running a program means reducing an expression by unfolding those definitions until no reduction remains. This chapter assembles the machinery: how to name a function, how to take a value apart by pattern matching, and how to make a function refer to itself without looping forever.

3.1 Naming with `def`

The keyword `def` attaches a name to a term. The general form lists the name, then the parameters with their types, then the result type after a colon, then the body after `:=`.

```
def double (n : Nat) : Nat := n + n
def area (w h : Nat) : Nat := w * h
def greeting : String := "hello"

#eval double 21        -- 42
#eval area 3 4         -- 12
#eval greeting         -- "hello"
```

When two consecutive parameters share a type, as with `w h : Nat`, they may be grouped. The result annotation is optional whenever Lean can infer it, but stating it is good discipline: it documents intent and it turns a mistake in the body into an immediate, local error rather than a puzzling one at the call site.

Remark 3.1. A definition introduced with `def` is transparent: the type checker may unfold it whenever convenient. This is what allows `double 21` to reduce to `42`. Definitions marked

`opaque` or built as proofs behave differently, but the ordinary case is simple substitution.

3.2 Anonymous functions

A function need not be named to be used. The keyword `fun` introduces a function value directly, binding its argument before a `=>` and giving the body after it.

```
#check fun x => x + 1         -- Nat → Nat
#eval  (fun x => x + 1) 9     -- 10
#eval  (fun x y => x * y) 6 7 -- 42
```

For short bodies Lean offers a terser form: a centred dot · marks the position of an argument, and the surrounding parentheses delimit the function. Thus `(· + 1)` abbreviates `fun x => x + 1`, and `(· * ·)` abbreviates `fun x y => x * y` with two holes filled left to right.

Currying, concretely

Every multi-argument function is a nest of single-argument ones. The definition `fun x y => x * y` is sugar for `fun x => fun y => x * y`, and applying it to arguments peels the layers one at a time.

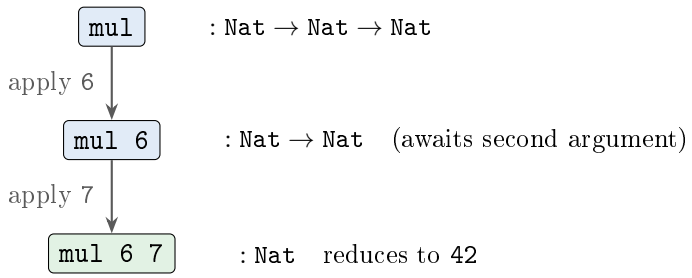


Figure 3.1: Applying a curried function one argument at a time. Each application consumes one input and returns a function of the remaining inputs, until no arrows are left and the result is a plain value.

The practical payoff is that partial application costs nothing special. A function of two arguments is already, by construction, a function that returns a function, so supplying one argument and keeping the rest for later is the ordinary behaviour rather than a bolted-on feature.

3.3 Taking values apart

To compute with a value one must inspect its shape. Pattern matching does this: it lists the possible forms the input can take and gives a result for each. Booleans have two forms, so a function on them has two branches.

```
def negate (b : Bool) : Bool :=
  match b with
  | true  => false
  | false => true

#eval negate true      -- false
#eval negate false     -- true
```

Matching binds the pieces of a compound value to names. A pair splits into its components, and a list splits into either the empty list or a head followed by a tail.

```
def fst' (p : Nat × Nat) : Nat :=
  match p with
  | (a, _) => a

def firstOrZero (xs : List Nat) : Nat :=
  match xs with
  | []      => 0
  | x :: _  => x

#eval fst' (3, 8)              -- 3
#eval firstOrZero [7, 8, 9]    -- 7
#eval firstOrZero []           -- 0
```

The underscore is a wildcard: it matches anything and binds no name, signalling that the piece is deliberately unused. Lean checks that the branches cover every possible form; a missing case is an error, not a silent gap, which is what makes a well-typed function on a finite variety of shapes total by construction.

3.4 Recursion

A function may call itself. The essential requirement is that each recursive call be made on a structurally smaller argument, so that the descent cannot continue forever. Factorial is the archetype: the case for `0` is immediate, and the case for a successor is defined in terms

of the predecessor.

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

#eval fact 0      -- 1
#eval fact 5      -- 120
#eval fact 10     -- 3628800
```

The pattern `n + 1` is not arithmetic; it is a way of naming the shape "some number one larger than `n`". Matching a natural against it binds `n` to the predecessor, which is strictly smaller, so the recursion is well founded. Lean verifies this automatically and accepts the definition.

How a recursive call unfolds

Evaluation proceeds by repeatedly replacing a call with the body it abbreviates. Tracing `fact 3` exposes the descent and the subsequent multiplication as the stack unwinds.

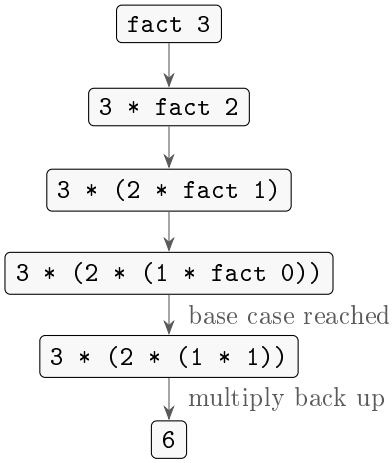


Figure 3.2: Reduction of `fact 3`. The descent replaces each call by an open multiplication until the base case supplies `1`; then the pending multiplications collapse to the answer.

Not every recursion decreases a single argument in the obvious way. The Fibonacci function recurses twice, on the two predecessors, and still terminates because both arguments are smaller than the original.

```
def fib : Nat → Nat
  | 0     => 0
```

```
  | 1     => 1
  | n + 2 => fib n + fib (n + 1)

#eval fib 10      -- 55
#eval fib 15      -- 610
```

Example 3.2 (Length of a list)**.** Recursion on a list mirrors recursion on a number: the empty list is the base case, and a non-empty list is one element added to a shorter list.

```
def length' : List α → Nat
  | []      => 0
  | _ :: xs => 1 + length' xs

#eval length' [10, 20, 30]      -- 3
#eval length' ([] : List Nat)   -- 0
```

The recursive call is on `xs`, the tail, which is one element shorter than the input. The same structural descent that made `fact` legitimate makes `length'` legitimate.

3.5 Termination and the checker

Lean insists that recursion terminate, because a function that might run forever cannot be treated as a total mathematical object, and totality is what allows a function to double as a proof later on. For structural recursion the check is automatic. When the decreasing quantity is less obvious, the author can supply a measure with a `termination_by` clause, naming an expression that strictly decreases at each call.

```
def gcd : Nat → Nat → Nat
  | 0, n => n
  | m + 1, n => gcd (n % (m + 1)) (m + 1)
termination_by m n => m
```

If no measure can be found, or the author wishes to opt out of the guarantee, the modifier `partial` marks a definition as possibly nonterminating. Such a function still runs, but Lean no longer regards it as a total object and will not reason about it as one.

Remark 3.3. The termination requirement is the price of a language in which functions and proofs are the same kind of thing. A partial function is perfectly usable for computation; it simply forfeits the mathematical guarantees that totality would otherwise confer.

3.6 Local definitions

Two constructs introduce names of limited scope. A `let` binds a value inside an expression, and a `where` block attaches auxiliary definitions to the end of a function. Both keep helper names out of the global namespace and let a computed intermediate be shared.

```
def roots (a b c : Int) : Int × Int :=
  let d := b * b - 4 * a * c
  (d, -d)

def sumSquares (n : Nat) : Nat :=
  go n
where
  go : Nat → Nat
    | 0     => 0
    | k + 1 => (k + 1) * (k + 1) + go k

#eval roots 1 (-3) 2      -- (1, -1)
#eval sumSquares 3        -- 14
```

The `let` form evaluates `d` once and uses it twice, avoiding both repetition and recomputation. The `where` form gives `sumSquares` a private recursive helper `go` that is invisible elsewhere.

3.7 Functions that take functions

Because functions are ordinary values, a function may accept another function as an argument or return one as a result. These *higher-order* functions capture patterns of computation once and reuse them everywhere.

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

#eval applyTwice (· + 3) 10      -- 16
#eval applyTwice double 5        -- 20
```

The standard library builds its list operations this way. `List.map` applies a function to every element, and `List.filter` keeps the elements satisfying a test.

```
#eval List.map (· * ·) [1,2,3] |>.map (· 10)   -- [10, 20, 30]
#eval List.map (· * 2) [1, 2, 3, 4]             -- [2, 4, 6, 8]
```

```
#eval List.filter (· % 2 == 0) [1,2,3,4,5,6]   -- [2, 4, 6]
```

Example 3.4 (Composition)**.** Composing two functions produces a third that applies them in sequence. Lean writes composition with the circle ∘, and it is itself an ordinary higher-order function.

```
def inc (n : Nat) : Nat := n + 1
def sq  (n : Nat) : Nat := n * n

#eval (sq ∘ inc) 4        -- 25, squares after incrementing
#eval (inc ∘ sq) 4        -- 17, increments after squaring
```

The order matters: `sq ∘ inc` feeds its input to `inc` first, then to `sq`, so the rightmost function acts first.

3.8 Nested and multiple patterns

A match may inspect several arguments at once and may look several layers deep into a value. Listing patterns for each argument, separated by commas, produces a table of cases; nesting a pattern inside another reaches into a compound value in a single step. Deciding whether two booleans are equal is a four-line table.

```
def beq : Bool → Bool → Bool
  | true,  true  => true
  | false, false => true
  | _,     _     => false

#eval beq true true      -- true
#eval beq true false     -- false
```

The wildcard row collapses the two remaining cases, since both mismatched combinations yield `false`. Nesting goes the other direction, taking a structured value apart in one pattern. Detecting a list of at least two elements matches a `cons` whose tail is itself a `cons`.

```
def firstTwo : List α → Option (α × α)
  | x :: y :: _ => some (x, y)
  | _           => none

#eval firstTwo [1, 2, 3]           -- some (1, 2)
```

```
#eval firstTwo [1]                -- none
```

The pattern `x :: y :: _` binds the first two elements and ignores the rest, matching only lists long enough to supply them. Reading two layers deep in one pattern is clearer than two nested matches, and the checker still confirms that the cases are exhaustive.

3.9 Guards and conditionals

Sometimes the branch depends not on the shape of a value but on a test of it. A conditional `if...then...else` chooses on a boolean, and it may appear as the body of a branch, combining shape-matching with value-testing. Classifying a number by sign uses a match on nothing and a pair of tests.

```
def sign (n : Int) : Int :=
  if n > 0 then 1
  else if n < 0 then -1
  else 0

#eval sign (-5)      -- -1
#eval sign 0         -- 0
#eval sign 42        -- 1
```

The conditional is itself an expression: it has a value, of the common type of its two branches, and can appear anywhere a value is expected. This is unlike the statement-oriented conditional of many languages, where a branch performs an action; here a branch *is* a value, and the whole `if` evaluates to one.

3.10 Accumulators and tail recursion

The recursion in `fact` of the earlier section builds a tower of pending multiplications, each waiting for the recursive call beneath it to return. An alternative carries a running result as an extra argument, an *accumulator*, so that nothing is left pending and the recursive call is the last thing the function does. Such *tail-recursive* functions run in constant stack space.

```
def factAux : Nat → Nat → Nat
  | acc, 0     => acc
  | acc, n + 1 => factAux (acc * (n + 1)) n

```

```
def factTail (n : Nat) : Nat := factAux 1 n

#eval factTail 5      -- 120
```

The two styles compute the same numbers by different routes. The plain version defers its multiplications and performs them on the way out; the accumulating version performs each multiplication on the way in, reaching the base case with the answer already in hand. Tracing both on a small input exposes the difference.

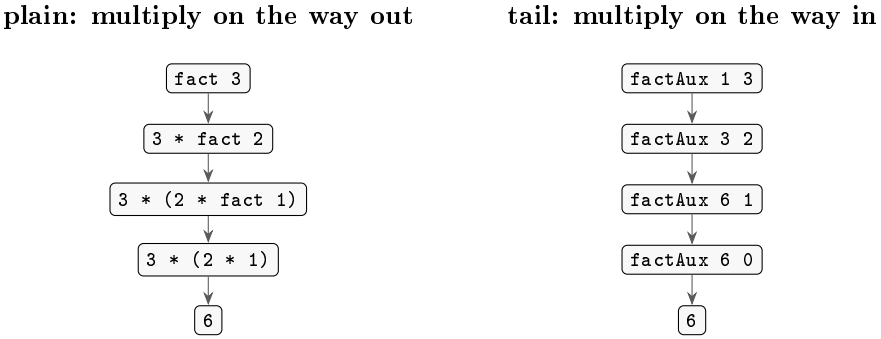


Figure 3.3: Two evaluations of factorial at 3. The plain recursion leaves a growing stack of pending multiplications; the accumulating recursion keeps a single running product and returns it directly at the base case.

Remark 3.5. The accumulating version is not merely faster; for large inputs it is the difference between running and exhausting the stack. The transformation, turning a deferred computation into a running one carried as an argument, is a standard technique, and recognising when a plain recursion can be rewritten this way is a useful reflex.

3.11 Mutual recursion

Two functions may be defined in terms of each other. The keyword `mutual` groups them so that each may call the other, and Lean checks that the combined recursion still descends. Testing a number's parity without arithmetic is the classic pair: even and odd defined through one another.

```
mutual
  def isEven : Nat → Bool
    | 0     => true
    | n + 1 => isOdd n
  def isOdd : Nat → Bool
```

```
    | 0     => false
    | n + 1 => isEven n
end

#eval isEven 10      -- true
#eval isOdd 7        -- true
```

Each call passes to the other function an argument one smaller, so the mutual descent terminates for the same reason a single recursion would. The two definitions are legible precisely because they mirror the mathematical characterisation: a successor is even exactly when its predecessor is odd, and conversely.

3.12 Folding a list

A great many list functions share one shape: start from a value for the empty list, and combine each element with the result of the rest. This shape is captured once by `foldr`, which takes the combining operation and the starting value and applies them down the list. Summing and multiplying are two instances.

```
#eval List.foldr (· + ·) 0 [1, 2, 3, 4]      -- 10
#eval List.foldr (· * ·) 1 [1, 2, 3, 4]      -- 24
#eval List.foldr (· :: ·) [] [1, 2, 3]       -- [1, 2, 3]
```

The name records the direction of association: `foldr` brackets from the right, so summing $[1,2,3,4]$ computes $1+(2+(3+(4+0)))$. Its companion `foldl` brackets from the left and carries an accumulator, relating to `foldr` as the tail-recursive factorial relates to the plain one. Seeing a list operation as a fold reveals how much of the standard library is one pattern instantiated many ways.

Example 3.6 (Length and map as folds)**.** Even functions that seem structural are folds in disguise. Length folds with a step that ignores the element and adds one; mapping folds with a step that applies a function and prepends.

```
def lengthF (xs : List α) : Nat :=
  List.foldr (fun _ acc => acc + 1) 0 xs

def mapF (f : α → β) (xs : List α) : List β :=
  List.foldr (fun x acc => f x :: acc) [] xs

#eval lengthF [10, 20, 30]          -- 3
```

```
#eval mapF (· * 2) [1, 2, 3]        -- [2, 4, 6]
```

Both are the fold pattern with a particular step and starting value. That so many operations collapse to `foldr` is why it, rather than any one of them, is the primitive worth knowing.

3.13 Pipelines

When a value passes through a series of operations, nesting the calls inside out becomes hard to read. The pipeline operator `|>` reverses the order, feeding a value into the function on its right, so a computation reads left to right in the order it happens. A dotted form `|>.f` continues with a method call.

```
#eval [1, 2, 3, 4] |>.map (· * 2) |>.filter (· > 4)   -- [6, 8]
#eval "hello" |>.toList |>.length                      -- 5
#eval 5 |> (· + 1) |> (· * 2)                          -- 12
```

The first line doubles every element and keeps those exceeding four, written in the sequence the operations occur rather than the reverse. The same computation with ordinary application, `List.filter (· > 4) (List.map (· * 2) [1,2,3,4])`, reads from the inside and is harder to follow. There is also a leftward form `<|` that applies a function to an argument without parentheses, useful when the argument is itself a long expression.

3.14 A tour of list operations

Most work with lists is done through a standard vocabulary of operations, each a higher-order function or a simple transformation. Knowing the handful in Table 3.1 covers the great majority of list processing, and each is itself defined by the structural recursion of Chapter 4.

```
#eval List.range 5                          -- [0, 1, 2, 3, 4]
#eval [1, 2, 3, 4, 5].filter (· % 2 == 1)   -- [1, 3, 5]
#eval [1, 2, 3, 4].foldl (· + ·) 0          -- 10
#eval List.zip [1, 2, 3] ["a", "b", "c"]    -- [(1,"a"),(2,"b"),(3,"c")]
#eval ([1,2,3,4,5].drop 2).take 2           -- [3, 4]
```

The operations compose freely, and a pipeline of them expresses a transformation as a sequence of stages. That each is an instance of a general recursion scheme is why they behave predictably and why proving facts about them, as Chapter 7 does, follows a uniform pattern.

Operation	Effect	Example result
`List.range n`	the numbers `0` to `n-1`	`[0,1,2,3]`
`.map f`	apply `f` to each element	`[2,4,6]`
`.filter p`	keep elements satisfying `p`	`[2,4]`
`.foldl f z`	combine left to right from `z`	a single value
`.foldr f z`	combine right to left from `z`	a single value
`.append`	concatenate two lists (`++`)	`[1,2,3,4]`
`.reverse`	reverse the order	`[3,2,1]`
`.take n`	the first `n` elements	`[1,2]`
`.drop n`	all but the first `n`	`[3,4]`
`List.zip`	pair up two lists	`[(1,'a'),(2,'b')]`

Table 3.1: Common list operations. Each is a higher-order function or a short structural recursion; combined through pipelines they express most list processing concisely.

3.15 Recursion that is not structural

Structural recursion descends into the immediate parts of a value. Some natural definitions instead shrink an argument by an amount the compiler cannot see is structural, and these need an explicit measure. Computing a binary logarithm by repeated halving is such a case: the argument drops to about half, which is smaller but not a syntactic predecessor.

```
def log2 : Nat → Nat
  | 0 => 0
  | 1 => 0
  | n + 2 => log2 ((n + 2) / 2) + 1
termination_by n => n
decreasing_by simp_wf; omega
```

The clause `termination_by` names the quantity that decreases, here the argument itself, and `decreasing_by` discharges the obligation that it really does: that `(n + 2) / 2` is less than `n + 2`, which `omega` confirms after `simp_wf` exposes the goal. With the measure supplied, Lean accepts the definition as total and it computes like any other.

```
#eval log2 1       -- 0
#eval log2 8       -- 3
#eval log2 100     -- 6
```

Remark 3.7. Well-founded recursion is the general form of which structural recursion is the automatic special case. Any argument that strictly decreases in a well-founded order licenses a recursive call; the naturals under `<` are the usual order, and the measure is whatever

quantity one can show descends. This is the computational twin of the strong induction of Chapter 7, and the two are proved sound together.

3.16 Sequencing computations that may fail

When several steps each return an `Option`, threading the failure by hand is tedious. A `do` block over `Option` does it automatically: binding a result with an arrow proceeds only if the step succeeded, and any `none` short-circuits the whole block. Indexing a list with `[i]?` returns an `Option`, absent when the index is out of range.

```
def firstTwoSum (xs : List Nat) : Option Nat := do
  let a ← xs[0]?
  let b ← xs[1]?
  return a + b

#eval firstTwoSum [10, 20, 30]      -- some 30
#eval firstTwoSum [10]              -- none
#eval firstTwoSum []                -- none
```

Each bound step may fail; if the list is too short, the corresponding `[i]?` yields `none` and the block returns `none` without evaluating the rest. The `do` notation makes a chain of optional computations read as a straight-line sequence, with the failure handling implicit. The same notation serves `IO` and other effectful types, so the single construct threads whatever the surrounding type requires.

Example 3.8 (Combining pipelines and options)**.** The two ideas compose. A safe average divides a sum by a length, failing on the empty list, and reads naturally as a short sequence.

```
def average (xs : List Nat) : Option Nat :=
  if xs.isEmpty then none
  else some (xs.foldl (· + ·) 0 / xs.length)

#eval average [2, 4, 6]      -- some 4
#eval average []             -- none
```

The guard rules out division by zero, and the pipeline of `foldl` and `length` computes the mean. Making the empty case a `none`, rather than a crash, keeps the failure visible in the type.

3.17 Functions that capture

A function returned from another may refer to the arguments of the outer function, which it carries with it. Such a function is a *closure*: it closes over the values in scope when it was built, and those values persist in it after the outer call has returned. An adder factory makes the idea concrete.

```
def adder (n : Nat) : Nat → Nat := fun m => n + m

def add5 : Nat → Nat := adder 5
def add100 : Nat → Nat := adder 100

#eval add5 10        -- 15
#eval add100 10      -- 110
```

Each call to `adder` produces a different function, remembering the `n` it was given. The function `add5` carries the value `5`, `add100` carries `100`, and neither forgets it. This is how a function can be specialised by supplying some of its data in advance, and it is the mechanism behind partial application: `adder 5` is a closure over `5`, exactly as `Nat.add 5` is.

Example 3.9 (Building functions in a list)**.** Closures may be stored like any value, so a list of functions, each capturing a different number, is unremarkable. Applying each to a common input yields a list of results.

```
def adders : List (Nat → Nat) := [adder 1, adder 10, adder 100]

#eval adders.map (fun f => f 5)      -- [6, 15, 105]
```

Each function in the list remembers its own captured value, and mapping application over the list applies all three to `5`. Functions are values, and capturing is what gives each stored function its distinct behaviour.

3.18 Local recursion

A helper function needed only inside one definition can be declared locally with `let rec`, keeping it out of the surrounding namespace. This is the term-level counterpart of the `where` block, and it is the natural home for an accumulator loop that no other definition should see.

```
def sumList (xs : List Nat) : Nat :=
  let rec go : List Nat → Nat → Nat
    | [],      acc => acc
    | y :: ys, acc => go ys (acc + y)
  go xs 0

#eval sumList [1, 2, 3, 4]      -- 10
```

The local `go` carries the running total and is invisible outside `sumList`. Declaring it with `let rec` rather than as a top-level definition signals that it is an implementation detail, and it keeps the file's namespace uncluttered by helpers. The recursion is checked for termination exactly as a top-level one would be; locality changes the scope, not the rules.

3.19 The function toolkit

A handful of small operations on functions recur so often that the standard library names them. The identity returns its argument unchanged; `const` ignores an argument and returns a fixed value; `flip` swaps the order of two arguments; and composition, written ∘, chains two functions. Each is an ordinary higher-order function.

```
#eval id 5                          -- 5
#eval Function.const Nat 7 99       -- 7,   second argument ignored
#eval flip Nat.sub 3 10            -- 7,   computes 10 - 3
#eval ((· + 1) ∘ (· * 2)) 5        -- 11,  double then increment
```

Operation	Meaning	Effect
`id x`	the identity	returns `x`
`const b a`	the constant function	returns `a`, ignores `b`
`flip f a b`	swap arguments	computes `f b a`
`(g ∘ f) x`	composition	computes `g (f x)`

Table 3.2: The common function combinators. Each is a higher-order function that builds a new function from given ones, and together they express many small manipulations without a fresh `fun`.

These combinators let a transformation be assembled from parts rather than written out. A pipeline of compositions, or a `flip` to adapt an argument order, often replaces an explicit anonymous function, and the named form states the intent more directly than the lambda would.

Remark 3.10. The combinators are not magic; each has a one-line definition in terms of `fun`. `id` is `fun x => x`, `const` is `fun a _ => a`, and `flip f` is `fun a b => f b a`. Naming them pays off because they compose: a small vocabulary of function-building operations covers a large range of manipulations, and reading `flip Nat.sub` is quicker than decoding the equivalent lambda.

3.20 Loops and local mutation

A functional language can wear an imperative face. Inside a `do` block a variable may be declared `mut` and reassigned, and a `for` loop may iterate over a collection, updating it. This reads exactly like imperative code, yet it compiles to ordinary pure functions; the mutation is local and invisible from outside.

```
def sumLoop (xs : List Nat) : Nat := Id.run do
  let mut total := 0
  for x in xs do
    total := total + x
  return total

#eval sumLoop [1, 2, 3, 4]      -- 10
```

The variable `total` is initialised, updated in each iteration, and returned, precisely as a loop in an imperative language would. The wrapper `Id.run` runs the block as a pure computation, so `sumLoop` is an ordinary function with no side effects, despite its appearance. This style is available whenever a loop with an accumulator is clearer than the equivalent fold, and it costs nothing: the compiled result is the same.

Example 3.11 (Building a list in a loop)**.** The imperative style extends to constructing a result. Reversing a list by repeatedly prepending reads naturally as a loop that grows an accumulator.

```
def reverseLoop (xs : List Nat) : List Nat := Id.run do
  let mut acc := []
  for x in xs do
    acc := x :: acc
  return acc

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

Each iteration prepends the current element, so the accumulator ends reversed. The loop and the accumulating recursion of an earlier section compute the same thing; the loop simply presents it in the shape a programmer coming from an imperative language expects.

3.21 Partial functions and fuel

Every recursive function so far was total, its termination checked. A function whose termination is unknown, or genuinely absent, may still be written by marking it `partial`, which opts out of the check. Such a function runs, but Lean no longer treats it as a total mathematical object and will not reason about it.

```
partial def collatz (n : Nat) : Nat :=
  if n ≤ 1 then 0
  else if n % 2 == 0 then 1 + collatz (n / 2)
  else 1 + collatz (3 * n + 1)

#eval collatz 27      -- 111
```

Whether `collatz` terminates for every input is a famous open problem, so no termination measure can be supplied, and `partial` is the only way to define it. To recover a total function, one bounds the recursion with a *fuel* argument: a counter that decreases at every call, forcing termination and reporting failure if the fuel runs out.

```
def collatzFuel : Nat → Nat → Option Nat
  | 0,        _ => none                  -- out of fuel
  | _ + 1,    n =>
      if n ≤ 1 then some 0
      else if n % 2 == 0 then (collatzFuel · (n / 2)) _ |>.map (· + 1)
      else none                          -- (odd case elided)
```

The fuel version recurses on the counter, which plainly decreases, so it is total and Lean accepts it. It returns `none` when the fuel is exhausted before the computation finishes, converting a possible non-termination into an explicit, total signal of failure. The fuel technique is the standard way to give a total, reasoned-about approximation of a function whose termination cannot be established.

3.22 Structural versus generative recursion

Two kinds of recursion have appeared, and naming the distinction clarifies when each is available. *Structural* recursion calls itself on a syntactic sub-part of its argument, a tail of a list, a predecessor of a number, and terminates automatically because the parts are finite. *Generative* recursion calls itself on a newly computed value, a halved list, a remainder, and needs an explicit measure to justify termination.

	Structural	**Generative**
recurses on	a sub-part of the input	a computed value
termination	automatic	needs a measure
examples	`map`, `foldr`, `length`	`mergeSort`, `gcd`, `log2`
proved by	structural induction	functional induction

Table 3.3: Structural versus generative recursion. The first descends into existing parts and terminates for free; the second recurses on freshly computed data and must exhibit a decreasing measure.

The two are not rivals but a spectrum, and Lean handles both, automatically for the structural case and with a `termination_by` clause for the generative one. Recognising which a definition uses tells one whether a termination proof will be needed and, later, which induction principle will prove facts about it: structural induction for the first, the generated functional induction for the second.

Remark 3.12. The distinction reaches back to the termination requirement of this chapter and forward to the induction principles of Chapter 7. A structural definition and structural induction are two uses of one recursor; a generative definition and its functional induction are two uses of another. In every case the shape of the recursion and the shape of the reasoning about it coincide, which is the single idea that unifies computation and proof in the language.

Exercises

Exercise 3.1. Write `cube : Nat → Nat` two ways: once with `def` and a named parameter, and once as an anonymous function assigned to a name with the dot notation `(· * · * ·)` suitably adapted. Check that both give `cube 3 = 27`.

Exercise 3.2. Define `max3 : Nat → Nat → Nat → Nat` returning the largest of three naturals, using `if...then...else` or nested matching. Evaluate it on `(4, 9, 7)` and `(9, 2, 5)`.

Exercise 3.3. Using pattern matching, define `isEmpty : List α → Bool` that returns `true` exactly on the empty list. Why does this function need no recursion?

Exercise 3.4. Trace the reduction of `fact 4` in the style of Figure 3.2, writing each intermediate expression until you reach the numeral.

Exercise 3.5. Define `sumTo : Nat → Nat` computing $0+1+\cdots+n$ by structural recursion. State the pattern that names the predecessor and identify the strictly decreasing argument.

Exercise 3.6. Define `pow : Nat → Nat → Nat` so that `pow b e` computes b^e by recursion on `e`. Evaluate `pow 2 10`.

Exercise 3.7. The function `applyTwice` applies its argument twice. Write `applyN : Nat → (Nat → Nat) → Nat → Nat` that applies a function a given number of times, and check that `applyN 3 (· + 1) 0 = 3`.

Exercise 3.8. Using `List.map` and `List.filter`, write a single expression that takes a list of naturals, keeps the even ones, and squares each survivor. Test it on `[1,2,3,4,5,6]`.

Exercise 3.9. Predict the values of `(sq ∘ inc) 5` and `(inc ∘ sq) 5`, then verify. Explain why the two disagree.

Exercise 3.10. ⋆ Define `ackermann : Nat → Nat → Nat` by the usual double recursion. Lean will not accept it with the default structural check; add a `termination_by` clause with a lexicographic measure on the pair of arguments and explain why that measure decreases.

Exercise 3.11. ⋆ A function `collatzStep : Nat → Nat` halves an even input and maps an odd n to $3n+1$. Iterating it is not known to terminate for all inputs. Write `collatzStep` as a total function, but explain why a function that iterates it to `1` must be marked `partial`.

Exercise 3.12. Define `eqColor` on the type `Color` of Chapter 4 using multiple patterns, returning `true` exactly when the two arguments match. How many rows does the wildcard save?

Exercise 3.13. Write `firstThree : List α → Option (α × α × α)` using a nested pattern three layers deep. Test it on a list of length two and of length four.

Exercise 3.14. Rewrite `sumTo` in accumulator-passing style as `sumToAux`, and trace both the plain and the accumulating versions on input `3` in the manner of Figure 3.3.

Exercise 3.15. Define `isEven` and `isOdd` mutually as in the text, and explain in one sentence why the combined recursion terminates.

Exercise 3.16. Express the product of a list of naturals first with `foldr` and then with `foldl`. Do the two agree on `[2, 3, 4]`? On what property of multiplication does the agreement rest?

Exercise 3.17. ⋆ Argue informally that `foldr (· + ·) 0` and `foldl (· + ·) 0` compute the same sum for every list of naturals, and identify the law of addition that the argument requires.

Exercise 3.18. Rewrite the nested expression

```
List.reverse (List.filter (· > 2) xs)
```

as a pipeline using `|>`. Which reading is clearer?

Exercise 3.19. Using the operations of Table 3.1, write a single pipeline that takes `List.range 10`, keeps the even numbers, and doubles each. Report the result.

Exercise 3.20. Define `log2` as in the text and evaluate it at `16` and `63`. State precisely what obligation `decreasing_by` discharges.

Exercise 3.21. Write a `do` block over `Option` that reads three elements of a list by index and returns their sum, failing if the list is too short. Test it on a list of length two and of length four.

Exercise 3.22. Define `safeHead` and `safeTail` returning `Option`, and chain them in a `do` block to extract the second element. Compare with a nested match.

Exercise 3.23. ⋆ Define `gcd : Nat → Nat → Nat` by the Euclidean algorithm, supplying a `termination_by` measure. Explain why the recursive argument decreases and how this parallels strong induction.

Exercise 3.24. Write a closure factory `multiplier (n : Nat) : Nat → Nat`, build two specialisations, and evaluate each. What value does each closure capture?

Exercise 3.25. Store three closures in a list and map application by a common argument over the list, as in the text. Report the result.

Exercise 3.26. Rewrite `sumList` using a `let rec` accumulator, and explain why the helper is kept local rather than made a top-level definition.

Exercise 3.27. Use each of `id`, `Function.const`, `flip`, and $\circ$ once, predicting the result of each before evaluating.

Exercise 3.28. Express the transformation "double, then subtract one" using composition of two functions rather than a single anonymous function.

Exercise 3.29. ⋆ Define `flip` yourself in terms of `fun`, and prove that `flip (flip f) = f` using `funext` twice.

Exercise 3.30. Rewrite `List.foldl (· + ·) 0` as a `for` loop with a `let mut` accumulator inside `Id.run do`, and confirm it agrees on a sample list.

Exercise 3.31. Write a loop that builds the list of squares of `1` through `n`, and compare it with the equivalent use of `List.map` over `List.range`.

Exercise 3.32. Define `collatz` as a `partial` function. Explain why it cannot be given a `termination_by` measure.

Exercise 3.33. Write a fueled version of a function whose termination is not obvious, returning `none` when the fuel runs out. Why is the fueled version total?

Exercise 3.34. Classify each of `List.map`, `mergeSort`, and `Nat.gcd` as structural or generative recursion, and state how you can tell.

Exercise 3.35. ⋆ Take a `partial` function of your own and convert it to a total function with a fuel argument returning `Option`, so that exhausting the fuel yields `none`.

Chapter 4

Inductive Types

The types met so far were given in advance: `Nat`, `Bool`, `List`. Each of them is in fact defined inside Lean by the same mechanism, available to any user, that builds a type from a finite list of ways to construct its elements. A type so defined is called *inductive*, and inductive types are the sole means by which data is introduced. Learning to read and write them is learning where all the data in a Lean program comes from.

4.1 A type is its constructors

An inductive declaration names a new type and lists its *constructors*: the basic expressions that produce elements. The simplest case is a finite enumeration, where each constructor takes no arguments and simply is an element.

```
inductive Color where
  | red
  | green
  | blue

#check Color.red        -- Color.red : Color
#check Color.green      -- Color.green : Color
```

The declaration asserts three things at once: that `Color` is a new type, that `red`, `green`, and `blue` are elements of it, and, crucially, that these are the *only* elements. Nothing outside the list inhabits the type. This closure is what makes a function defined by cases on `Color` total: three branches exhaust the possibilities.

```
def isWarm : Color → Bool
```

```
  | Color.red   => true
  | Color.green => false
  | Color.blue  => false

#eval isWarm Color.red      -- true
#eval isWarm Color.blue     -- false
```

Remark 4.1. The constructors live in a namespace named after the type, so the full name is `Color.red`. Inside a `match` the prefix may often be dropped, and opening the namespace with `open Color` makes the short names available everywhere. The dotted form is always safe.

4.2 Constructors that carry data

A constructor may take arguments, in which case it packages those arguments into an element. This is how a type comes to have infinitely many, or structured, inhabitants. The natural numbers are the founding example: a natural is either zero or the successor of another natural.

```
inductive Natural where
  | zero : Natural
  | succ : Natural → Natural
```

The constructor `succ` takes a `Natural` and returns a `Natural`, so elements pile up without bound: `zero`, `succ zero`, `succ (succ zero)`, and so on. There is no separate stock of numerals; the numeral 3 is notation for `succ (succ (succ zero))`, and Lean displays the long form back in that compact way.

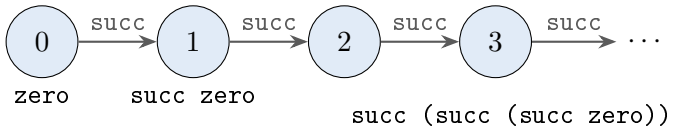


Figure 4.1: The natural numbers as an inductive type. A single base element `zero` and a single generating operation `succ` produce an unbounded chain; each numeral names a finite tower of successors.

Functions on such a type are written by matching the two shapes an element can have. Addition recurses on the second argument, peeling one `succ` at a time and rebuilding it around the result.

```
def add : Natural → Natural → Natural
  | m, Natural.zero   => m
  | m, Natural.succ n => Natural.succ (add m n)
```

The definition mirrors the shape of the type exactly: one branch per constructor of the argument being taken apart. This correspondence between the constructors of a type and the branches of a function is the heart of the inductive method.

4.3 The recursor

Every inductive declaration silently generates a companion, the *recursor*, which encapsulates its principle of definition by cases and recursion. For `Natural` the recursor expects a result for `zero` and a rule that produces the result for `succ n` given the result for `n`; from these it produces a result for every natural.

```
#check @Natural.rec
-- roughly: (motive : Natural → Sort u) →
--   motive zero →
--   ((n : Natural) → motive n → motive (succ n)) →
--   (t : Natural) → motive t
```

Pattern-matching definitions are compiled down to applications of this recursor, so nothing is lost by thinking in terms of `match`; the recursor is the primitive, and the convenient syntax is a faithful surface over it. The same object, read with a proposition in place of the `motive`, becomes the principle of proof by induction.

4.4 Polymorphic inductive types

An inductive type may take parameters, producing a family of types uniformly. Lists are the standard case: for each element type α there is a type `List` α, built from an empty list and a prepend operation.

```
inductive Lst (α : Type) where
  | nil  : Lst α
  | cons : α → Lst α → Lst α
```

The parameter α stands fixed throughout: `cons` takes a head of type α and a tail of type

`Lst` α and produces a `Lst` α. A concrete list is a chain of `cons` cells ending in `nil`, exactly the linked structure the name suggests.

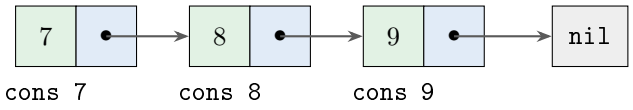


Figure 4.2: The list `[7, 8, 9]` as nested `cons` cells. Each cell holds a head and a pointer to the rest; the chain terminates at `nil`. The notation `[7, 8, 9]` abbreviates `cons 7 (cons 8 (cons 9 nil))`.

Operations on lists recurse down this chain. Concatenation walks the first list to its end and grafts the second on in place of `nil`.

```
def append : Lst α → Lst α → Lst α
  | Lst.nil,       ys => ys
  | Lst.cons x xs, ys => Lst.cons x (append xs ys)
```

Example 4.2 (Mapping over a list)**.** Applying a function to every element is again structural recursion: the empty list maps to itself, and a `cons` maps its head and recurses on its tail.

```
def lmap (f : α → β) : Lst α → Lst β
  | Lst.nil       => Lst.nil
  | Lst.cons x xs => Lst.cons (f x) (lmap f xs)
```

The shape of the recursion is forced by the shape of the type, which is why so many list functions look alike: they are all instances of the single recursor of `Lst`.

4.5 Branching data: trees

Nothing restricts a constructor to a single recursive argument. A binary tree has a node constructor with *two* subtrees, so elements branch rather than form a chain.

```
inductive Tree (α : Type) where
  | leaf : Tree α
  | node : Tree α → α → Tree α → Tree α
```

Counting the values stored in a tree recurses into both subtrees and adds one for the node itself. The two recursive calls correspond to the two recursive arguments of `node`, just as the single call for a list corresponded to the single recursive argument of `cons`.

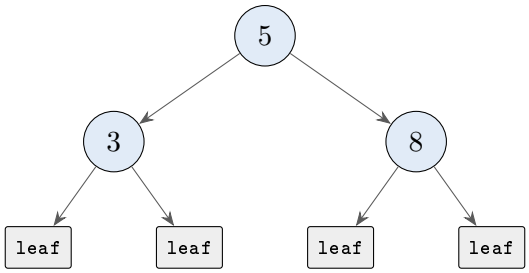


Figure 4.3: A binary tree with root value 5 and two interior nodes. Interior nodes carry a value and two subtrees; empty positions are `leaf`. Recursion on a tree makes two recursive calls, one per subtree.

```
def size : Tree α → Nat
  | Tree.leaf       => 0
  | Tree.node l _ r => size l + 1 + size r

def mirror : Tree α → Tree α
  | Tree.leaf       => Tree.leaf
  | Tree.node l x r => Tree.node (mirror r) x (mirror l)
```

The function `mirror` swaps left and right at every node, reflecting the whole tree; applying it twice restores the original, a fact provable once the language of proof is available.

4.6 Constructors with no data and none at all

Two extreme cases are worth naming. A type with a single constructor taking no arguments has exactly one element and carries no information; `Unit` is this type, and its lone element is written `()`. A type with *no* constructors at all has no elements whatsoever; it is called `Empty`, and the impossibility of producing an element of it is exactly what makes it useful for expressing that a situation cannot arise.

```
inductive Unit' where
  | unit

inductive Empty'      -- no constructors: no inhabitants
```

Remark 4.3. The spectrum runs from `Empty`, with zero elements, through `Unit`, with one, through `Bool`, with two, to `Nat`, with infinitely many. All four are ordinary inductive types differing only in their lists of constructors. The same uniform scheme accommodates the empty type and the natural numbers alike.

4.7 Structures as single-constructor types

When a type has exactly one constructor bundling several fields, Lean offers the keyword `structure` as a convenient front end. It generates the constructor together with named projections for the fields.

```
structure Point where
  x : Int
  y : Int

def origin : Point := { x := 0, y := 0 }
def p : Point := ⟨3, 4⟩

#eval p.x            -- 3
#eval p.y            -- 4
```

The anonymous-constructor brackets `⟨3, 4⟩` build a `Point` positionally, while the projection `p.x` reads a field by name. A structure is nothing more than an inductive type with one constructor, dressed up with projections; the record syntax is sugar over the same foundation.

4.8 A worked type: arithmetic expressions

Inductive types describe not only data but the syntax of small languages. An arithmetic expression is a number, or the sum of two expressions, or their product; that description is an inductive declaration with three constructors, one recursive constructor for each binary operation.

```
inductive Expr where
  | num : Nat → Expr
  | add : Expr → Expr → Expr
  | mul : Expr → Expr → Expr

def three_plus_four : Expr := Expr.add (Expr.num 3) (Expr.num 4)
```

An element of `Expr` is a tree: leaves are numbers, internal nodes are operations. Evaluating such a tree is structural recursion, one branch per constructor, computing the numeric value each node denotes.

```
def eval : Expr → Nat
  | Expr.num n   => n
  | Expr.add a b => eval a + eval b
  | Expr.mul a b => eval a * eval b

#eval eval three_plus_four                          -- 7
#eval eval (Expr.mul (Expr.add (Expr.num 2)
               (Expr.num 3)) (Expr.num 4))          -- 20
```

The second computation evaluates the expression $(2+3)\times 4$, whose tree branches at the multiplication into a sum and a leaf. The evaluator recurses into both children of each node and combines their values with the operation the node names.

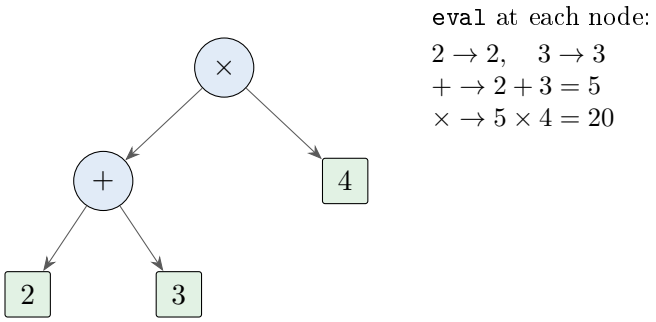


Figure 4.4: The expression $(2+3)\times 4$ as a value of `Expr`, and the bottom-up evaluation the recursor performs. Leaves return their numbers; each operation combines the values of its two subexpressions.

This pattern, an inductive type of syntax paired with a recursive interpreter, recurs whenever a language is formalised. The same shape scales from this three-constructor arithmetic to the abstract syntax of whole programming languages, and the evaluator scales with it.

4.9 Indexed families

A parameter, like the element type of `Lst`, stays fixed across a declaration. An *index* may instead vary from constructor to constructor, so that a single declaration produces a family of types distinguished by a value. Length-indexed lists, or *vectors*, illustrate this: the type records the length, and the constructors adjust it.

```
inductive Vec (α : Type) : Nat → Type where
  | nil  : Vec α 0
  | cons : α → Vec α n → Vec α (n + 1)

```

```
def v3 : Vec Nat 3 :=
  Vec.cons 1 (Vec.cons 2 (Vec.cons 3 Vec.nil))
```

The type `Vec` α `n` is inhabited only by vectors of exactly n elements: `nil` has length 0, and `cons` produces a vector one longer than its tail. The length is not stored and checked at run time; it is part of the type, verified once when the term is elaborated. A function may then demand a vector of a particular length, and the checker enforces the demand.

```
def head : Vec α (n + 1) → α
  | Vec.cons x _ => x

#check head v3        -- head v3 : Nat,   accepted: v3 has length 3
-- #check head Vec.nil  -- rejected: nil has length 0, not n+1
```

The function `head` accepts only vectors whose length is a successor, so applying it to the empty vector is rejected before the program runs. There is no missing case and no possibility of taking the head of nothing; the type has excluded it. This is the first taste of what dependent types buy: invariants enforced by the type checker rather than by run-time guards.

4.10 Inductive predicates

The same mechanism defines relations. A proposition indexed by a value is an inductive family living in `Prop`, and its constructors are the ways the relation can hold. Evenness is the standard example: zero is even, and adding two to an even number keeps it even.

```
inductive Ev : Nat → Prop where
  | zero            : Ev 0
  | add_two (n : Nat) : Ev n → Ev (n + 2)

theorem four_even : Ev 4 :=
  Ev.add_two 2 (Ev.add_two 0 Ev.zero)
```

A proof of `Ev 4` is a chain of constructors: start from `zero` and apply `add_two` twice, once to reach 2 and once to reach 4. The proof is data of type `Ev 4`, built exactly as an element of any inductive type is built. Defining a relation as the smallest one closed under given rules is precisely defining an inductive family, and the two notions coincide.

Remark 4.4. An inductive predicate captures the least relation closed under its rules. Noth-

ing is even except what `zero` and `add_two` force to be, so `Ev 3` has no proof, and its absence is provable. This "no other way in" is the same closure that made a function on a finite enumeration total, now doing logical rather than computational work.

4.11 Deriving standard operations

Many types need routine operations: a way to display a value, a test for equality, an ordering. Writing these by hand is mechanical, so Lean will generate them on request with a `deriving` clause. The generated code is ordinary and could have been written out, but deriving spares the effort and the chance of error.

```
inductive Suit where
  | hearts | diamonds | clubs | spades
  deriving Repr, DecidableEq, BEq

#eval Suit.hearts                    -- Suit.hearts   (via Repr)
#eval Suit.hearts == Suit.spades     -- false          (via BEq)
#eval decide (Suit.clubs = Suit.clubs) -- true         (via DecidableEq)
```

`Repr` supplies a printable form, so `#eval` can display a value; `BEq` supplies a boolean equality test; `DecidableEq` supplies a proof-producing decision procedure for equality, which is what lets `decide` settle equations between elements. Each is generated from the list of constructors, and each becomes available to every function and tactic that needs it.

Example 4.5 (Equality that computes)**.** A derived `DecidableEq` does more than return a boolean: it returns a proof, either of equality or of its negation. This is what allows a match on whether two values are equal to produce evidence usable in a later proof.

```
def sameSuit (a b : Suit) : Bool :=
  if a = b then true else false

#eval sameSuit Suit.hearts Suit.hearts    -- true
#eval sameSuit Suit.hearts Suit.clubs     -- false
```

The condition `a = b` is a proposition, and the `if` can branch on it only because `DecidableEq` makes it decidable. Without the derived instance the same `if` would not type-check, since Lean would not know how to evaluate the test.

4.12 Mutually inductive types

Two types may be defined together, each mentioning the other. A `mutual` block groups the declarations so that a constructor of one may take an argument of the other. A tree whose nodes have arbitrarily many children is naturally described this way, as a tree paired with a type of forests, a forest being a list of trees.

```
mutual
  inductive Tree' where
    | node : Nat → Forest → Tree'
  inductive Forest where
    | empty : Forest
    | cons  : Tree' → Forest → Forest
end
```

A `Tree'` is a number together with a `Forest` of subtrees, and a `Forest` is either empty or a tree followed by more forest. The two refer to each other, so neither can be declared alone. Functions over such types are likewise mutually recursive, each calling the other on smaller arguments.

```
mutual
  def treeSum : Tree' → Nat
    | Tree'.node n f => n + forestSum f
  def forestSum : Forest → Nat
    | Forest.empty    => 0
    | Forest.cons t f => treeSum t + forestSum f
end
```

The summation of a tree adds its label to the sum of its forest of subtrees; the summation of a forest adds each tree in turn. The mutual recursion terminates because every call is on a structurally smaller piece, a subtree or a shorter forest, so the combined descent is well founded exactly as a single recursion would be.

4.13 Structures in depth

A structure may give default values for its fields, so that constructing one requires supplying only the fields that differ from the defaults. It also supports an update syntax, building a new structure from an existing one by changing named fields. Together these make records of configuration convenient.

```
structure Config where
  width  : Nat := 80
  height : Nat := 24
  title  : String := "untitled"

def defaultConfig : Config := {}
def wide : Config := { defaultConfig with width := 120 }

#eval wide.width       -- 120
#eval wide.height      -- 24,  inherited unchanged
#eval wide.title       -- "untitled"
```

The empty braces `{}` build a `Config` entirely from defaults. The update `{ defaultConfig with width := 120 }` copies `defaultConfig`, overriding only `width`, and leaves the other fields as they were. This is how a small change to a large record is expressed without restating every field, and it reads as the modification it performs.

Remark 4.6. A structure remains a single-constructor inductive type underneath, so everything said of inductive types applies. Defaults and update syntax are conveniences the `structure` keyword provides over the raw constructor; the anonymous-constructor brackets and field projections of Chapter 4 still work, and a structure may be taken apart by `match` like any other inductive value.

4.14 A worked type: propositional formulas

Inductive types capture the syntax of logic as readily as that of arithmetic. A propositional formula is a variable, a constant, a conjunction, a disjunction, or a negation, an inductive type with one constructor per form. Variables are named by strings.

```
inductive Form where
  | var : String → Form
  | tt  : Form
  | ff  : Form
  | and : Form → Form → Form
  | or  : Form → Form → Form
  | neg : Form → Form
```

Evaluating a formula requires an *environment* assigning a boolean to each variable. Given one, evaluation is structural recursion: a variable looks itself up, the constants are fixed, and each connective combines the values of its parts with the matching boolean operation.

```
def eval (env : String → Bool) : Form → Bool
  | Form.var x   => env x
  | Form.tt      => true
  | Form.ff      => false
  | Form.and a b => eval env a && eval env b
  | Form.or a b  => eval env a || eval env b
  | Form.neg a   => not (eval env a)
```

The environment is itself a function `String → Bool`, so supplying one is supplying a truth assignment. A concrete formula and a concrete assignment evaluate to a boolean.

```
def sample : Form :=
  Form.and (Form.var "p") (Form.neg (Form.var "q"))

def env1 : String → Bool := fun x => x == "p"

#eval eval env1 sample      -- true:  p is true, q is false
```

The formula $p \wedge \neg q$ evaluates to `true` under an assignment making `p` true and everything else false. This pairing of an inductive syntax with a recursive semantics is the pattern behind every interpreter, and building on it, one could define substitution, test for tautology, or convert to a normal form, each a further recursion over `Form`.

Example 4.7 (Counting connectives)**.** A different recursion measures a formula rather than evaluating it. Counting the logical connectives ignores variables and constants and adds one at each `and`, `or`, or `neg`.

```
def connectives : Form → Nat
  | Form.var _   => 0
  | Form.tt      => 0
  | Form.ff      => 0
  | Form.and a b => 1 + connectives a + connectives b
  | Form.or a b  => 1 + connectives a + connectives b
  | Form.neg a   => 1 + connectives a

#eval connectives sample      -- 2
```

The same type supports many recursions, one per question asked of it. Evaluation and counting share the shape dictated by `Form` and differ only in what each branch returns.

4.15 Why constructors must be positive

Not every inductive declaration is accepted. A constructor may take arguments involving the type being defined, but only in *positive* positions, never to the left of an arrow within an argument. This restriction, *strict positivity*, is what keeps the theory consistent, and a declaration that violates it is rejected outright.

```
-- Rejected: Bad occurs to the left of an inner arrow.
-- inductive Bad where
--   | mk : (Bad → Bad) → Bad

-- Accepted: Good occurs only to the right of the inner arrow.
inductive Good where
  | mk : (Nat → Good) → Good
```

In `Good` the type appears only in a positive position, to the right of the arrow in `Nat → Good`, describing an infinitely branching tree, and Lean accepts it. In `Bad` the type appears to the left of an inner arrow, a negative position, and Lean rejects it. The reason is not arbitrary: a negative occurrence would permit encoding a term that reduces to itself without end, and from such a term a proof of `False` could be built, collapsing the distinction between true and false.

Remark 4.8. The positivity condition is the inductive counterpart of the termination requirement on recursion. Both forbid exactly the self-reference that would break normalisation, one in the shape of a type, the other in the shape of a function. A language admitting either would be inconsistent as a logic, so the checker enforces both, and the price is that a handful of self-applicative definitions are unavailable.

4.16 Parameters versus indices

An inductive type may depend on other data in two distinct ways, and the difference governs what its constructors may do. A *parameter* is fixed across the whole declaration and appears identically in every constructor's result; an *index* may vary from constructor to constructor. The element type of a list is a parameter; the length of a vector is an index.

```
inductive Lst (α : Type) where          -- α is a parameter
  | nil  : Lst α
  | cons : α → Lst α → Lst α

```

```
inductive Vec (α : Type) : Nat → Type where  -- Nat is an index
  | nil  : Vec α 0
  | cons : α → Vec α n → Vec α (n + 1)
```

The parameter α stands to the left of the colon and is the same in `nil` and `cons`; neither constructor may change it. The index `Nat` stands to the right of the colon, and the constructors do change it: `nil` builds a vector at index `0`, `cons` one at a successor. This is precisely why the length can be tracked in the type, and why the empty list's element type cannot vary: one is an index, free to move, the other a parameter, held fixed.

	Parameter	**Index**
written	before the colon	after the colon
across constructors	identical	may vary
example	α in `Lst` α	`Nat` in `Vec` α `n`
role	uniform data	tracked invariant

Table 4.1: Parameters versus indices in an inductive declaration. A parameter is fixed throughout; an index may differ per constructor, which is what lets the type record a varying quantity.

4.17 A well-typed interpreter

Indices earn their keep when they enforce an invariant the type checker then guarantees. A language of expressions can be indexed by the type of value each expression produces, so that only well-typed expressions can be built and an interpreter for them cannot go wrong. First, the object language's types and their meanings.

```
inductive Ty where
  | nat | bool

def interp : Ty → Type
  | Ty.nat  => Nat
  | Ty.bool => Bool
```

The function `interp` sends each object type to the Lean type of its values. Expressions are then indexed by their object type, and each constructor's index records the type it yields: a numeric literal has type `nat`, a comparison `bool`, and a conditional whatever type its branches share.

```
inductive Exp : Ty → Type where
  | lit  : Nat → Exp Ty.nat
  | bLit : Bool → Exp Ty.bool
  | add  : Exp Ty.nat → Exp Ty.nat → Exp Ty.nat
  | ite  : Exp Ty.bool → Exp α → Exp α → Exp α
```

The constructor `add` accepts only numeric subexpressions, and `ite` requires its condition to be boolean and its two branches to agree in type. An expression adding a boolean to a number simply cannot be formed; the mistake is a type error at construction, not a fault discovered when running. The interpreter can therefore be total, returning a value of exactly the type the expression's index names.

```
def evalE : Exp t → interp t
  | Exp.lit n    => n
  | Exp.bLit b   => b
  | Exp.add a b  => evalE a + evalE b
  | Exp.ite c x y => if evalE c then evalE x else evalE y

#eval evalE (Exp.add (Exp.lit 2) (Exp.lit 3))         -- 5
#eval evalE (Exp.ite (Exp.bLit true)
               (Exp.lit 1) (Exp.lit 0))               -- 1
```

The result type `interp t` depends on the expression's index, so `evalE` of a numeric expression is a `Nat` and of a boolean one a `Bool`, each computed without any run-time type check or possibility of failure. The index has carried the object language's typing discipline into Lean's own types, and correctness of the interpreter is guaranteed by construction rather than established after the fact.

Remark 4.9. This is the characteristic power of indexed types: an invariant expressed as an index becomes impossible to violate, because a term breaking it cannot be written. The length of a vector, the type of an expression, the balance of a tree, each can be made an index, and the type checker then enforces it everywhere for free. The cost is the greater care indexed definitions demand; the reward is a class of errors that ceases to exist.

4.18 Constructors are injective and distinct

Two facts about constructors hold for every inductive type and are generated automatically. Constructors are *injective*: if two applications of the same constructor are equal, their arguments are equal. And distinct constructors are *disjoint*: a value built one way never equals

one built another. These are exactly what make pattern matching sound, and they are available as lemmas.

```
example (m n : Nat) (h : Nat.succ m = Nat.succ n) : m = n :=
  Nat.succ.inj h

example (n : Nat) (h : Nat.succ n = 0) : False :=
  Nat.noConfusion h
```

The first uses injectivity: from `succ m = succ n` it extracts `m = n`. The second uses disjointness through `noConfusion`: a successor can never equal zero, so a proof of `succ n = 0` yields a proof of anything. Both facts come free with the declaration of `Nat`, and the same holds for the constructors of any inductive type. The simplifier knows them as well, so `simp` often discharges goals that reduce to constructor injectivity or disjointness.

```
example (a b : Nat) : (a :: [b]) = (a :: [b]) ↔ True := by simp
example (n : Nat) : Nat.succ n ≠ 0 := by simp
```

Remark 4.10. Injectivity and disjointness are what a case analysis relies on. When a proof matches on a natural and treats `zero` and `succ` separately, it uses that these are the only two possibilities and that they cannot overlap; when it peels a `succ` to expose the predecessor, it uses injectivity. The properties feel obvious precisely because they are built into how inductive types work, but they are theorems, generated and available by name.

4.19 An order defined inductively

Relations, like data, are defined by giving the ways they can hold. The order relation on the naturals is an inductive family: a number is at most itself, and if it is at most `m` then it is at most `m + 1`. A proof of `n ≤ m` is a chain of steps climbing from `n` to `m`.

```
inductive Le (n : Nat) : Nat → Prop where
  | refl : Le n n
  | step : Le n m → Le n (m + 1)

theorem le_two : Le 0 2 := Le.step (Le.step Le.refl)
```

The proof `Le.step (Le.step Le.refl)` builds a witness that `0 ≤ 2`: begin at `0 ≤ 0` with `refl`, and apply `step` twice to climb to `2`. This is the definition the standard library

uses for `Nat.le`, and facts about the order are proved by induction on such a chain, exactly as facts about numbers are proved by induction on their successors.

```
theorem le_trans {a b c : Nat}
    (h1 : Le a b) (h2 : Le b c) : Le a c := by
  induction h2 with
  | refl      => exact h1
  | step _ ih => exact Le.step ih
```

Transitivity is an induction on the second chain: if `b` $\leq$ `c` is just reflexivity, the first proof already gives `a` $\leq$ `c`; if it ends in a step, the inductive hypothesis handles the shorter chain and one more step finishes. Defining an order this way, as the least relation closed under reflexivity and successor, makes its proofs inductions over evidence, the technique of Chapter 7 applied to a relation.

4.20 Nested inductive types

A constructor may take, as an argument, a *collection* of the type being defined, not merely finitely many direct occurrences. A rose tree, whose node holds a value and a list of subtrees, is the standard example: `Rose` appears nested inside `List` in its own definition.

```
inductive Rose (α : Type) where
  | node : α → List (Rose α) → Rose α

def leaf (a : α) : Rose α := Rose.node a []
def small : Rose Nat :=
  Rose.node 1 [leaf 2, leaf 3, Rose.node 4 [leaf 5]]
```

The nesting, `List (Rose α)`, is what allows a node to have any number of children, unlike the fixed two of a binary tree. Recursion over a rose tree is subtler than over a binary one, because the recursive occurrences are buried inside a list and must be reached through a list operation. The size of a rose tree sums the sizes of its children.

```
def Rose.size : Rose α → Nat
  | Rose.node _ children =>
      1 + (children.map Rose.size).foldr (· + ·) 0

#eval small.size      -- 5
```

The recursive call `Rose.size` is applied to each child through `map`, and the results are summed. Lean accepts this because the children are structurally smaller than the whole, even though they are reached via the list; the termination checker sees through the nesting. Nested inductive types are the natural representation for tree-shaped data of unbounded branching, and they arise whenever a structure contains a collection of substructures of the same kind.

Remark 4.11. The nesting must still respect strict positivity: `Rose` occurs inside `List`, which uses its argument positively, so the definition is legitimate. Were the nesting through a type that used its argument negatively, the declaration would be rejected for the same reason a direct negative occurrence is. Positivity is checked through the nesting, not only at the surface, so a nested type is safe exactly when every layer of the nesting is.

Exercises

Exercise 4.1. Define an inductive type `Direction` with four constructors `north`, `east`, `south`, `west`. Write `turnRight : Direction → Direction` cycling through them, and evaluate it on each.

Exercise 4.2. Using the type `Natural` of this chapter, write out the element denoted by the numeral 4 as an explicit tower of `succ` applied to `zero`.

Exercise 4.3. Define `mul : Natural → Natural → Natural` by recursion on the second argument, using the `add` of the text. Explain which constructor each branch matches.

Exercise 4.4. For the list type `Lst`, define `length : Lst` $\alpha \to$ `Nat` by structural recursion. Identify the recursive argument and the base case.

Exercise 4.5. Draw, in the style of Figure 4.2, the cell diagram for the list `[1, 2]`, and write the same list as a nested `cons` expression ending in `nil`.

Exercise 4.6. Define `reverse : Lst` $\alpha \to$ `Lst` α using `append`. Trace its action on `[1, 2, 3]` to confirm the order is reversed.

Exercise 4.7. For the tree type, define `depth : Tree` $\alpha \to$ `Nat` returning the length of the longest root-to-leaf path. How does the two-way branching of the type show up in your definition?

Exercise 4.8. Draw the tree denoted by `Tree.node Tree.leaf 1 (Tree.node Tree.leaf 2 Tree.leaf)` and compute its `size` and `depth` by hand.

Exercise 4.9. Define a structure `Rectangle` with fields `width` and `height` of type `Nat`, and a function `area : Rectangle → Nat`. Construct a 3×4 rectangle two ways, with record syntax and with anonymous-constructor brackets.

Exercise 4.10. Explain why a function `Empty` $\to \alpha$ exists for every type α, whereas a function $\alpha \to$ `Empty` generally does not. What does the first function have to do in each branch?

Exercise 4.11. ⋆ The recursor `Natural.rec` takes a result for `zero` and a step turning a result for `n` into one for `succ n`. Using only `Natural.rec` (no `match`), define the function that doubles a natural, and describe the two arguments you supply.

Exercise 4.12. ⋆ Define an inductive type `Rose` of trees whose nodes carry a value and a `List` of children, rather than exactly two subtrees. Write `size : Rose` $\alpha \to$ `Nat`, and explain why counting a rose tree requires recursion over a list of subtrees.

Exercise 4.13. Extend `Expr` with a constructor `sub` for subtraction, add the matching branch to `eval`, and evaluate the expression $10 - (2 + 3)$.

Exercise 4.14. Draw the tree, in the style of Figure 4.4, for the expression $2 \times (3 + 4)$, and carry out the bottom-up evaluation at each node.

Exercise 4.15. Define `append : Vec` α `m → Vec` α `n → Vec` α `(m + n)` for vectors, and explain how the length index of the result is determined by the two inputs.

Exercise 4.16. Prove `Ev 6` by writing an explicit chain of the constructors `zero` and `add_two`. How many applications of `add_two` does it take?

Exercise 4.17. Add `deriving Repr` to an inductive type of your own and confirm that `#eval` can now display its values. What changes if you also derive `BEq`?

Exercise 4.18. ⋆ Define `vmap : (`$\alpha \to \beta$`) → Vec` α `n → Vec` β `n`, preserving the length index. Explain why the type guarantees the output has the same length as the input without any run-time check.

Exercise 4.19. Using the mutual types `Tree'` and `Forest`, define mutually recursive functions counting the number of nodes in a tree and in a forest. Test them on a small tree.

Exercise 4.20. Using structure defaults and update syntax, define a `Config` entirely from defaults and then a variant that changes only the `height`.

Exercise 4.21. Extend `Form` with a constructor `imp` for implication, and add the matching branch to `eval` using the fact that $p \to q$ is $\neg p \vee q$.

Exercise 4.22. Evaluate the formula `sample` under an environment making both `p` and `q` true. What value results, and why?

Exercise 4.23. Define `vars : Form → Nat` counting the variable occurrences in a formula, by recursion mirroring `connectives`.

Exercise 4.24. ⋆ Define `subst : String → Form → Form → Form` that replaces every occurrence of a given variable in a formula with another formula. Which constructor is the base case where the replacement happens?

Exercise 4.25. Explain why the declaration `inductive Bad | mk : (Bad → Bad) → Bad` is rejected. In which position does `Bad` occur, and why is that forbidden?

Exercise 4.26. Give a constructor in which the type being defined occurs only to the right of an arrow. What kind of structure does the resulting type describe?

Exercise 4.27. Classify the α in `Lst α` and the `Nat` in `Vec α n` as parameter or index, and state the rule distinguishing them.

Exercise 4.28. Extend `Exp` with a constructor `le : Exp Ty.nat → Exp Ty.nat → Exp Ty.bool`, and add the matching branch to `evalE`. What type does its result have?

Exercise 4.29. Attempt to build an expression adding a boolean literal to a numeric literal, and report the error. Why can this mistake never reach the interpreter?

Exercise 4.30. ⋆ Add a constructor `eq : Exp α → Exp α → Exp Ty.bool` testing two subexpressions of the same type for equality. What must `evalE` assume about `interp α` for its branch to type-check?

Exercise 4.31. Using `Nat.succ.inj`, prove that `Nat.succ m = Nat.succ n` implies `m = n`. What property of constructors is this?

Exercise 4.32. Using `Nat.noConfusion`, prove `False` from a hypothesis `Nat.succ n = 0`. Why can a successor never equal zero?

Exercise 4.33. Prove `Le 0 3` by an explicit chain of `Le.step` applications ending in `Le.refl`. How many steps does it take?

Exercise 4.34. Prove that `Le` is reflexive, and prove the particular instance that `Le a b` and `Le b c` give `Le a c` for concrete small values.

Exercise 4.35. Define `Rose` and construct a rose tree with a node of three children, one of which has children of its own. Compute its `size`.

Exercise 4.36. ⋆ Define `Rose.depth` returning one more than the maximum depth among the children, and explain how the recursion reaches the subtrees nested inside the list.

Chapter 5

Propositions as Types

The move that turns a programming language into a proof assistant is a single identification: a proposition is a type, and a proof of it is an element of that type. To prove a statement is to construct an inhabitant of the corresponding type, and the type checker that verifies programs thereby verifies proofs. This correspondence, between logic and computation, is the organizing principle of the whole system, and every logical connective turns out to be a type former already familiar from the study of data.

5.1 Propositions are types

A statement in Lean has type `Prop`. An element of a statement is a proof of it. The two readings sit side by side: `p : Prop` says p is a proposition, and `h : p` says h is a proof of p.

```
#check 2 + 2 = 4         -- 2 + 2 = 4 : Prop
#check 2 + 2 = 5         -- 2 + 2 = 5 : Prop  (a proposition, just false)

theorem two_plus_two : 2 + 2 = 4 := rfl
#check two_plus_two      -- two_plus_two : 2 + 2 = 4
```

Both equations are perfectly good propositions; being false is not the same as being ill-formed. What distinguishes a true proposition is that its type is inhabited: a proof exists. Here `rfl`, the proof that a thing equals itself, inhabits `2 + 2 = 4` because both sides reduce to `4`. No inhabitant of `2 + 2 = 5` can be built, which is precisely what its falsity means.

Definition 5.1 (Propositions as types)**.** Under the correspondence, each proposition P is read as a type $[\![P]\!]$ whose elements are the proofs of P. The proposition P holds exactly

when $[\![P]\!]$ is inhabited. Proving P and constructing a term of $[\![P]\!]$ are one and the same activity.

The keyword `theorem` is, at the level of the kernel, identical to `def`: both bind a name to a term of a stated type. The two words differ only in intent and in Lean's treatment of the result as irrelevant data. A proof is a program whose type happens to be a proposition.

5.2 Implication is the function arrow

To prove "if P then Q" is to give a method transforming any proof of P into a proof of Q. That is exactly a function from $[\![P]\!]$ to $[\![Q]\!]$. Implication and the function arrow are the same connective.

```
theorem imp_example (P Q : Prop) : P → (Q → P) :=
  fun hp => fun _ => hp
```

The statement $P \to Q \to P$ asks for a proof of P given a proof of P and a proof of Q. The inhabitant is the function that takes the first proof `hp`, ignores the second, and returns `hp`: the same constant-function term that would inhabit the analogous data type. Reading the arrow as implication and reading it as a function type give identical proofs.

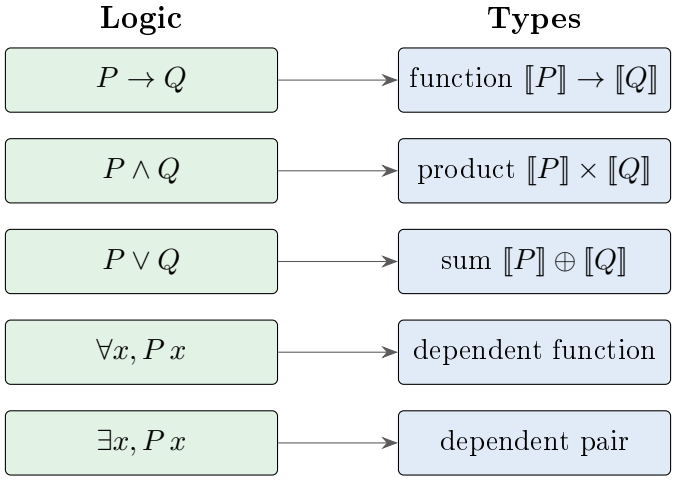


Figure 5.1: The correspondence between logical connectives and type formers. Each row is one identification: proving the statement on the left is constructing an element of the type on the right.

Figure 5.1 lists the whole dictionary. The remaining sections work through it row by row, and in each case the logical rule turns out to be a construction on types that has already appeared in the study of data.

5.3 Conjunction is the product

A proof of $P \wedge Q$ is a proof of P together with a proof of Q: a pair. Lean writes the conjunction `And` and pairs its proofs with the anonymous-constructor brackets, exactly as for a product of data types.

```
theorem and_intro (P Q : Prop) (hp : P) (hq : Q) : P ∧ Q :=
  ⟨hp, hq⟩

theorem and_swap (P Q : Prop) (h : P ∧ Q) : Q ∧ P :=
  ⟨h.right, h.left⟩
```

Building a proof of a conjunction pairs the two component proofs; using one projects out a component with `.left` or `.right`. The connective behaves in every respect like the product type of Chapter 4, because it is one: `And` is a structure with two fields.

5.4 Disjunction is the sum

A proof of $P \vee Q$ is a proof of one side, tagged with which side it is. This is the sum type: two constructors, `Or.inl` for a proof of the left and `Or.inr` for a proof of the right.

```
theorem or_intro_left (P Q : Prop) (hp : P) : P ∨ Q :=
  Or.inl hp

theorem or_swap (P Q : Prop) (h : P ∨ Q) : Q ∨ P :=
  match h with
  | Or.inl hp => Or.inr hp
  | Or.inr hq => Or.inl hq
```

Introducing a disjunction chooses a side; using one matches on which side was chosen and handles each. The asymmetry with conjunction is exactly the asymmetry between sum and product: a product bundles both, a sum commits to one, and the proofs of $\wedge$ and $\vee$ inherit that difference wholesale.

5.5 True, False, and negation

The constant proposition `True` holds trivially; it is a type with a single proof `True.intro`, the analogue of `Unit`. The constant proposition `False` has no proof at all; it is the analogue

of `Empty`. From a proof of `False` anything follows, because a function out of the empty type may return a value of any type without ever having to produce one.

```
theorem trivially_true : True := True.intro

theorem from_false (P : Prop) (h : False) : P :=
  False.elim h
```

Negation is defined from these: $\neg P$ is shorthand for $P \to$ `False`. To prove $\neg P$ is to show that a proof of P would yield a proof of the impossible.

```
theorem not_false_proof : ¬ False :=
  fun h => h

theorem contra (P : Prop) (hp : P) (hnp : ¬ P) : False :=
  hnp hp
```

The last proof simply applies the function `hnp : P → False` to the proof `hp : P`. Contradiction, in this reading, is nothing more exotic than function application landing in the empty type.

Remark 5.2. This treatment is constructive: `False.elim` builds a proof of P from a proof of `False`, but there is no general principle producing a proof of $P \vee \neg P$ out of nothing. The law of the excluded middle is available in Lean as a classical axiom when wanted, but it is not forced by the correspondence itself.

5.6 Quantifiers

The universal quantifier is the dependent function type of Chapter 2. A proof of $\forall x, P\,x$ is a method that, given any x, produces a proof of $P\,x$: a function whose result type depends on its argument.

```
theorem all_refl : ∀ n : Nat, n = n :=
  fun n => rfl

theorem all_imp (P Q : Nat → Prop) (h : ∀ n, P n → Q n)
    (hp : ∀ n, P n) : ∀ n, Q n :=
  fun n => h n (hp n)
```

Proving a universal statement is writing a function of the bound variable; using one is applying that function to a particular value. The quantifier $\forall n, n = n$ is inhabited by `fun n => rfl`, the function returning the reflexivity proof at each n.

The existential quantifier is the dependent pair. A proof of $\exists x, P\,x$ is a witness a together with a proof of $P\,a$, packaged with the same brackets as a conjunction but with the second component depending on the first.

```
theorem exists_even : ∃ n : Nat, n + n = 6 :=
  ⟨3, rfl⟩

theorem exists_imp (P : Nat → Prop) (h : ∃ n, P n)
    (Q : Prop) (k : ∀ n, P n → Q) : Q :=
  match h with
  | ⟨n, hn⟩ => k n hn
```

Introducing an existential supplies a witness and a proof about it; using one matches to extract both. The proof `⟨3, rfl⟩` names 3 as the witness and offers `rfl` as evidence that $3 + 3 = 6$.

5.7 Reading a proof as a derivation

Because a proof is a term built by applying constructors and functions, it can be displayed as a derivation tree in the notation of natural deduction. The proof of $P \to Q \to P$ corresponds to two nested introductions of an implication.

$$\dfrac{\dfrac{[\,hp : P\,]^{2}}{Q \to P}\,{\to}\mathrm{I}^{1}\ (\text{discharge } hq{:}Q)}{P \to (Q \to P)}\,{\to}\mathrm{I}^{2}\ (\text{discharge } hp{:}P)$$

Each introduction of an arrow discharges an assumption, matching a `fun` in the term; each application of a hypothesis matches a function application. The tree and the term are two renderings of one object, and Lean checks the term with the same certainty that a reader checks the tree.

Example 5.3 (A conjunction rearranged)**.** The proof that $P \wedge Q$ implies $Q \wedge P$ pairs the two projections in the opposite order. As a derivation it splits the assumption into its halves and reassembles them.

$$\dfrac{\dfrac{h : P \wedge Q}{h.\mathsf{right} : Q} \qquad \dfrac{h : P \wedge Q}{h.\mathsf{left} : P}}{\langle h.\mathsf{right}, h.\mathsf{left}\rangle : Q \wedge P}$$

The two premises are the two uses of the single hypothesis h, and the conclusion pairs them. The term ⟨`h.right, h.left`⟩ is this tree written on one line.

5.8 The biconditional

To say that two propositions are equivalent is to give implications both ways. The biconditional $P \leftrightarrow Q$ is therefore a pair of functions, one from P to Q and one back, and it is built and used exactly like a conjunction whose conjuncts are implications.

```
theorem iff_example (P : Prop) : P ↔ P :=
  ⟨fun hp => hp, fun hp => hp⟩

theorem and_comm_iff (P Q : Prop) : (P ∧ Q) ↔ (Q ∧ P) :=
  ⟨fun h => ⟨h.right, h.left⟩, fun h => ⟨h.right, h.left⟩⟩
```

The proof of an equivalence is a pair ⟨`forward, backward`⟩, and the two directions are recovered with `.mp` (the forward implication) and `.mpr` (the reverse). An equivalence is thus no new connective but a conjunction of implications, and everything known about pairs and functions applies to it directly.

5.9 Why proofs may be erased

Propositions live in `Prop`, data in `Type`, and the separation has a concrete consequence: two proofs of the same proposition are treated as identical, whereas two elements of a data type need not be. This principle, *proof irrelevance*, means the particular proof of a fact can never influence a computed value, and so proofs may be discarded before a program runs.

```
theorem proofs_equal (P : Prop) (h1 h2 : P) : h1 = h2 :=
  rfl
```

The equation `h1 = h2` holds by `rfl` for any two proofs of the same `P`, an equation that would be false for two elements of a typical data type. Because a proof carries no distinguishing information, a function may take a proof as an argument, use it to satisfy the type checker, and compile to code that never inspects it. The proof is a compile-time certificate, not a run-time value.

Remark 5.4. The distinction is what makes a dependently typed language usable for programming as well as proof. Correctness proofs, however elaborate, impose no run-time cost:

they justify a program to the checker and then vanish. A vector's length index and a sortedness certificate are both erased, leaving ordinary code behind.

5.10 Decidable propositions

Some propositions can be settled by computation. That a proposition P is *decidable* is itself a piece of data: an element of `Decidable P`, which is either a proof of P or a proof of its negation. Where such an element exists, a program may branch on the truth of P and a tactic may resolve it by evaluation.

```
#check (inferInstance : Decidable (2 < 5))    -- an instance exists

example : 2 < 5 := by decide
example : ¬ (5 < 2) := by decide
```

The tactic `decide` works by computing the `Decidable` instance: it evaluates the decision procedure, and if the answer is the positive one it extracts the proof. Decidability is the bridge between the computational and the logical readings of the system, letting a finite check stand in for a proof whenever one is available.

```
def isSmall (n : Nat) : Bool := n < 10

example : isSmall 3 = true := rfl
example : (if 3 < 10 then "yes" else "no") = "yes" := rfl
```

The conditional in the last line branches on `3 < 10`, a proposition, and computes because that proposition is decidable. The boolean `isSmall` and the propositional `3 < 10` are linked by their shared decision procedure, and moving between the two is routine.

5.11 Classical reasoning

The correspondence is constructive: a proof must build a witness, and the law of the excluded middle, $P \vee \neg P$, has no uniform constructive proof, since no procedure decides an arbitrary proposition. Lean nonetheless makes classical reasoning available as an axiom, for the substantial parts of mathematics that use it. Importing it supplies `em` and the tactic `by_contra`.

```
open Classical

theorem dne (P : Prop) (h : ¬ ¬ P) : P := by
  by_contra hnp
  exact h hnp

example (P : Prop) : P ∨ ¬ P := em P
```

Double-negation elimination, unprovable constructively, follows at once from `by_contra`, which assumes the negation of the goal and derives a contradiction. The axiom `em` directly provides $P \vee \neg P$ for every P. Adopting these does not make the system inconsistent; it enlarges what can be proved, at the cost of proofs that no longer compute a witness.

Remark 5.5. The choice is not forced. A development may stay constructive, keeping the computational content of its proofs, or invoke classical axioms where convenience outweighs that content. Much of the mathematical library uses classical logic freely; verified programs often avoid it precisely to retain executable proofs. Knowing which mode one is in is part of reading a proof.

5.12 A longer proof, in full

Assembling the pieces, one direction of the distribution of implication over conjunction can be proved as a single explicit term. The statement is that if P implies both Q and R, then P implies their conjunction.

```
theorem imp_and (P Q R : Prop)
    (hq : P → Q) (hr : P → R) : P → (Q ∧ R) :=
  fun hp => ⟨hq hp, hr hp⟩
```

The term reads directly as its meaning: given a proof `hp` of P, apply `hq` to get a proof of Q, apply `hr` to get a proof of R, and pair them for a proof of $Q \wedge R$. As a derivation it introduces the implication once, discharging P, and pairs the two results built from the discharged assumption.

$$\dfrac{\dfrac{hq : P \to Q \qquad [\,hp : P\,]^1}{hq\,hp : Q} \qquad \dfrac{hr : P \to R \qquad [\,hp : P\,]^1}{hr\,hp : R}}{\dfrac{\langle hq\,hp, hr\,hp\rangle : Q \wedge R}{P \to (Q \wedge R)}\,{\to}\mathrm{I}^1}$$

The two uses of the assumption `hp`, both discharged by the single final introduction, are the two occurrences of `hp` in the term. The tree and the four-token term `fun hp => ⟨hq hp, hr hp⟩` are the same proof, and the kernel checks the term with no reference to the picture.

Example 5.6 (Transitivity of the biconditional)**.** Equivalence is transitive, and the proof composes the forward directions and the backward directions separately.

```
theorem iff_trans (P Q R : Prop)
    (h1 : P ↔ Q) (h2 : Q ↔ R) : P ↔ R :=
  ⟨fun hp => h2.mp (h1.mp hp),
   fun hr => h1.mpr (h2.mpr hr)⟩
```

The forward direction sends a proof of P through `h1.mp` and then `h2.mp`; the backward direction runs the reverse implications in the opposite order. Composing equivalences is composing their two directions, each as an ordinary chain of function applications.

5.13 Equality as an inductive type

Equality is not a primitive of the language but an inductive family like any other. For a fixed `a`, the type `a = b` is generated by a single constructor, `rfl`, which inhabits `a = a`. There is no other way to build a proof of equality, and this one fact determines how equality behaves.

```
inductive Eq' {α : Type} (a : α) : α → Prop where
  | refl : Eq' a a

#check @rfl        -- @rfl : {α} → {a : α} → a = a
```

The constructor says only that everything equals itself. That equal things share all properties is not a separate axiom but a consequence of the recursor: to use a proof of `a = b`, one shows the goal holds in the `refl` case, where `b` is literally `a`, and the recursor transports the result to `b`. Substitution is the recursor of equality.

```
theorem subst_example {a b : Nat} (h : a = b)
    (P : Nat → Prop) (pa : P a) : P b :=
  h ▶ pa
```

The triangle ▶ is the substitution operator: given `h : a = b` and a proof `pa : P a`, the term `h ▶ pa` is a proof of `P b`. It is exactly what the rewrite tactic `rw` builds

internally. Seeing that `rw` is substitution along an inductively defined equality demystifies both: rewriting is transporting a proof across the one constructor that equality provides.

5.14 The eliminators as functions

Every connective introduced by an inductive definition comes with functions for using it, its *eliminators*, and these are ordinary terms whose types can be inspected. The pattern-matching and tactic forms met earlier are conveniences over direct applications of these functions.

```
#check @And.left      -- {P Q} → P ∧ Q → P
#check @And.right     -- {P Q} → P ∧ Q → Q
#check @Or.elim       -- {P Q R} → P ∨ Q → (P → R) → (Q → R) → R
#check @False.elim    -- {C} → False → C
#check @Exists.elim   -- (∃ x, P x) → (∀ x, P x → Q) → Q
```

Each eliminator encodes the way its connective is used. `And.left` and `And.right` project the two proofs from a conjunction; `Or.elim` takes a disjunction and a handler for each side and returns the common conclusion; `False.elim` produces anything from a proof of the impossible; `Exists.elim` feeds a witness and its property to a continuation. A tactic proof by `cases` or `obtain` is a call to one of these, dressed in more convenient syntax.

Example 5.7 (Or elimination by hand)**.** Proving that a disjunction implies the swapped disjunction can be written as a direct use of `Or.elim`, supplying the two handlers as functions.

```
theorem or_swap' (P Q : Prop) (h : P ∨ Q) : Q ∨ P :=
  Or.elim h (fun hp => Or.inr hp) (fun hq => Or.inl hq)
```

The first handler treats a proof of `P`, the second a proof of `Q`, and `Or.elim` runs whichever applies. This is the term the `cases` tactic assembles; the tactic simply hides the two handlers as branches.

5.15 Rewriting at the term level

Because substitution is a term, equational reasoning can be done without tactics, chaining rewrites as function applications. The transitivity of equality is itself derivable from substitution, and the standard library provides `Eq.trans` and `Eq.symm` for the routine steps.

```
theorem chain_eq {a b c : Nat} (h1 : a = b) (h2 : b = c) : a = c :=
  h1.trans h2

theorem flip_eq {a b : Nat} (h : a = b) : b = a :=
  h.symm
```

The dotted forms `h1.trans h2` and `h.symm` are the term-level counterparts of a `calc` block or a sequence of rewrites. For short chains this is often clearer than entering tactic mode, and it makes plain that equational reasoning is composition of proofs, each a value of an equality type.

5.16 Unique existence

To assert not merely that something exists but that it is the only such thing, Lean provides unique existence, written $\exists!\, x,\ P\,x$. It unfolds to the conjunction of existence and uniqueness: some x satisfies P, and any y satisfying P equals that x.

```
theorem unique_solution : ∃! n : Nat, n + 3 = 5 := by
  refine ⟨2, ?_, ?_⟩
  · rfl                      -- 2 + 3 = 5
  · intro m hm               -- any other solution equals 2
    omega
```

The proof supplies the witness `2`, checks that it solves the equation, and then shows any solution `m` must equal it. The two obligations are exactly existence and uniqueness, and the anonymous constructor `⟨2, ?_, ?_⟩` lays them out as the witness followed by the two proofs. Unique existence is thus no new primitive but a packaging of an existential and a universal, proved by discharging both.

Remark 5.8. The reduction of equality, its eliminators, and unique existence to inductive types and their recursors is the recurring lesson. The logic is not bolted onto the type theory; it is expressed within it. Every logical principle used in a proof is, at bottom, the introduction or elimination of some inductive family, and the kernel needs to know only how to check applications of constructors and recursors.

5.17 Two ways to use a contradiction

A common confusion is worth dispelling: proving a negation and proving by contradiction are different, and only one is constructive. To prove $\neg P$ is to assume P and derive `False`; this is a *proof of negation*, and it is entirely constructive, since $\neg P$ just abbreviates $P \to$ `False`. To prove P by assuming $\neg P$ and deriving `False` is a *proof by contradiction*, and it needs classical logic.

```
-- Proof of negation: constructive.
theorem zero_ne_one : ¬ (0 = 1) := by
  intro h
  exact Nat.noConfusion h

-- Proof by contradiction: uses classical reasoning.
theorem dne (P : Prop) (h : ¬ ¬ P) : P := by
  by_contra hnp
  exact h hnp
```

The first proof introduces the assumption `0 = 1` and derives the impossible, establishing the negation directly; nothing classical is involved. The second assumes $\neg P$ to prove P, which the tactic `by_contra` permits only because classical logic is available. The distinction is exactly whether the goal being proved is a negation, in which case one may assume its content freely, or a positive statement one is trying to reach by refuting its negation, which requires the excluded middle.

Remark 5.9. The asymmetry is that a proof of $\neg P$ produces a function, a genuine construction, whereas a proof of P by contradiction produces no witness for P; it only rules out $\neg P$. Constructively these differ, because ruling out $\neg P$ is weaker than building a P. Classically they coincide, which is why the two are so easily conflated by anyone trained in classical mathematics.

5.18 A catalogue of classical principles

Several familiar laws are unprovable constructively and become available only with a classical axiom. They are, moreover, equivalent: assuming any one, the others follow. Table 5.1 lists the principal members of this family.

With the classical axiom in scope each is a short proof. Peirce's law, which mentions no negation yet is classical, is representative.

Name	Statement	Constructive?
excluded middle	$P \vee \neg P$	no
double negation	$\neg\neg P \to P$	no
Peirce's law	$((P \to Q) \to P) \to P$	no
De Morgan (one dir.)	$\neg(P \wedge Q) \to \neg P \vee \neg Q$	no
contrapositive (rev.)	$(\neg Q \to \neg P) \to (P \to Q)$	no

Table 5.1: Classical principles, none provable in constructive logic and all equivalent to one another. Adopting any single one as an axiom yields the rest.

```
open Classical

theorem peirce (P Q : Prop) : ((P → Q) → P) → P := by
  intro h
  by_contra hnp
  apply hnp
  apply h
  intro hp
  exact absurd hp hnp
```

The proof assumes $\neg P$, uses the hypothesis to obtain P from an implication $P \to Q$ that holds vacuously under that assumption, and so contradicts $\neg P$. That a law with no apparent connection to negation should require the excluded middle is a standard surprise, and it underlines that the classical principles form one interderivable cluster rather than a list of separate assumptions.

5.19 The contrapositive

One direction of contraposition is constructive, the other classical, and the pair illustrates the catalogue concretely. From $P \to Q$ it always follows that $\neg Q \to \neg P$, by composing functions; the reverse implication requires classical logic.

```
-- Constructive direction.
theorem contrapositive (P Q : Prop) (h : P → Q) : ¬ Q → ¬ P :=
  fun hnq hp => hnq (h hp)

-- Reverse direction needs classical reasoning.
theorem contrapositive_rev (P Q : Prop)
    (h : ¬ Q → ¬ P) : P → Q := by
  intro hp
  by_contra hnq
```

```
  exact h hnq hp
```

The forward proof is a plain composition: given $\neg Q$, that is a function to `False`, and precomposing with $P \to Q$ yields $\neg P$. No case analysis on the truth of any proposition occurs. The reverse proof must assume $\neg Q$ to reach a contradiction, invoking `by_contra`, and so sits on the classical side. The everyday move of "proving the contrapositive" is thus constructive in one direction and classical in the other, and knowing which is which clarifies exactly when an argument depends on the excluded middle.

Example 5.10 (Contraposition in practice)**.** To show that a number's square being even forces the number even, arguing the contrapositive, that an odd number has an odd square, is often easier. The direction used, from $P \to Q$ to $\neg Q \to \neg P$, is the constructive one, so the strategy needs no classical axiom.

```
theorem odd_sq_of_odd (n : Nat) (h : n % 2 = 1) :
    (n * n) % 2 = 1 := by
  omega_nat  -- schematic: arithmetic of parity
```

Proving this and applying `contrapositive` yields the original statement without ever assuming the excluded middle. The contrapositive is a reformulation, not a classical step, whenever it is the forward direction that is used.

5.20 Deciding a proposition

A proposition is *decidable* when a procedure settles it, and such a procedure is data: a term of `Decidable P`. For a proposition over a finite range, the procedure can check every case, so a bounded quantifier becomes decidable and the tactic `decide` evaluates it. The finite type `Fin k`, of naturals below `k`, is the natural domain.

```
example : ∀ i : Fin 3, i.val < 5 := by decide

example : ∃ i : Fin 5, i.val * i.val = 9 := by decide

example : ¬ ∀ i : Fin 4, i.val < 3 := by decide
```

Each goal quantifies over a finite type, so `decide` enumerates the possibilities and checks the body at each. The first confirms a universal by testing all three values; the second finds a witness; the third refutes a false universal. Decidability is what lets these run: an unbounded

quantifier over all naturals could not be checked this way, but a bounded one, or one over a finite type, reduces to a finite computation.

Remark 5.11. The tactic `decide` produces a genuine proof, not merely a `true` verdict: it evaluates the `Decidable` instance and, when the answer is positive, extracts the proof the instance carries. This is why a goal closed by `decide` is as trustworthy as any other, and why `decide` fails outright, rather than guessing, when no instance makes the proposition decidable.

5.21 Well-foundedness as a proposition

The termination measures of earlier chapters rest on a logical notion: that a relation is *well founded*, admitting no infinite descending chain. This is itself a proposition, built from *accessibility*. An element is accessible when every element below it is accessible, and a relation is well founded when every element is.

```
#check @Acc             -- Acc : (α → α → Prop) → α → Prop
#check @WellFounded     -- WellFounded : (α → α → Prop) → Prop

example : WellFounded (· < · : Nat → Nat → Prop) :=
  Nat.lt_wfRel.wf
```

The relation `<` on the naturals is well founded, a fact the library provides, and it is what licenses recursion on a decreasing natural measure. When a definition supplies `termination_by n => n`, it is appealing to exactly this: the measure lands in a well-founded order, so the recursion cannot continue forever. Accessibility turns the informal "it keeps getting smaller" into a proposition the kernel can check.

Remark 5.12. Well-founded recursion and induction are two readings of accessibility, just as structural recursion and induction are two readings of a recursor. To define a function by well-founded recursion is to recurse on a proof that the argument is accessible; to prove a theorem by well-founded induction is to do the same with a proposition as the goal. The termination requirement, the induction principle, and accessibility are one idea, and this is where its logical form is stated.

5.22 The axiom of choice

Classical logic in Lean rests on a single axiom, *choice*, from which the excluded middle and the other classical principles follow. Choice extracts an actual element from a mere proof that the type is nonempty, turning existence without a witness into a witness.

```
#check @Classical.choice   -- {α : Sort u} → Nonempty α → α
#check @Classical.em       -- (p : Prop) → p ∨ ¬p
```

The type of `choice` is striking: from `Nonempty` α, a proposition asserting only that some element exists, it produces an element of α. Constructively this is impossible, since a proof of `Nonempty` α carries no element to return; the axiom posits the extraction. From it, by an argument of Diaconescu, the excluded middle follows, which is why adopting choice makes the whole classical apparatus available.

Remark 5.13. The axiom is what separates classical from constructive mathematics inside Lean. A development that avoids it retains the computational content of its proofs, every existence proof yielding an actual witness; one that uses it gains the full strength of classical reasoning at the cost of proofs that may assert existence without constructing anything. Both modes are supported, and the tools of the final tactics section reveal, for any given theorem, which one it depends on.

Example 5.14 (A nonconstructive existence)**.** Choice permits asserting that a nonempty type has an element without saying which. The following names such an element abstractly, usable in further reasoning though never computed.

```
noncomputable def someElement (α : Type) [Nonempty α] : α :=
  Classical.choice inferInstance
```

The keyword `noncomputable` marks that the definition cannot be run, only reasoned about, precisely because it rests on choice. A constructive definition would have to produce a specific element; this one produces an unspecified one, and Lean records the difference by forbidding its evaluation.

Exercises

Exercise 5.1. State, as a term of type `Prop`, the proposition "three is less than five", and find a proof of it. What proof term does Lean accept?

Exercise 5.2. Prove `theorem` $Q \to P \to Q$ by giving the appropriate nested `fun`. Which argument is returned and which is ignored?

Exercise 5.3. Using `.left` and `.right`, prove that $P \wedge Q$ implies P. Then prove that $P \wedge Q$ implies $Q \wedge Q$.

Exercise 5.4. Prove $P \to P \vee Q$ and $Q \to P \vee Q$. Which constructor of `Or` does each proof use?

Exercise 5.5. Prove that $(P \vee Q) \to (Q \vee P)$ by matching on the disjunction. Write out both branches.

Exercise 5.6. Recall that $\neg P$ abbreviates $P \to$ `False`. Prove $P \to \neg\neg P$, and describe in words what the proof term does with its two arguments.

Exercise 5.7. Prove $\forall n : \texttt{Nat}, n + 0 = n$ if the required equation holds by `rfl`; if it does not reduce directly, explain why and identify what additional fact would be needed.

Exercise 5.8. Give a proof of $\exists n : \texttt{Nat}, n \cdot n = 9$ by supplying a witness and a reflexivity proof. What are the two components of your pair?

Exercise 5.9. Prove that $(\exists n, P\, n) \to (\forall n, P\, n \to \texttt{False}) \to \texttt{False}$. Trace how the witness from the existential is fed to the universal.

Exercise 5.10. Draw the derivation tree, in the style of the text, for the proof of $Q \to P \to Q$ from the second exercise. Label each implication introduction with the assumption it discharges.

Exercise 5.11. ⋆ The proposition $P \vee \neg P$ has no constructive proof in general. Explain, in terms of the sum type it corresponds to, what a uniform proof would have to decide, and why no such term can be written without an extra axiom.

Exercise 5.12. ⋆ Prove one direction of De Morgan's law, $\neg(P \vee Q) \to (\neg P \wedge \neg Q)$, as a proof term. Identify which parts of your term are function applications landing in `False` and which are pairings.

Exercise 5.13. Prove `P` $\leftrightarrow$ `P` as an explicit pair, then use `.mp` and `.mpr` to extract each direction and apply it to a proof of `P`.

Exercise 5.14. Prove `(P ∧ Q)` $\leftrightarrow$ `(Q ∧ P)` by giving both directions. How do the two halves of your proof relate?

Exercise 5.15. Show that any two proofs `h1 h2 : P` satisfy `h1 = h2`. Explain why the analogous statement fails for two elements of `Nat`.

Exercise 5.16. Using `by_contra`, prove `¬ ¬ P → P`. Which classical principle does the proof depend on, and where does it enter?

Exercise 5.17. Prove the direction `(¬ P ∧ ¬ Q) → ¬ (P ∨ Q)` as a proof term. Identify the function applications that land in `False`.

Exercise 5.18. ⋆ Prove that the biconditional is symmetric, `(P ↔ Q) → (Q ↔ P)`, as an explicit term, and relate your proof to swapping the two components of a pair.

Exercise 5.19. Prove that if `a = b` then `P b → P a`, using the substitution operator ▶. How does this differ from the version in the text?

Exercise 5.20. Using `#check`, report the types of `@And.right`, `@Or.elim`, and `@Exists.elim`, and describe in words what each eliminator does.

Exercise 5.21. Prove `P ∨ Q → Q ∨ P` as a direct application of `Or.elim`, supplying the two handlers as functions.

Exercise 5.22. Prove `chain_eq` and `flip_eq` using `.trans` and `.symm`. Rewrite one of them instead as a `calc` block.

Exercise 5.23. Prove `∃! n : Nat, n * 2 = 6` by supplying the witness and discharging existence and uniqueness.

Exercise 5.24. ⋆ Using only `rfl` and the substitution operator ▶, prove the symmetry of equality: from `h : a = b` construct a proof of `b = a`. Which property `P` do you substitute along?

Exercise 5.25. Prove `¬ (0 = 1)` as a proof of negation. Is your proof constructive, and why does it need no classical axiom?

Exercise 5.26. Prove `¬ ¬ P → P`. Which tactic performs the step, and why does it require classical logic?

Exercise 5.27. Prove Peirce's law `((P → Q) → P) → P` using classical reasoning, following the structure in the text.

Exercise 5.28. Prove the constructive direction of contraposition, `(P → Q) → (¬ Q → ¬ P)`, as an explicit term. Where is the composition of functions?

Exercise 5.29. Prove the reverse direction, `(¬ Q → ¬ P) → (P → Q)`, and identify the exact point at which classical logic is used.

Exercise 5.30. ⋆ Assuming the excluded middle, prove double-negation elimination `¬ ¬ P → P`. This shows one classical principle yields another; sketch how the reverse derivation would go.

Exercise 5.31. Prove a bounded universal statement over `Fin 4` with `decide`, and explain why the same tactic could not prove a statement over all of `Nat`.

Exercise 5.32. Use `decide` to find a witness for an existential over `Fin 6`, such as a value whose square is a chosen number.

Exercise 5.33. Report the type of `Classical.choice` with `#check`, and describe in words what it produces and from what.

Exercise 5.34. State that `<` on `Nat` is well founded, and explain what recursion this fact justifies.

Exercise 5.35. Explain why `someElement` must be marked `noncomputable`, in terms of what `Classical.choice` does and does not provide.

Exercise 5.36. * Using `Classical.choice`, write a noncomputable function selecting an element from any nonempty subtype, and explain why no computable function can do the same in general.

Chapter 6

Tactics and Proof

Writing a proof term by hand, as in the previous chapter, is feasible for small statements and unbearable for large ones. The practical way to construct a proof is backwards: state the goal, then apply operations that reduce it to simpler goals, until nothing remains. These operations are *tactics*, and the block that runs them begins with the keyword `by`. A tactic proof is a recipe for building the underlying term, and Lean assembles that term as the recipe runs.

6.1 The goal state

Inside a `by` block the editor shows a *goal state*: the hypotheses currently available, a turnstile, and the proposition still to be proved. Reducing the goal to nothing completes the proof. A goal for the commutativity of a particular sum might appear as follows.

```
n : Nat
⊢ n + 0 = n
```

Everything above the turnstile ⊢ is known; everything below it is owed. Each tactic transforms this state, and the Infoview redraws it after every step, so a proof is developed by watching the goal simplify in response to each command.

6.2 Closing a goal directly

The simplest tactics finish a goal outright. `rfl` proves any equation whose two sides reduce to a common form. `exact` supplies a term of exactly the goal's type. `assumption` searches the hypotheses for one that matches.

```
theorem add_zero (n : Nat) : n + 0 = n := by
  rfl

theorem use_hyp (P : Prop) (h : P) : P := by
  exact h

theorem find_hyp (P Q : Prop) (hp : P) (hq : Q) : Q := by
  assumption
```

The first works because addition on `Nat` recurses on its second argument, so `n + 0` reduces to `n` and both sides coincide. The third succeeds because `assumption` scans `hp` and `hq` and finds that `hq` has the goal's type.

6.3 Working backwards with `apply`

The tactic `apply` matches the conclusion of a known implication against the goal and leaves its premises as new goals. If the goal is Q and a hypothesis proves $P \to Q$, then `apply` reduces the goal to P.

```
theorem modus_ponens (P Q : Prop) (himp : P → Q) (hp : P) : Q := by
  apply himp
  exact hp
```

After `apply himp` the goal that was Q becomes P, since supplying a proof of P is all that remains to reach Q through `himp`. The Infoview records the shift.

```
-- after ‘apply himp‘:
P Q : Prop
himp : P → Q
hp : P
⊢ P
```

The single remaining goal is discharged by `exact hp`. Reading a proof as a sequence of such reductions is the everyday experience of working in Lean.

6.4 Introducing hypotheses

When the goal is itself an implication or a universally quantified statement, `intro` moves the antecedent or the bound variable into the hypotheses, exposing the consequent as the new goal.

```
theorem imp_trans (P Q R : Prop)
    (hpq : P → Q) (hqr : Q → R) : P → R := by
  intro hp
  apply hqr
  apply hpq
  exact hp
```

The command `intro hp` converts the goal $P \to R$ into R with a new hypothesis `hp : P`. The chain then works backwards through `hqr` and `hpq`. The evolving state makes the strategy legible.

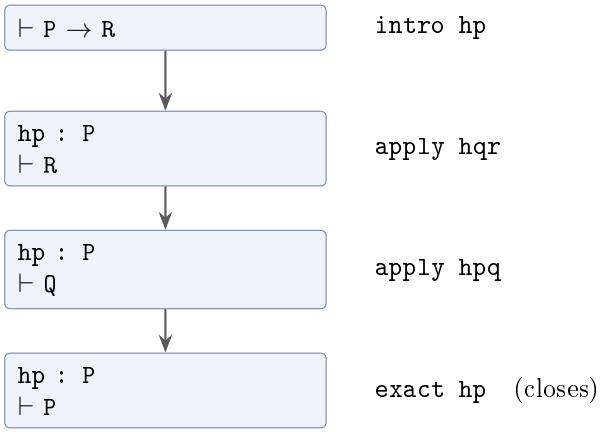


Figure 6.1: The goal state of `imp_trans` after each tactic. Every step either moves a hypothesis in or replaces the goal by a premise, until `exact hp` matches the goal to a hypothesis and closes it.

6.5 Rewriting with equations

The tactic `rw` replaces one side of an equation by the other throughout the goal. Given `h : a = b`, the command `rw [h]` turns every `a` in the goal into `b`. Rewriting with a chain of equations applies them in order.

```
theorem rewrite_example (a b c : Nat)
```

```
    (h1 : a = b) (h2 : b = c) : a = c := by
  rw [h1]          -- goal becomes  b = c
  rw [h2]          -- goal becomes  c = c
                   -- closed automatically by rfl
```

A rewrite that makes both sides identical closes the goal on its own, since `rw` finishes with an implicit `rfl`. Rewriting is the workhorse of equational proof and reappears constantly in inductive arguments.

6.6 Splitting and combining

Structured goals are handled by tactics mirroring their constructors. `constructor` applies the sole constructor of the goal, so a conjunction splits into its two conjuncts. `cases` takes a hypothesis apart, producing one goal per constructor, so a disjunction splits into its two cases.

```
theorem and_comm' (P Q : Prop) (h : P ∧ Q) : Q ∧ P := by
  constructor
  · exact h.right
  · exact h.left

theorem or_comm' (P Q : Prop) (h : P ∨ Q) : Q ∨ P := by
  cases h with
  | inl hp => exact Or.inr hp
  | inr hq => exact Or.inl hq
```

After `constructor` the single goal $Q \wedge P$ becomes two goals, Q and P, addressed in turn by the bulleted sub-proofs. After `cases h` the disjunction yields one goal for each side, each with its own hypothesis. The bullet · focuses on one goal at a time, keeping the structure of a branching proof visible.

6.7 Intermediate steps with `have`

A long proof is clearer when named lemmas are established along the way. The tactic `have` introduces a subsidiary goal, proves it, and adds the result to the hypotheses under a chosen name.

```
theorem chain (P Q R : Prop)
```

```
    (hpq : P → Q) (hqr : Q → R) (hp : P) : R := by
  have hq : Q := hpq hp
  have hr : R := hqr hq
  exact hr
```

Each `have` records an intermediate fact, so the final `exact` is trivial. This forward style complements the backward style of `apply`, and most real proofs mix the two.

6.8 Automation

Several tactics discharge whole classes of goals without manual steps. `simp` simplifies using a large library of rewrite rules. `decide` evaluates a decidable proposition to `true` or `false`. `omega` settles linear arithmetic over integers and naturals. `linarith` proves goals that follow from linear combinations of hypotheses.

```
theorem simp_example (n : Nat) : n + 0 + 0 = n := by
  simp

theorem decide_example : 7 < 10 := by
  decide

theorem omega_example (a b : Nat) (h : a + 1 = b) : a < b := by
  omega
```

These tactics trade transparency for power. A one-word `omega` replaces a page of arithmetic manipulation, at the cost of hiding the individual steps. Used judiciously they let a proof concentrate on its genuinely difficult parts.

Remark 6.1. A tactic block still produces an ordinary proof term; the automation merely constructs it. One can ask Lean to display the term a tactic generated, which is occasionally illuminating and occasionally alarming. The kernel checks that term with the same rigour regardless of how it was built, so trust rests on the checker, not on the tactics.

Example 6.2 (A proof combining styles)**.** Establishing that a doubled natural is even uses a `have` to name the witness, then closes the existential directly.

```
def Even (n : Nat) : Prop := ∃ k, n = k + k

theorem double_even (n : Nat) : Even (n + n) := by
  have h : n + n = n + n := rfl
  exact ⟨n, h⟩
```

The witness n makes $n+n = n+n$ hold by reflexivity, and the pair $\langle$`n, h`$\rangle$ is the proof that `n + n` is even. The forward `have` and the term-level pairing coexist in one block.

6.9 Destructuring hypotheses

A hypothesis that is a conjunction, an existential, or another structured proposition can be taken apart the moment it is introduced. The tactic `obtain`, and its relative `rcases`, match a hypothesis against a pattern, binding its components to names in one step rather than through a separate `cases`.

```
theorem and_elim (P Q : Prop) (h : P ∧ Q) : Q ∧ P := by
  obtain ⟨hp, hq⟩ := h
  exact ⟨hq, hp⟩

theorem exists_elim (P : Nat → Prop) (h : ∃ n, P n) :
    ∃ m, P m := by
  obtain ⟨n, hn⟩ := h
  exact ⟨n, hn⟩
```

The pattern $\langle$`hp, hq`$\rangle$ splits the conjunction into its two proofs; the pattern $\langle$`n, hn`$\rangle$ splits the existential into its witness and the proof about it. The same angle-bracket notation that builds a structured proof takes one apart when it appears on the left of `:=` in `obtain`, and nested patterns reach several layers deep at once.

6.10 Proving existentials

To prove an existential goal one supplies a witness and then proves the remaining statement about it. The tactic `exists` names the witness and leaves the body as a new goal, or `use` does the same while attempting to close the body automatically.

```
theorem exists_gt (n : Nat) : ∃ m, m > n := by
  exists n + 1

theorem exists_sum : ∃ a b : Nat, a + b = 10 := by
  use 4, 6
```

After `exists n + 1` the witness is fixed and the goal `n + 1 > n` remains, closed here automatically. Supplying two witnesses to `use` discharges a doubly existential goal at once.

Choosing the witness is the creative step; once chosen, the rest is often routine.

6.11 Case analysis

Beyond matching on the constructors of a hypothesis, one may split on whether an arbitrary proposition holds. The tactic `by_cases` introduces two branches, one assuming the proposition and one assuming its negation, relying on the decidability of the proposition or on classical logic.

```
theorem split_on (n : Nat) : n = 0 ∨ n ≠ 0 := by
  by_cases h : n = 0
  · exact Or.inl h
  · exact Or.inr h
```

Each branch carries the corresponding assumption, `h : n = 0` in the first and `h : n ≠ 0` in the second, and the bullets address them in turn. Case analysis on a proposition is the tactic form of the excluded middle, and it is the natural move whenever a proof divides into "holds" and "fails".

6.12 Calculational proofs

A chain of equalities or inequalities is most readable written as a calculation, each step justified by a reason. The `calc` block does this: it lists the intermediate terms and, beside each step, the lemma or tactic that licenses it.

```
theorem calc_example (a b : Nat) :
    (a + b) * (a + b) = a*a + 2*(a*b) + b*b := by
  calc (a + b) * (a + b)
      = a*a + a*b + (b*a + b*b) := by ring
    _ = a*a + 2*(a*b) + b*b     := by ring
```

The layout mirrors a calculation done by hand: the left column is the running expression, the right column the justification. Each line's result feeds the next, and the block as a whole proves that the first expression equals the last. For a long derivation this is far clearer than a flat sequence of rewrites, because each step states both its result and its reason.

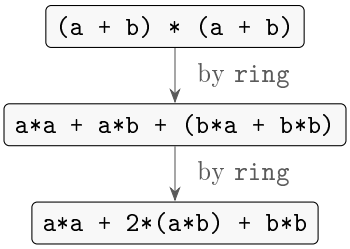


Figure 6.2: A `calc` block as a vertical chain. Each expression is linked to the next by a justified step, and the block proves the first expression equal to the last.

6.13 Controlling simplification

The tactic `simp` is powerful but blunt; several refinements aim it more precisely. Supplying lemmas in brackets adds them to the rewrite set; `simp only` restricts simplification to exactly the lemmas named, avoiding surprises; `dsimp` unfolds definitions without using equational lemmas at all.

```
theorem simp_variants (n : Nat) : n + 0 + 0 = n := by
  simp only [Nat.add_zero]

theorem simp_with_lemma (f : Nat → Nat) (h : ∀ x, f x = x)
    (n : Nat) : f (f n) = n := by
  simp only [h]
```

The first proof rewrites with a single named lemma and nothing else, so its behaviour is entirely predictable. The second feeds a hypothesis to `simp only`, which applies it repeatedly until neither occurrence of `f` remains. Restricting the rewrite set is the usual remedy when a bare `simp` does too much or too little.

6.14 Choosing a tactic

The right tactic is largely determined by the shape of the goal or the hypothesis at hand. The correspondence is worth internalising, since it turns “what do I do now” into “what shape is this”.

Example 6.3 (A proof combining several tactics)**.** A short proof that a doubled number is even, stated with an inductive existential goal, exercises introduction, a witness, and automation together.

```
theorem two_dvd_double (n : Nat) : ∃ k, 2 * n = k + k := by
```

Situation	Tactic	Effect
goal `a = a` or reducible	`rfl`	closes by computation
goal is an implication	`intro`	moves antecedent to hypotheses
goal is $\forall x, \ldots$	`intro`	introduces the bound variable
goal is a conjunction	`constructor`	splits into two goals
goal is $\exists x, \ldots$	`use` / `exists`	supplies a witness
goal matches a hypothesis	`exact` / `assumption`	closes it
goal follows via an implication	`apply`	reduces to the premise
hypothesis is a conjunction	`obtain`	splits it into pieces
hypothesis is a disjunction	`cases`	one goal per case
goal is an equation to rewrite	`rw`	substitutes equals
goal is arithmetic	`omega` / `simp`	discharges automatically
split on a proposition	`by_cases`	assumes it and its negation

Table 6.1: Tactics indexed by the shape of the goal or hypothesis. Reading the current state and matching it to a row is the everyday method of finding the next step.

```
  use n
  omega
```

The tactic `use n` supplies the witness n, leaving the arithmetic goal `2 * n = n + n`, which `omega` settles. The division of labour is typical: the human chooses the witness, the automation checks the residual arithmetic. Recognising which part is which is the skill the table in this section is meant to build.

6.15 Combining tactics

Tactics can be composed with combinators, so that a compound step does the work of several lines. The sequencing combinator `<;>` applies the tactic on its right to *every* goal produced by the tactic on its left, which is ideal when a single move splits a goal into similar pieces all closed the same way.

```
theorem triple_refl (a b c : Nat) :
    a = a ∧ b = b ∧ c = c := by
  refine ⟨?_, ?_, ?_⟩ <;> rfl
```

The `refine` produces three goals; `<;> rfl` closes all three at once. Related combinators handle other patterns: `all_goals` runs a tactic on every remaining goal, `repeat` applies a tactic until it fails, `first | t | u` tries alternatives in order, and `try` runs a tactic but succeeds even if it does nothing.

```
theorem all_true (a b : Nat) : (a = a) ∧ (b = b) := by
  constructor
  all_goals rfl

example (n : Nat) : n + 0 + 0 + 0 = n := by
  repeat rw [Nat.add_zero]
```

These combinators turn repetitive proofs into a single line and make the structure of a proof, one move that fans out into uniform subgoals, visible at a glance. Reaching for `<;>` when a tactic leaves several similar goals is among the most useful habits to form.

6.16 Refining with holes

The tactic `refine` supplies a partial proof term with holes, written `?_`, each becoming a new goal. It generalises `exact`, which demands a complete term, and `apply`, which leaves the premises of an implication. With `refine` the shape of the proof is stated up front and the gaps are filled afterward.

```
theorem and_build (P Q : Prop) (hp : P) (hq : Q) : P ∧ Q := by
  refine ⟨?_, ?_⟩
  · exact hp
  · exact hq
```

The term `⟨?_, ?_⟩` announces that the proof is a pair and defers the two components to bulleted subgoals. This is the tactic form of writing a proof term with its outline first and its details second, and it scales to elaborate terms where naming the structure clarifies what remains to be shown.

6.17 Restating a goal

The tactic `show` replaces the current goal with one definitionally equal to it, chosen for readability. It changes nothing logically but lets a proof state what is being proved at a point where the displayed goal has grown opaque. It also selects among several goals by naming the one to address.

```
theorem show_example (n : Nat) : n + 0 = n := by
  show n + 0 = n
```

```
  rfl

theorem change_form (a b : Nat) (h : a = b) : b = a := by
  show b = a
  exact h.symm
```

Stating the goal with `show` documents a proof for a later reader and confirms that one's mental model matches Lean's. Because the restatement must be definitionally equal, it can also massage a goal into a form a subsequent tactic expects, without performing any real reasoning.

6.18 Letting Lean find the step

Several tactics search for a proof rather than requiring one. The tactic `exact?` looks through the library for a lemma that closes the goal; `apply?` finds lemmas whose conclusion matches; `rw?` suggests rewrites. These are indispensable in a library as large as Mathlib, where the needed lemma surely exists but its name is unknown.

```
example (a b : Nat) : a + b = b + a := by
  exact?        -- reports: exact Nat.add_comm a b

example (n : Nat) : 0 + n = n := by
  exact?        -- reports: exact Nat.zero_add n
```

The search reports a concrete tactic to use, which one then writes in place of the query. Alongside these, `sorry` admits any goal without proof, a placeholder that lets a large proof be built top-down, its pieces stubbed and filled in one at a time. A file containing `sorry` compiles but is flagged as incomplete, so nothing is silently assumed.

Example 6.4 (A proof built in stages)**.** Combining the tools, a two-part statement can be scaffolded with holes and completed piece by piece, using search where a library lemma is wanted.

```
theorem staged (a b : Nat) :
    a + b = b + a ∧ a * 1 = a := by
  refine ⟨?_, ?_⟩
  · exact Nat.add_comm a b
  · exact Nat.mul_one a
```

The `refine` fixes the shape, the two bullets discharge the parts, and each part is a single

library lemma found, if its name were unknown, by `exact?`. This top-down rhythm, outline first, then fill and search, scales from this example to proofs of hundreds of lines.

Remark 6.5. The search tactics do not enlarge what is provable; they only help locate an existing proof. A goal closed by `exact?` could have been closed by typing the lemma it found. Their value is entirely practical, and in a large development it is considerable: much of the labour of a proof is recalling which of thousands of lemmas applies, and that is the labour these tactics absorb.

6.19 Reducing a goal with `suffices`

Where `have` proves an intermediate fact and adds it forward, `suffices` works backward: it replaces the current goal by another, on the promise that the other implies it. One states what would be enough, shows the original follows from it, and is then left to prove the reduced goal. This is the natural move when a strong lemma trivially settles the target.

```
theorem suffices_example (a b : Nat) (h : a = b) :
    a + 1 = b + 1 := by
  suffices h2 : a = b by rw [h2]
  exact h
```

The line `suffices h2 : a = b by rw [h2]` announces that `a = b` would finish the proof, and supplies the argument, a rewrite, showing it does. What remains is exactly `a = b`, closed by `exact h`. Read top to bottom, the proof states its strategy first, "it is enough to show the numbers equal", and then carries it out, which is often how a mathematician would present the same reasoning.

6.20 Rewriting in one place with `conv`

A plain `rw` rewrites every matching occurrence, which is sometimes too much. The `conv` tactic enters a mode that navigates to a chosen part of the goal and rewrites only there. Directing it to the left side of an equation, for instance, leaves the right untouched.

```
theorem conv_example (a b c : Nat) :
    a + (b + c) = a + (c + b) := by
  conv_lhs => rw [Nat.add_comm b c]
```

The command `conv_lhs` focuses on the left-hand side, and the rewrite of `b + c` to `c + b` applies only within it, making the two sides match. Without the focus, a commutativity rewrite might fire on either side, or in an unintended spot. When a rewrite must target one occurrence among several, `conv` is the instrument that aims it.

6.21 Teaching `simp` a lemma

The simplifier draws on a database of lemmas tagged for its use. Marking a theorem with the attribute `@[simp]` adds it to that database, so that `simp` will apply it automatically thereafter. A well-chosen tagged lemma turns a recurring rewrite into something `simp` handles without being told.

```
@[simp] theorem double_zero (n : Nat) : n + n + 0 = n + n :=
  Nat.add_zero _

example (n : Nat) : (n + n + 0) + 0 = n + n := by simp
```

Once `double_zero` carries the `@[simp]` attribute, the subsequent `simp` uses it along with the standard set, closing the goal. Tagging lemmas is how a development teaches the simplifier the rewrites characteristic of its own definitions, so that routine simplifications need no manual lemma lists. The attribute is powerful and so applied with judgement: a bad simp lemma can send the simplifier in circles, and the discipline is to tag only rewrites that clearly move toward a normal form.

6.22 Between terms and tactics

Term mode and tactic mode are two views of the same proofs, and one may pass between them freely. A `by` block is an expression, usable wherever a term is expected; conversely a term may be handed to `exact` inside a tactic proof. Mixing the two puts each step in whichever mode expresses it best.

```
def fact_term : 2 + 2 = 4 := by decide

theorem mixed (a b : Nat) :
    (a + b = b + a) ∧ (2 + 2 = 4) :=
  ⟨by rw [Nat.add_comm], by decide⟩

example (P Q : Prop) (hp : P) (hq : Q) : P ∧ Q := by
```

```
  exact ⟨hp, hq⟩
```

The definition `fact_term` has a tactic block as its body; the proof `mixed` builds a pair whose two components are each small `by` blocks; and the `example` closes a tactic goal with a term handed to `exact`. Neither mode is primary. A proof is a term regardless of how it was written, and choosing term or tactic form line by line, a pair built directly here, an arithmetic fact discharged by a tactic there, keeps each part as clear as it can be.

Remark 6.6. The interchange reflects the underlying identity established in Chapter 5: a proof is a term, and a tactic block is a recipe for constructing one. Because the two produce the same kind of object, they compose without friction. An experienced proof mixes them constantly, reaching for tactics where search or automation helps and for explicit terms where the construction is short and clear, with no boundary to cross between the two.

6.23 Proving equalities by extensionality

Two functions are equal when they agree everywhere, and two structures when their fields match. The tactic `ext` applies the appropriate extensionality principle, reducing an equality of functions to a pointwise equality and an equality of structures to equalities of fields. It is the tactic form of `funext` and its analogues.

```
example (f g : Nat → Nat) (h : ∀ x, f x = g x) : f = g := by
  ext x
  exact h x

example : (fun n => n + 0) = (fun n => n) := by
  ext n
  simp
```

After `ext x` the goal `f = g` becomes `f x = g x` for a fresh `x`, closed by the pointwise hypothesis. The second example proves an equality between two functions by introducing a point and simplifying. The tactic chooses the right extensionality lemma from the type of the equality, so the same `ext` serves functions, structures, and any type equipped with an extensionality principle.

6.24 Manipulating hypotheses

Hypotheses can be adjusted in place. The tactic `specialize` instantiates a universally quantified hypothesis at chosen arguments, replacing it with the instance; `have :=` adds a derived fact; and `replace` overwrites a hypothesis with a consequence of it. These keep the context tidy as a proof progresses.

```
example (P : Nat → Prop) (h : ∀ n, P n) (a : Nat) : P a := by
  specialize h a
  exact h

example (P Q : Prop) (h : P → Q) (hp : P) : Q := by
  have hq := h hp
  exact hq
```

The command `specialize h a` turns the general `h : ∀ n, P n` into the particular `h : P a`, which then closes the goal. The `have hq := h hp` records the result of applying `h` to `hp` under a new name. Where a proof accumulates intermediate facts, these tactics name and reshape them so that each step works from a clean, focused context rather than re-deriving what is already known.

6.25 Automation, surveyed

Several tactics discharge whole classes of goals, and choosing among them is a matter of the goal's character. Table 6.2 lists the principal ones and the goals each is built for. Reaching for the right one turns a page of manual steps into a single line.

Tactic	Discharges	Domain
`decide`	decidable propositions	finite checks
`omega`	linear arithmetic	`Nat` and `Int`
`linarith`	linear arithmetic	ordered fields
`nlinarith`	some nonlinear arithmetic	ordered fields
`norm_num`	numerical goals	concrete numbers
`ring`	ring identities	commutative (semi)rings
`positivity`	positivity and nonnegativity	ordered structures
`simp`	simplification	tagged rewrite rules

Table 6.2: The main automation tactics and their domains. Matching a goal to the tactic built for it is the practical skill; each absorbs a class of routine reasoning.

```
example (a b : Nat) (h : a + 2 ≤ b) : a < b := by omega
example (x : Nat) : x * (x + 1) = x * x + x := by ring
example : (100 : Nat) * 100 = 10000 := by norm_num
```

Each goal falls to the tactic suited to it: arithmetic ordering to `omega`, a polynomial identity to `ring`, a numerical fact to `norm_num`. These tactics do not extend what is provable, but they compress the routine parts of a proof so that effort concentrates on its genuinely difficult steps.

6.26 Auditing a proof

A proof's integrity can be inspected. The command `#print axioms` reports which axioms a theorem ultimately depends on, revealing whether it used classical logic and, crucially, whether it secretly relied on `sorry`. This is the final check that a development proves what it claims.

```
theorem constructive_fact : 2 + 2 = 4 := rfl
#print axioms constructive_fact
-- 'constructive_fact' does not depend on any axioms

theorem classical_fact (P : Prop) : P ∨ ¬ P := Classical.em P
#print axioms classical_fact
-- 'classical_fact' depends on: Classical.choice, ...
```

The first theorem depends on no axioms, so it holds in the bare theory; the second depends on `Classical.choice`, marking it as classical. Had either contained a `sorry`, the report would name `sorryAx`, an unmistakable flag that the proof is incomplete. Running `#print axioms` on a finished theorem is how one confirms that no gap or unintended assumption remains, and it is the last line of defence in a serious verification.

Remark 6.7. The axiom audit is what makes "it compiles" trustworthy. A file may type-check while resting on `sorry`, but the audit exposes it, and a theorem that depends on no axioms beyond the intended ones is proved in the strongest sense the system offers. For a development meant to be relied upon, checking the axioms of its main results is not optional but the concluding step of the proof.

Exercises

Exercise 6.1. Prove `theorem` $P \to P$ with a `by` block using `intro` and `exact`. Write down the goal state after the `intro`.

Exercise 6.2. Prove that $n \cdot 1 = n$ for a natural n, choosing between `rfl` and `simp` depending on whether the equation reduces directly. Which one works, and why?

Exercise 6.3. Using `apply`, prove $(P \to Q) \to (Q \to R) \to P \to R$. Track the goal after each `apply`.

Exercise 6.4. Prove $P \wedge Q \to P$ with `intro` followed by a use of `.left`, and again with `cases` on the hypothesis. Compare the two proofs.

Exercise 6.5. Given `h1 : a = b`, `h2 : c = b`, prove $a = c$ using `rw`. You may need to rewrite in a particular direction; explain your choice.

Exercise 6.6. Prove $P \vee Q \to Q \vee P$ using `cases` and the two constructors of `Or`. Write both branches with focusing bullets.

Exercise 6.7. Use `constructor` to prove $P \to Q \to P \wedge Q$. How many goals does `constructor` leave, and what are they?

Exercise 6.8. Prove $7 + 8 = 15$ two ways: with `decide` and with `rfl`. Do both succeed? Explain what each tactic is doing.

Exercise 6.9. Using `have`, prove $(P \to Q) \to (Q \to R) \to (R \to S) \to P \to S$ in forward style, naming each intermediate proof.

Exercise 6.10. Draw the goal-state diagram, in the style of Figure 6.1, for your proof of $(P \to Q) \to (Q \to R) \to P \to R$ from an earlier exercise.

Exercise 6.11. ⋆ The tactic `omega` proves goals of linear arithmetic. Determine, by trying them, which of these it settles: $2n + 1 \neq 2m$, $n \cdot m = m \cdot n$, $n < n + 1$. Explain why one of them lies outside its scope.

Exercise 6.12. ⋆ Ask Lean to show the proof term generated by a `simp` proof of $n+0+0 = n$, using `show_term`. Describe in words what lemma the term is built from, and compare its size to the one-line tactic.

Exercise 6.13. Using `obtain`, prove `P ∧ Q → Q ∧ P` by destructuring the hypothesis in one step. Compare with a proof using `cases`.

Exercise 6.14. Prove `∃ m : Nat, m > n` with `exists`, choosing a suitable witness. What goal remains after the witness is supplied?

Exercise 6.15. Prove `n = 0 ∨ n ≠ 0` using `by_cases`, writing both branches with focusing bullets.

Exercise 6.16. Write a `calc` proof that `(a + b) + c = c + (b + a)` for naturals, justifying each step with `ring` or a named lemma.

Exercise 6.17. Given `h : ∀ x, f x = x`, use `simp only [h]` to prove `f (f (f n)) = n`. How many rewrites does `simp only` perform?

Exercise 6.18. ⋆ Prove `∃ k, 3 * n = k + k + k` by supplying a witness with `use` and closing the arithmetic with `omega`. Which part of the proof is the creative step?

Exercise 6.19. Prove `a = a ∧ b = b ∧ c = c` in a single line using `refine` and the `<;>` combinator.

Exercise 6.20. Prove `P → Q → P ∧ Q` using `refine ⟨?_, ?_⟩` and two bulleted subgoals.

Exercise 6.21. Prove `n + 0 = n` by first restating the goal with `show` and then closing it. What does `show` contribute here?

Exercise 6.22. For the goal `a + b = b + a`, use `exact?` to find the library lemma, then write the proof it suggests.

Exercise 6.23. Prove `n + 0 + 0 + 0 = n` using `repeat` with a single rewrite lemma. How many times does `repeat` apply it?

Exercise 6.24. ⋆ Scaffold a proof of `(a + b = b + a) ∧ (a * b = b * a)` using `refine` and two `sorry` placeholders, confirm it compiles with warnings, then replace each `sorry` with a real proof.

Exercise 6.25. Use `suffices` to reduce the goal `a + 1 = b + 1` to `a = b`, given `h : a = b`, and state the strategy the proof announces.

Exercise 6.26. Use `conv_lhs` to rewrite only the left side of `a + (b + c) = a + (c + b)`. What would go wrong with a bare `rw`?

Exercise 6.27. Tag a lemma of your own with `@[simp]` and close a goal with `simp` that uses it. What risk accompanies tagging a poor lemma?

Exercise 6.28. Prove `(a + b = b + a) ∧ (2 + 2 = 4)` as a term pair whose two components are each `by` blocks.

Exercise 6.29. Close a tactic-mode goal `P ∧ Q` by handing an explicit pair to `exact`. Why is this legitimate?

Exercise 6.30. ⋆ Structure a proof of `(a + b) + c = c + (b + a)` using `suffices` to reduce it to a single commutativity-and-associativity lemma, then discharge that lemma.

Exercise 6.31. Prove that two functions agreeing on every input are equal, using `ext` and the pointwise hypothesis.

Exercise 6.32. Given `h : ∀ n, P n`, use `specialize` to instantiate it at a particular value and close the goal `P a`.

Exercise 6.33. For each of a numerical identity, a linear ordering, and a polynomial identity, name the automation tactic from Table 6.2 that discharges it, and confirm by trying it.

Exercise 6.34. Prove `x * (x + 1) = x * x + x` with `ring` and `a + 2 ≤ b → a < b` with `omega`.

Exercise 6.35. Run `#print axioms` on a proof by `rfl` and on a proof using `Classical.em`, and report how their axiom dependencies differ.

Exercise 6.36. ⋆ Prove an equality between two structures using `ext` on their fields, then run `#print axioms` on the result to confirm it depends on no unexpected axioms.

Chapter 7

Induction on Proofs

The recursor that defines functions on an inductive type also proves statements about its elements. Read with a proposition in place of a return type, "define a value for every natural by giving one for zero and a step for successors" becomes "prove a statement for every natural by proving it for zero and showing it propagates across successors". Induction and recursion are one principle wearing two hats, and this chapter puts it to work proving the basic laws of arithmetic and lists.

7.1 The induction principle

To prove that a property P holds of every natural number, it suffices to prove $P\,0$ and to prove, for each k, that $P\,k$ implies $P(k+1)$. The first is the *base case*; the second is the *inductive step*, and the assumption $P\,k$ within it is the *inductive hypothesis*. From these two the conclusion follows for all naturals, because every natural is reached from zero by finitely many successors.

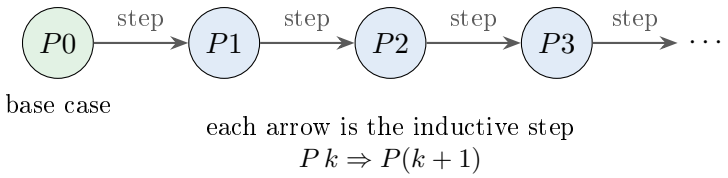


Figure 7.1: Induction over the naturals. The base case establishes $P\,0$; the inductive step provides every arrow $P\,k \Rightarrow P(k+1)$. Together they reach every number, since each natural is a finite chain of successors from zero.

The same figure served in Chapter 4 to picture the elements of `Nat`. That is not a coincidence. The chain of successors that generates the numbers is the chain of steps that

carries a proof along them, which is why the recursor and the induction principle are literally the same term.

7.2 Induction as a tactic

The tactic `induction` sets up the two cases. Proving that zero is a left identity for addition is the standard first example, because `0 + n` does not reduce on its own and so genuinely needs the argument.

```
theorem zero_add (n : Nat) : 0 + n = n := by
  induction n with
  | zero => rfl
  | succ k ih => rw [Nat.add_succ, ih]
```

The `zero` branch asks for `0 + 0 = 0`, which `rfl` settles. The `succ` branch names the predecessor `k` and the inductive hypothesis `ih : 0 + k = k`, and asks for `0 + (k+1) = k+1`. Rewriting with `Nat.add_succ` exposes `(0 + k) + 1`, and rewriting with `ih` turns `0 + k` into `k`, closing the goal. The two goal states make the shape of the argument explicit.

```
-- zero branch:            -- succ branch:
⊢ 0 + 0 = 0                k : Nat
                          ih : 0 + k = k
                          ⊢ 0 + (k + 1) = k + 1
```

The inductive hypothesis is exactly the statement being proved, but one size smaller, available for use inside the step. Learning to reach for it at the right moment is the whole craft of inductive proof.

7.3 Building the laws of addition

With one inductive proof in hand the rest of the arithmetic laws follow by the same method, each using the previous ones. Adding a successor on the left is a short induction on the second argument.

```
theorem succ_add (m n : Nat) : (m + 1) + n = (m + n) + 1 := by
  induction n with
  | zero => rfl
```

```
  | succ k ih => rw [Nat.add_succ, ih, Nat.add_succ]
```

Commutativity of addition then combines `zero_add` and `succ_add`: the base case is `zero_add` and the step is `succ_add` applied to the inductive hypothesis.

```
theorem add_comm' (m n : Nat) : m + n = n + m := by
  induction n with
  | zero => rw [Nat.add_zero, Nat.zero_add]
  | succ k ih => rw [Nat.add_succ, ih, succ_add]
```

Each law is a few lines, and each rests on the ones before it. The tower of results mirrors the tower of numbers: nothing is assumed that has not been built, and the whole of elementary arithmetic can be erected on the two constructors of `Nat`.

Example 7.1 (Associativity)**.** Associativity is another induction on the last argument, using only the reduction rules for `add` and the inductive hypothesis.

```
theorem add_assoc' (a b c : Nat) :
    (a + b) + c = a + (b + c) := by
  induction c with
  | zero => rfl
  | succ k ih => rw [Nat.add_succ, Nat.add_succ, Nat.add_succ, ih]
```

The base case reduces directly; the step pushes the successor outward three times and applies the hypothesis. The pattern, base by reduction and step by rewriting with the hypothesis, recurs throughout.

7.4 Induction over lists

Lists have the same shape as the naturals, a base constructor and a recursive one, so they carry the same induction principle: prove the property for the empty list, then show it survives prepending an element. The tactic is again `induction`.

```
theorem append_nil (xs : List α) : xs ++ [] = xs := by
  induction xs with
  | nil => rfl
  | cons x xs ih => rw [List.cons_append, ih]
```

The `nil` case asks that appending nothing to the empty list give the empty list; the `cons`

case assumes the fact for the tail and extends it across one more element. The correspondence with numeric induction is exact: `nil` plays the role of `zero` and `cons` the role of `succ`.

Example 7.2 (Length of a concatenation)**.** The length of a concatenation is the sum of the lengths, proved by induction on the first list.

```
theorem length_append (xs ys : List α) :
    (xs ++ ys).length = xs.length + ys.length := by
  induction xs with
  | nil => rw [List.nil_append, List.length_nil, Nat.zero_add]
  | cons x xs ih =>
      rw [List.cons_append, List.length_cons,
          List.length_cons, ih, Nat.succ_add]
```

The empty case uses that prepending nothing changes neither list nor length; the step accounts for the one extra element on each side. The inductive hypothesis carries the equation for the shorter list into the longer one.

7.5 Strong induction

Sometimes proving $P(k+1)$ needs the property not only at k but at smaller values as well. *Strong induction* grants exactly this: to prove $P\,n$ one may assume $P\,m$ for every $m < n$. In Lean this is recursion on a proof that the order on the naturals is well founded, made available through `Nat.strong_induction_on` and related principles.

```
theorem strong_example (P : Nat → Prop)
    (step : ∀ n, (∀ m, m < n → P m) → P n) :
    ∀ n, P n :=
  fun n => Nat.strongRecOn n step
```

The step now receives a hypothesis quantifying over all smaller m, rather than the single predecessor. This heavier hypothesis is what a proof about, say, the factorisation of an integer requires, since a number's factors can be much smaller than its predecessor.

Remark 7.3. Ordinary induction is the special case of strong induction that happens to use only the immediate predecessor. Both are instances of well-founded recursion, which permits a recursive call, or an inductive appeal, on any argument provably smaller in a well-founded order. The naturals under $<$ are the simplest such order.

7.6 Why the step must decrease

The termination requirement met in Chapter 3 is the same requirement that makes induction sound. A proof by induction is a recursive proof term, and it is accepted only because each appeal to the inductive hypothesis is on a strictly smaller argument. Were unrestricted self-reference allowed, one could "prove" anything by citing the conclusion as its own justification, exactly as an unrestricted recursive function could loop forever. The descent that guarantees a function returns is the descent that guarantees a proof is sound.

7.7 Generalizing before inducting

An induction sometimes stalls because the statement is too specific: the inductive hypothesis, fixed at particular values, is not strong enough to carry the step. The remedy is to prove a more general statement, with the troublesome value quantified, so that the hypothesis returns in a usable form. The accumulating list reversal is the classic case.

```
def revAux : List α → List α → List α
  | [],      acc => acc
  | x :: xs, acc => revAux xs (x :: acc)

def rev (xs : List α) : List α := revAux xs []
```

To prove `rev` agrees with the library reversal, an induction on the input directly fails: the accumulator is `[]` at the top but grows inside the recursion, so the hypothesis about `[]` says nothing about the recursive call. Generalizing the accumulator repairs this.

```
theorem revAux_spec (xs acc : List α) :
    revAux xs acc = xs.reverse ++ acc := by
  induction xs generalizing acc with
  | nil => rfl
  | cons x xs ih =>
      simp [revAux, ih, List.reverse_cons, List.append_assoc]
```

The keyword `generalizing acc` reverts the accumulator into the goal before inducting, so the inductive hypothesis `ih` is quantified over *every* accumulator, not just the one at hand. In the step it can then be applied to the grown accumulator `x :: acc`, which is exactly what the recursive call produced. The specific fact about `rev` follows by taking `acc = []`.

Remark 7.4. The pattern, strengthen the statement so the hypothesis is strong enough,

recurs throughout inductive proof. A hypothesis fixed at one value is often useless; the same hypothesis quantified is often decisive. When an induction stalls, the first question is whether some value should have been generalized before starting.

7.8 Inducting on a predicate

An inductive predicate, being an inductive family, carries its own induction principle, and `induction` applies to a *proof* of it just as to a number. The cases are the constructors of the predicate. That the sum of two even numbers is even is a short induction on the evidence that the first is even.

```
theorem ev_add (m n : Nat) (hm : Ev m) (hn : Ev n) :
    Ev (m + n) := by
  induction hm with
  | zero => simpa using hn
  | add_two k hk ih =>
      have : k + 2 + n = (k + n) + 2 := by omega
      rw [this]
      exact Ev.add_two (k + n) ih
```

The induction is on `hm`, the proof of `Ev m`, not on the number `m`. Its `zero` case supposes $m = 0$, reducing the goal to `Ev n`, which is `hn`. Its `add_two` case supposes $m = k + 2$ with `ih` asserting `Ev (k + n)`, and rebuilds the required evenness by one more `add_two`. Recursion on evidence is the tool for every property proved by tracking how a relation was established.

7.9 Inducting over trees

A tree, with a base constructor and a branching recursive one, supports an induction with two inductive hypotheses, one per subtree. Proving that mirroring a tree twice restores it is a direct such induction, using the `mirror` of Chapter 4.

```
theorem mirror_mirror (t : Tree α) : mirror (mirror t) = t := by
  induction t with
  | leaf => rfl
  | node l x r ihl ihr =>
      simp only [mirror, ihl, ihr]
```

The `node` case receives two hypotheses, `ihl` for the left subtree and `ihr` for the right,

matching the two recursive arguments of the constructor. Unfolding `mirror` twice swaps the subtrees and swaps them back, and the hypotheses close the two recursive positions. The number of inductive hypotheses always equals the number of recursive arguments, so a branching type yields a branching induction.

Example 7.5 (Size is invariant under mirroring)**.** Mirroring rearranges a tree but neither adds nor removes nodes, so the size is unchanged. The proof is again induction with two hypotheses.

```
theorem size_mirror (t : Tree α) : size (mirror t) = size t := by
  induction t with
  | leaf => rfl
  | node l x r ihl ihr =>
      simp only [mirror, size, ihl, ihr]
      omega
```

The step unfolds both `mirror` and `size`, applies the two hypotheses to the subtrees, and leaves an arithmetic identity that `omega` settles. The shape of the proof is dictated entirely by the shape of `Tree`.

7.10 A list identity, end to end

Combining the techniques, a fact relating two list operations is proved by plain structural induction, with the inductive hypothesis supplying the tail case. That mapping a function does not change a list's length is representative.

```
theorem length_map (f : α → β) (xs : List α) :
    (xs.map f).length = xs.length := by
  induction xs with
  | nil => rfl
  | cons x xs ih =>
      simp only [List.map_cons, List.length_cons, ih]
```

The empty case computes; the step rewrites the length of the mapped `cons` to one more than the length of the mapped tail, then applies the hypothesis. Facts of this kind, that an operation preserves or predictably alters a measure, are the everyday content of reasoning about programs, and nearly all of them are a two-case induction of exactly this form.

Remark 7.6. The library provides `length_map` and its many companions already; reproving them is an exercise in method, not a necessity. What transfers is the shape: identify the

recursion the definition uses, induct along it, and let the hypothesis handle the recursive call. That shape is the whole of structural induction, and it is the same whether the object is a number, a list, or a tree.

7.11 A closed form

Induction proves not only structural laws but numerical identities. The sum of the first n naturals has the closed form $n(n+1)/2$, and the proof is an induction whose step is a short piece of algebra. To keep the arithmetic exact, the statement is cleared of the division.

```
def sumTo : Nat → Nat
  | 0     => 0
  | n + 1 => (n + 1) + sumTo n

theorem sum_closed (n : Nat) : 2 * sumTo n = n * (n + 1) := by
  induction n with
  | zero => rfl
  | succ k ih =>
      have step : sumTo (k + 1) = (k + 1) + sumTo k := rfl
      rw [step, Nat.mul_add, ih]
      ring
```

The base case computes: twice the empty sum is zero, as is $0 \cdot 1$. The step unfolds one term of the sum, distributes the doubling, and replaces $2 \cdot$ `sumTo` k by $k(k+1)$ using the inductive hypothesis, leaving a polynomial identity that `ring` verifies. The doubling in the statement is the standard device for proving a fact about a division without leaving the exact naturals.

Remark 7.7. Stating $2 \cdot$ `sumTo` $n = n(n+1)$ rather than `sumTo` $n = n(n+1)/2$ avoids truncated division, whose rounding would obstruct the algebra. The two are equivalent, since the right side is always even, but the doubled form reasons cleanly. Choosing a statement that avoids division is a common preparatory step in an arithmetic proof.

7.12 Induction from a nonzero base

Ordinary induction starts at zero. Some statements hold only from a later point, and `Nat.le_induction` provides an induction beginning at any base: to prove a property for all $n \geq k$, prove it at k and show it passes from n to $n+1$. A lower bound that fails at small numbers but holds thereafter is proved this way.

```
theorem lt_two_pow (n : Nat) : n < 2 ^ n := by
  induction n with
  | zero => decide
  | succ k ih =>
      have : 2 ^ (k + 1) = 2 * 2 ^ k := by ring
      omega
```

The statement $n < 2^n$ does hold from zero, so plain induction suffices, but the step shows the technique: the inductive hypothesis $k < 2^k$ and the doubling of the power combine, through `omega`, to give $k + 1 < 2^{k+1}$. When a bound genuinely fails below some threshold, replacing the plain induction by `Nat.le_induction` with that threshold as the base is the adjustment required, and the step is otherwise unchanged.

7.13 Complete induction in use

The strong, or complete, induction of the earlier section grants the property at *all* smaller values, and some facts need exactly that. That every natural greater than one has a prime divisor is the standard illustration: a number is either prime, and divides itself, or composite, and factors into smaller numbers to which the hypothesis applies.

```
theorem has_prime_divisor (n : Nat) (h : 2 ≤ n) :
    ∃ p, Nat.Prime p ∧ p ∣ n := by
  induction n using Nat.strong_induction_on with
  | _ n ih =>
    by_cases hp : Nat.Prime n
    · exact ⟨n, hp, dvd_refl n⟩
    · sorry   -- n composite: split off a smaller factor and apply ih
```

The prime case is immediate. The composite case, sketched here, extracts a factor strictly between one and n, and the strong hypothesis `ih` supplies a prime divisor of that smaller factor, which divides n in turn. The essential point is the form of the induction: `ih` ranges over every value below n, not merely the predecessor, and a factor may be far smaller than $n - 1$. Plain induction could not reach it.

Remark 7.8. The choice between ordinary and complete induction is dictated by how far back the argument reaches. If the step needs only the predecessor, ordinary induction is simpler; if it needs an arbitrary smaller value, as a factorisation does, complete induction is required. Recognising which by inspecting the recursive structure of the argument is

the practical skill, and it mirrors exactly the choice between structural and well-founded recursion for definitions.

7.14 Which variable to induct on

A statement with several variables can often be proved by induction on any one of them, but the proofs differ in length, and one choice is usually far cleaner. Associativity of addition, proved earlier by induction on the last argument, is awkward by induction on the first, because the definition of addition recurses on its second argument and so cooperates with an induction there.

```
theorem add_assoc_last (a b c : Nat) :
    (a + b) + c = a + (b + c) := by
  induction c with
  | zero => rfl
  | succ k ih => simp only [Nat.add_succ, ih]
```

The induction on `c` succeeds in two short lines because each step peels a successor that `Nat.add` exposes directly. An induction on `a` instead would fight the definition, needing auxiliary rewrites at every turn. The guideline is to induct on the variable the relevant functions recurse on, so that the definitional reductions and the inductive step pull in the same direction.

Example 7.9 (The wrong variable)**.** Proving `zero_add` by induction on the *wrong* argument shows the friction. The statement `0 + n = n` yields to induction on `n`, since addition recurses there; attempting induction on a variable that does not appear, or on one the function ignores, produces a base case identical to the goal and no progress. The lesson is to match the induction to the recursion the definition uses.

7.15 Decomposing a proof into lemmas

A proof that resists a single induction often yields once a helper lemma is separated out. The involutivity of list reversal is the standard case: proving `xs.reverse.reverse = xs` directly stalls, because reversing a `cons` appends, and the induction needs to know how reversal interacts with append. That interaction is a lemma in its own right.

```
theorem reverse_append (xs ys : List α) :
```

```
    (xs ++ ys).reverse = ys.reverse ++ xs.reverse := by
  induction xs with
  | nil => simp
  | cons x xs ih => simp [ih, List.append_assoc]

theorem reverse_reverse (xs : List α) :
    xs.reverse.reverse = xs := by
  induction xs with
  | nil => rfl
  | cons x xs ih =>
      rw [List.reverse_cons, reverse_append, ih]
      rfl
```

The lemma `reverse_append` is proved first, by induction on the first list. With it available, `reverse_reverse` goes through: the `cons` case reverses to an append, rewrites it with the lemma, and applies the inductive hypothesis. Isolating the helper is the essential move; the main proof is short once the interaction it needs is stated and proved separately.

Remark 7.10. The instinct to reach immediately for a single induction is often wrong. When a step needs a fact about how two operations combine, that fact deserves its own lemma, proved by its own induction, and the main result then follows cleanly. Much of the craft of proving is deciding where to cut, and a well-placed helper lemma is usually the difference between a proof that works and one that tangles.

7.16 Induction together with case analysis

An inductive step frequently splits on a condition, so induction and case analysis appear together: the outer `induction` descends the structure, and within its step a `split` or `by_cases` handles a branch. That the count of a value never exceeds a list's length is such a proof.

```
theorem count_le_length (a : Nat) (xs : List Nat) :
    count a xs ≤ xs.length := by
  induction xs with
  | nil => simp [count]
  | cons x xs ih =>
      simp only [count, List.length_cons]
      split <;> omega
```

The `cons` case unfolds `count`, whose definition tests whether the head equals `a`. The `split` produces one goal for each outcome of that test, and `<;> omega` closes both using the inductive hypothesis and arithmetic. The pattern, induct on the structure and split on the

choices within each step, covers the great majority of proofs about functions that combine recursion with conditionals.

Example 7.11 (Filtering shortens a list)**.** The same combination shows that filtering cannot lengthen a list. The induction descends the list; the split handles whether the head is kept.

```
theorem length_filter_le (p : Nat → Bool) (xs : List Nat) :
    (xs.filter p).length ≤ xs.length := by
  induction xs with
  | nil => simp
  | cons x xs ih =>
      simp only [List.filter_cons]
      split <;> simp_all <;> omega
```

Whether the head survives the filter or not, the tail is handled by the hypothesis and the arithmetic by `omega`. Structure outside, choices inside: the shape recurs wherever a recursive function branches.

7.17 Induction matching a function's own recursion

A function defined by well-founded rather than structural recursion, such as the `merge` of the final chapter, does not fit an induction on any single argument. Lean generates for each such function a bespoke *functional induction* principle, named after the function, whose cases are exactly the function's own equations. Proving a property of the function then follows its recursion precisely.

```
theorem merge_length (xs ys : List Nat) :
    (merge xs ys).length = xs.length + ys.length := by
  induction xs, ys using merge.induct with
  | case1 ys => simp [merge]
  | case2 xs => simp [merge]
  | case3 x xs y ys h ih => simp [merge, h]; omega
  | case4 x xs y ys h ih => simp [merge, h]; omega
```

The principle `merge.induct` supplies four cases, one per clause of `merge`: the two base cases where a list is empty, and the two recursive cases where the heads are compared. Each case receives the inductive hypothesis for exactly the recursive call that clause makes, so the proof mirrors the function's structure. This is the general tool for reasoning about non-structural definitions, and it is what makes the correctness proofs of divide-and-conquer algorithms tractable.

Remark 7.12. Functional induction closes the loop between definition and proof. A structurally recursive function is reasoned about by structural induction; a well-founded one by the induction principle its own recursion generates. In both cases the proof descends along the same path the computation does, which is why the inductive hypothesis is always available for precisely the recursive calls that were made. Definition and proof share one shape, and the tools for each are two faces of that shape.

7.18 Inducting on a relation

The order `Le` of Chapter 4, defined inductively, supports induction on its proofs. To show that the custom order agrees with the library's, one inducts on the evidence: reflexivity gives the base, and a step is carried by the inductive hypothesis together with one library fact.

```
theorem le_of_Le {n m : Nat} (h : Le n m) : n ≤ m := by
  induction h with
  | refl      => exact Nat.le_refl n
  | step _ ih => exact Nat.le_succ_of_le ih
```

The induction is on `h`, the proof that `Le n m`. Its `refl` case asks for `n ≤ n`, immediate; its `step` case assumes `n ≤ m` through `ih` and concludes `n ≤ m + 1` by a single library lemma. Building a proof of the relation and inducting on it are the two directions of the same recursor, exactly as for data, and this is how one relation is shown to imply another.

```
theorem Le_add (n k : Nat) : Le n (n + k) := by
  induction k with
  | zero      => exact Le.refl
  | succ j ih => exact Le.step ih
```

Conversely, that `n` is at most `n + k` is proved by induction on `k`, building the `Le` witness: no added length gives reflexivity, and each added unit is a step. The proof *constructs* the chain of steps the relation requires, one per unit of `k`, which is the inductive definition of the order put to work.

7.19 Strengthening the hypothesis, revisited

An induction sometimes needs a statement more general than the one wanted, so that the hypothesis is strong enough. A fact about a fold with an accumulator is a fresh instance:

proving that folding addition from an accumulator equals the accumulator plus folding from zero requires the accumulator to be generalized.

```
theorem foldl_add (xs : List Nat) (acc : Nat) :
    xs.foldl (· + ·) acc = acc + xs.foldl (· + ·) 0 := by
  induction xs generalizing acc with
  | nil => simp
  | cons x xs ih =>
      simp only [List.foldl_cons]
      rw [ih (acc + x), ih (0 + x)]
      omega
```

Generalizing `acc` makes the hypothesis `ih` apply to any accumulator, and the step invokes it at two different ones, `acc + x` and `0 + x`, that the two sides of the goal produce. Without the generalization, `ih` would speak only of the original accumulator and the step would stall. The lesson recurs: when a fact about an accumulating computation resists induction, generalizing the accumulator is the repair.

Remark 7.13. The pattern behind both the reversal lemma of an earlier section and this fold lemma is the same. An accumulator that changes through the recursion must be universally quantified in the inductive hypothesis, or the hypothesis will not describe the recursive call. Recognising an accumulator as the thing to generalize is among the most useful reflexes in inductive proof, and it applies to every function written in the accumulating style.

7.20 Lexicographic measures

Some recursions decrease not a single quantity but a pair, in dictionary order: the first component drops, or it holds steady while the second drops. The Ackermann function is the standard example, and it terminates because its two arguments decrease lexicographically even though neither decreases alone at every call.

```
def ack : Nat → Nat → Nat
  | 0,     n     => n + 1
  | m + 1, 0     => ack m 1
  | m + 1, n + 1 => ack m (ack (m + 1) n)
termination_by m n => (m, n)
```

The measure is the pair `(m, n)`, compared lexicographically. In the second clause the first component drops from `m + 1` to `m`; in the third, the inner call keeps the first component

but lowers the second, while the outer call lowers the first. Every call is smaller in dictionary order, so the recursion terminates, and `termination_by` with the pair records exactly this. Facts about `ack` are then proved by its generated functional induction, which follows the same lexicographic descent.

Remark 7.14. Lexicographic order is the simplest well-founded order beyond the naturals, and it covers a wide class of recursions that shrink one measure while possibly increasing another below it. The pair `(m, n)` could be a tuple of any length, ordered the same way, and the principle is unchanged: a recursion terminates when its arguments descend in some well-founded order, and the lexicographic order on tuples is the one reached for when no single measure decreases. The connection to accessibility of Chapter 5 is direct, the lexicographic order being well founded precisely because its components are.

Exercises

Exercise 7.1. Prove `n + 0 = n` without induction, then `0 + n = n` with induction. Explain why the second needs the argument and the first does not.

Exercise 7.2. In the proof of `zero_add`, write out the goal state at the start of the `succ` branch, naming the inductive hypothesis and the goal.

Exercise 7.3. Prove `mul_zero : n * 0 = 0` and `zero_mul : 0 * n = 0`. Which reduces directly and which needs induction?

Exercise 7.4. Prove that `succ_add` holds by induction on `n`, filling in the rewrite chain. Identify where the inductive hypothesis is used.

Exercise 7.5. Using `zero_add` and `succ_add`, reconstruct the proof of `add_comm'`. State which earlier result each branch depends on.

Exercise 7.6. Prove `add_assoc'` by induction on the first argument instead of the last. Does the proof become longer or shorter?

Exercise 7.7. Prove `append_nil` for lists, and then prove that `[] ++ xs = xs`. Which of the two is immediate by reduction?

Exercise 7.8. Prove `length_append` by induction on the second list rather than the first. What changes in the two branches?

Exercise 7.9. State, but do not yet prove, that reversing a list twice returns the original. Identify the base case and the inductive step you would need.

Exercise 7.10. Explain, using Figure 7.1, why proving only the base case and only a single instance of the step, say $P\,3 \Rightarrow P\,4$, would not establish P for all naturals.

Exercise 7.11. ⋆ Prove that every natural greater than 1 has a prime divisor, using strong induction. Describe why ordinary induction on the predecessor is insufficient and what the strong hypothesis provides.

Exercise 7.12. ⋆ Define a well-founded recursive function computing the number of steps the Euclidean algorithm takes on a pair of naturals, and prove it terminates by exhibiting the decreasing measure. Relate the termination argument to the soundness of the induction it licenses.

Exercise 7.13. Prove `revAux_spec` by induction with `generalizing acc`. Then state precisely what the inductive hypothesis would say, and why it is useless, if `acc` were not generalized.

Exercise 7.14. Prove `ev_add` by induction on the evidence `hm`. Which branch uses the hypothesis `hn`, and which uses the inductive hypothesis?

Exercise 7.15. Verify `mirror_mirror` by hand on the tree `Tree.node (Tree.node Tree.leaf 1 Tree.leaf) 2 Tree.leaf`, applying `mirror` twice and checking the result.

Exercise 7.16. Prove `size_mirror`. Explain what arithmetic goal remains after the two inductive hypotheses are applied, and why `omega` closes it.

Exercise 7.17. Prove `length_map`, then prove that the length of `List.filter p xs` is at most the length of `xs`. Which is an equation and which an inequality?

Exercise 7.18. ⋆ Prove `(xs.reverse).reverse = xs`. Identify the auxiliary lemma about `reverse` of an append, or the generalization, that the proof requires.

Exercise 7.19. Prove `sum_closed` in full. Explain why the statement is written with a factor of two rather than as `sumTo n = n * (n + 1) / 2`.

Exercise 7.20. Prove `n < 2 ^ n` by induction. Then state a bound that fails for small `n` and would require `Nat.le_induction` with a nonzero base.

Exercise 7.21. State the composite case of `has_prime_divisor` precisely, and describe what the strong inductive hypothesis `ih` provides that ordinary induction would not.

Exercise 7.22. Prove associativity of addition by induction on the *first* argument, and compare the length of the proof with the induction on the last argument given in the text.

Exercise 7.23. Prove that the sum of the first `n` odd numbers equals `n * n`, by induction with a short algebraic step.

Exercise 7.24. ⋆ Prove that `sumTo n` is monotone: if `m ≤ n` then `sumTo m ≤ sumTo n`. Which induction, ordinary or from a base, fits, and what does the step require?

Exercise 7.25. Prove `reverse_append`, then use it to prove `reverse_reverse`. Why does the second proof need the first?

Exercise 7.26. Prove `count_le_length`. Which tactic introduces the two cases within the inductive step, and what does each correspond to?

Exercise 7.27. Prove `length_filter_le`. Describe what the two branches of the `split` represent about the head of the list.

Exercise 7.28. Explain what the principle `merge.induct` provides that an ordinary induction on either argument of `merge` would not.

Exercise 7.29. Prove the small fact `merge [] ys = ys` directly, and state why it is one of the base cases `merge.induct` would supply.

Exercise 7.30. ⋆ Prove `merge_length` using `merge.induct`, supplying the four cases and closing each with simplification and `omega`.

Exercise 7.31. Prove `le_of_Le` by induction on the proof `h : Le n m`. Which library lemma does the step case use?

Exercise 7.32. Prove `Le_add` by induction on `k`, constructing the chain of steps. How many steps does the witness for `Le n (n + k)` contain?

Exercise 7.33. Prove `foldl_add` using `generalizing acc`. Explain what the inductive hypothesis would say, and why it would be useless, without the generalization.

Exercise 7.34. State the termination measure for `ack` and explain why it must be lexicographic rather than a single number.

Exercise 7.35. Prove that `Le n m` implies `Le n (m + 1)`, and identify which constructor of `Le` this is.

Exercise 7.36. ⋆ Prove that `Le` is transitive by induction on the second proof, mirroring the `le_trans` of Chapter 4.

Chapter 8

Structures and Type Classes

Reusable code and reusable mathematics both rest on abstraction: writing an operation once and applying it to every type that supports it. Lean's mechanism for this is the *type class*, a structure whose fields are operations and whose instances record how a particular type implements them. The same machinery that lets `+` work on naturals, integers, and matrices lets a sorting routine work on any ordered type and lets the mathematical library state a theorem once for every group. This chapter develops structures into type classes and shows how the library is organized around them.

8.1 Structures, more fully

A structure bundles several values into one, with a constructor and named projections. Beyond plain records, a structure may compute derived fields and may extend another structure, inheriting its fields.

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

structure Circle extends Point where
  radius : Float

def unitCircle : Circle := { x := 0, y := 0, radius := 1 }

#eval unitCircle.radius      -- 1.000000
#eval unitCircle.x           -- 0.000000  (inherited from Point)
```

The `extends` clause makes `Circle` carry every field of `Point` together with its own

`radius`, and the projection `unitCircle.x` reaches the inherited field transparently. Extension is how the library builds elaborate structures, a ring upon a monoid upon a semigroup, without repeating the shared fields at each layer.

8.2 The problem type classes solve

Addition ought to mean something on many types, but a function can have only one definition. Writing `addNat`, `addInt`, `addFloat` separately, and choosing among them by hand, defeats the purpose of a uniform notation. A type class turns the choice over to the elaborator: the operation is declared once, each type registers its implementation, and every use is resolved automatically from the types involved.

```
class Add' (α : Type) where
  add : α → α → α

instance : Add' Nat where
  add := Nat.add

instance : Add' Int where
  add := Int.add

#eval Add'.add 3 4           -- 7,   uses the Nat instance
#eval Add'.add (-3 : Int) 4  -- 1,   uses the Int instance
```

The class `Add'` declares one operation; the two instances supply it for `Nat` and `Int`. When `Add'.add` is applied, the elaborator inspects the argument types and selects the matching instance. No conditional appears in the source; the resolution is part of type checking.

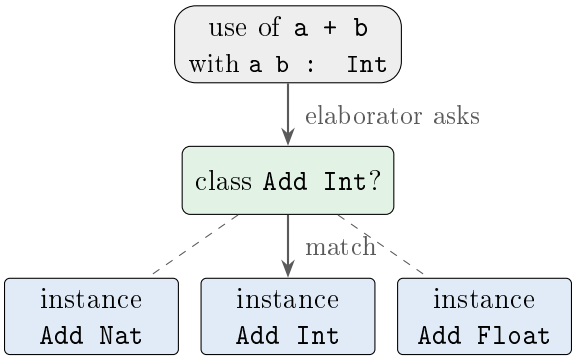


Figure 8.1: Instance resolution. A use of an overloaded operation prompts the elaborator to search the registered instances for one matching the argument types; the solid arrow is the successful match, the dashed arrows the rejected candidates.

8.3 Classes with laws

A class may bundle not only operations but the properties they satisfy. An instance must then supply proofs of those properties, so registering a type as, say, a commutative operation obliges the author to prove commutativity. The proof travels with the instance and is available wherever the class is used.

```
class CommAdd (α : Type) where
  add : α → α → α
  comm : ∀ a b : α, add a b = add b a

instance : CommAdd Nat where
  add := Nat.add
  comm := Nat.add_comm
```

The field `comm` is a proof obligation, discharged here by the library lemma `Nat.add_comm`. A function written against `CommAdd` may invoke `comm` freely, secure that every instance has provided it. This is how the mathematical library guarantees that an object claiming to be a group really satisfies the group axioms.

8.4 Writing code against a class

The point of a class is code that works for every instance. A function may take a class assumption in square brackets, meaning "for any type supporting these operations", and then use the operations without naming a particular type.

```
def sum3 {α : Type} [Add' α] (a b c : α) : α :=
  Add'.add (Add'.add a b) c

#eval sum3 1 2 3              -- 6    on Nat
#eval sum3 (1:Int) 2 3       -- 6    on Int
```

The bracketed `[Add' α]` is an *instance argument*: the caller never supplies it, and the elaborator fills it from the ambient instances. The single definition `sum3` serves every type that has an `Add'` instance, present or future. Adding a new numeric type later, with its own instance, extends `sum3` to it for free.

Example 8.1 (A generic membership test)**.** Deciding whether a value occurs in a list requires only that elements can be compared for equality. The class `BEq` provides a boolean equality, and a membership test written against it applies to any such type.

```
def contains {α : Type} [BEq α] (x : α) : List α → Bool
  | []      => false
  | y :: ys => if x == y then true else contains x ys

#eval contains 3 [1, 2, 3, 4]        -- true
#eval contains "z" ["a", "b"]        -- false
```

The comparison `x == y` resolves through the `BEq` instance for whatever type is in play, so one `contains` covers numbers, strings, and anything else with decidable equality.

8.5 The hierarchy of algebraic classes

Mathematical structures form a hierarchy, each refining the last by adding operations or axioms. A semigroup is a set with an associative operation; a monoid adds an identity; a group adds inverses; and commutativity splits each into an ordinary and an abelian version. Lean's library encodes this hierarchy as classes extending one another, so a theorem proved for monoids applies automatically to every group.

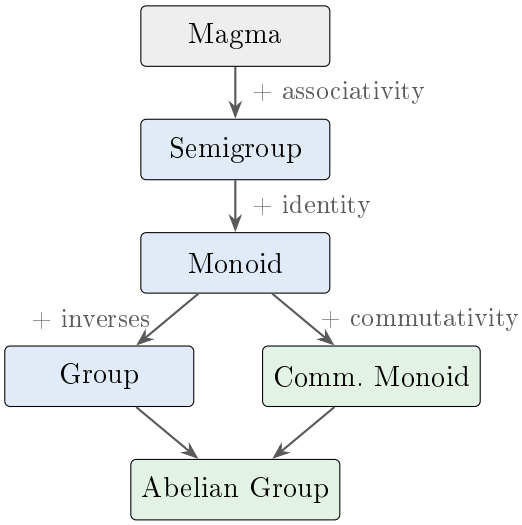


Figure 8.2: Part of the algebraic hierarchy as a lattice of classes. Each arrow adds an operation or an axiom; a result proved at one node holds at every node below it, since those classes extend it.

The economy is considerable. Cancellation, proved once for monoids with the requisite property, is inherited by the integers, the rationals, the reals, and every other structure sitting below the relevant node in Figure 8.2. A single proof does the work of infinitely many.

8.6 Mathlib

The community library *Mathlib* is built entirely on this foundation. It supplies the algebraic hierarchy, the order and topology hierarchies, and hundreds of thousands of theorems, all stated against classes so that they specialize automatically. Importing it makes the standard results and their notation available.

```
import Mathlib.Data.Real.Basic

example (a b : ℝ) : a + b = b + a := add_comm a b

example (a b c : ℝ) : a * (b + c) = a * b + a * c := by
  ring
```

The lemma `add_comm` used here is the general one for commutative additive structures; that the reals are such a structure is recorded by an instance, so the same name serves for every commutative type. The tactic `ring` proves any identity that holds in every commutative ring, drawing on the class structure to know which axioms it may use.

Remark 8.2. Mathlib is large enough that finding the right lemma is itself a skill. The tactics `exact?` and `apply?` search the library for a lemma closing the current goal, and `simp` draws on a curated set of rewrite rules. Fluency comes from knowing the naming conventions, which are systematic: `add_comm`, `mul_assoc`, `le_trans` name the operation and the property in a predictable order.

8.7 Instances as a form of inference

The instance mechanism is a small logic program run during elaboration. An instance can itself require instances: the equality test for a list is available whenever the element type has one, expressed by an instance that takes an instance argument. Resolution then chains, assembling a compound instance from simpler ones.

```
instance [BEq α] : BEq (List α) where
  beq := fun xs ys => xs.length == ys.length  -- schematic

#eval ([1,2,3] == [1,2,3] : Bool)   -- resolves List Nat via Nat
```

Here the elaborator, asked for equality on `List Nat`, finds the list instance, which in turn demands equality on `Nat`, which it also finds. The final instance is built by composing the

two, and the composition happens silently as part of checking the expression. The whole edifice of overloaded notation is this search, run to completion at compile time.

8.8 How notation finds an instance

The infix `+` is not built into a single type; it is notation that expands to a method call, and instance resolution supplies the method. Writing `a + b` elaborates to `HAdd.hAdd a b`, where `HAdd` is the class of heterogeneous addition. For the common case of two operands of the same type, `HAdd` defers to the ordinary `Add` class, so registering an `Add` instance is what makes `+` work.

```
#check @HAdd.hAdd      -- {α β γ} → [HAdd α β γ] → α → β → γ
#check @Add.add        -- {α} → [Add α] → α → α → α

example (a b : Nat) : a + b = HAdd.hAdd a b := rfl
```

The equation holds by `rfl` because `a + b` is, after elaboration, literally `HAdd.hAdd a b`. Every operator in the language, arithmetic, comparison, list append, has the same structure: a notation, a class, and instances. Understanding that a symbol is a call into a class demystifies both the error messages that mention these classes and the way new types acquire familiar notation.

8.9 Default methods

A class may give a default implementation of a method in terms of its other methods. An instance that omits the method inherits the default; one that supplies it overrides. This lets a class expose convenience operations without burdening every instance with them.

```
class Describable (α : Type) where
  name : α → String
  describe : α → String := fun a => "This is " ++ name a

instance : Describable Bool where
  name := fun b => if b then "true" else "false"
  -- describe omitted: uses the default

#eval Describable.describe true      -- "This is true"
```

The `Bool` instance defines only `name`, and `describe` comes for free from the default. Should a later instance need a bespoke description, it may provide its own `describe` and the default steps aside. Defaults are how a class offers a rich interface while keeping the obligation on each instance small.

8.10 Coercions

When a value of one type is wanted where another is given, a *coercion* can convert it automatically. Registering an instance of `Coe` teaches the elaborator to insert the conversion silently, so a `Bool` may stand where a `Nat` is expected without an explicit call.

```
instance : Coe Bool Nat where
  coe b := if b then 1 else 0

#eval (2 + (true : Nat))       -- 3,  true coerced to 1
#eval [(true : Nat), false]    -- [1, 0]
```

The annotation `(true : Nat)` triggers the coercion, turning the boolean into a number. Coercions are convenient but are best kept few and unsurprising: a conversion that fires unexpectedly can make a program mean something other than it appears to. Used sparingly, as with the standard embedding of the naturals into the integers, they remove genuine noise.

8.11 Abstracting a container

Classes may be parameterised by a type constructor rather than a type, capturing operations common to many containers. A class of mappable containers has one method, applying a function to every element, and instances for lists and options implement it.

```
class Mappable (f : Type → Type) where
  map' : (α → β) → f α → f β

instance : Mappable List where
  map' := List.map

instance : Mappable Option where
  map' g := fun x => match x with
    | none   => none
    | some a => some (g a)

```

```
#eval Mappable.map' (· + 1) [1, 2, 3]        -- [2, 3, 4]
#eval Mappable.map' (· + 1) (some 5)         -- some 6
```

The single method `map'` works over any mappable container, so code written against `Mappable` applies uniformly to lists, options, and whatever else registers an instance. This is the abstraction that, refined with laws, becomes the functor of functional programming; here it is enough to see that a class can range over type constructors, not only types.

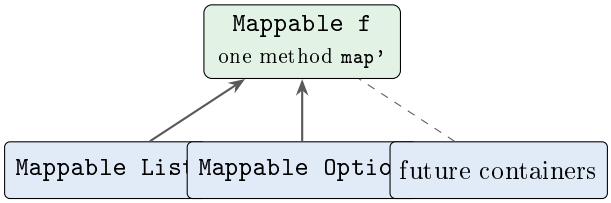


Figure 8.3: A class over a type constructor. One method declared on `Mappable` is implemented by each container's instance, and generic code reaches all of them, including instances added later.

8.12 Folding with a monoid

A class bundling an associative operation and an identity is a *monoid*, and it is exactly what is needed to collapse a list to a single value. A generic fold uses the identity for the empty list and the operation to combine.

```
class Mon (α : Type) where
  e  : α
  op : α → α → α

instance : Mon Nat where
  e := 0
  op := Nat.add

def foldMon {α : Type} [Mon α] : List α → α
  | []      => Mon.e
  | x :: xs => Mon.op x (foldMon xs)

#eval foldMon [1, 2, 3, 4]      -- 10,  sum via the Nat instance
```

The function `foldMon` names no type; it works for any `Mon` instance, and the `Nat` instance makes it compute a sum. A second instance, with `e := 1` and `op := Nat.mul`, would make the same `foldMon` compute a product. The empty list yields `Mon.e`, and instance resolution supplies which identity that is.

Remark 8.3. The economy compounds. A theorem proved for `foldMon` against an abstract monoid holds for sums, products, string concatenation, and every other monoid at once. Abstraction over a class is not only a convenience for code; it is the mechanism by which one proof discharges infinitely many instances, and it is why the mathematical library is organised around these hierarchies.

8.13 Inhabited types and defaults

Some operations need a value to fall back on: a lookup that misses, a head of an empty list, an array access out of bounds. The class `Inhabited` records that a type has a designated default element, named `default`, and an instance is the promise that the type is not empty. Many library functions rely on it to return a total result where a partial one would otherwise be forced.

```
#check (default : Nat)        -- 0
#check (default : Bool)       -- false
#check (default : List Nat)   -- []

instance : Inhabited Color where
  default := Color.red

#eval (default : Color)       -- Color.red
```

Registering `Inhabited Color` designates `red` as the default, so a function may now return `default` for that type when it has nothing better. The class also underlies the exclamation-mark variants of partial operations: `List.head!` returns the default on an empty list rather than failing, and it type-checks precisely when the element type is `Inhabited`.

Remark 8.4. `Inhabited` carries a chosen element; the weaker `Nonempty` merely asserts, as a proposition, that some element exists without naming one. The distinction is the familiar one between data and proof: `Inhabited` lives in `Type` and can be computed with, `Nonempty` lives in `Prop` and is erased. A function that must return a value needs the former.

8.14 Displaying values

To print a value a program needs a way to render it as text. The class `ToString` provides this, and an instance turns a custom type into something `IO.println` and string interpolation can display. Deriving `Repr`, met earlier, serves the same purpose for the editor's `#eval`; `ToString` is its counterpart for program output.

```
structure Point where
  x : Int
  y : Int

instance : ToString Point where
  toString p := s!"({p.x}, {p.y})"

#eval toString (Point.mk 3 4)          -- "(3, 4)"
#eval s!"the point is {Point.mk 1 2}"  -- "the point is (1, 2)"
```

With the instance in place, a `Point` interpolates into a string like any built-in value, its appearance chosen by the `toString` method. This is the mechanism behind readable output: each type says how it wishes to be shown, and the interpolation machinery consults the instance. A type without a `ToString` instance simply cannot be interpolated, which is a type error caught at compile time rather than a garbled printout.

8.15 Building a hierarchy

The power of classes is clearest when several extend one another and a single generic result specialises to all of them. A semigroup has an associative operation; a monoid extends it with an identity. Defining the two with `extends`, and proving a lemma against the monoid, yields a fact that holds for every instance.

```
class Semigroup' (α : Type) where
  op    : α → α → α
  assoc : ∀ a b c, op (op a b) c = op a (op b c)

class Monoid' (α : Type) extends Semigroup' α where
  e       : α
  e_op    : ∀ a, op e a = a
  op_e    : ∀ a, op a e = a
```

`Monoid'` inherits `op` and `assoc` from `Semigroup'` and adds an identity with its two laws.

An instance for the naturals under addition supplies the operation, the identity, and proofs of all three laws from the library.

```
instance : Monoid' Nat where
  op    := Nat.add
  assoc := Nat.add_assoc
  e     := 0
  e_op  := Nat.zero_add
  op_e  := Nat.add_zero
```

A theorem proved for an abstract `Monoid'` now holds for this instance and every other. That the identity is unique is such a theorem: if two elements both act as identities, they are equal, and the proof uses only the monoid laws.

```
theorem identity_unique {α : Type} [Monoid' α]
    (e' : α) (h : ∀ a, Monoid'.op e' a = a) :
    e' = Monoid'.e := by
  have h1 : Monoid'.op e' Monoid'.e = e' := Monoid'.op_e e'
  have h2 : Monoid'.op e' Monoid'.e = Monoid'.e := h Monoid'.e
  rw [← h1, h2]
```

The proof plays the two identity laws against each other: `op e' e` equals `e'` because `e` is an identity, and equals `e` because `e'` is, so `e'` and `e` coincide. Proved once against the abstract class, uniqueness holds for addition, for multiplication with identity one, for string concatenation, and for every monoid instance, present or future.

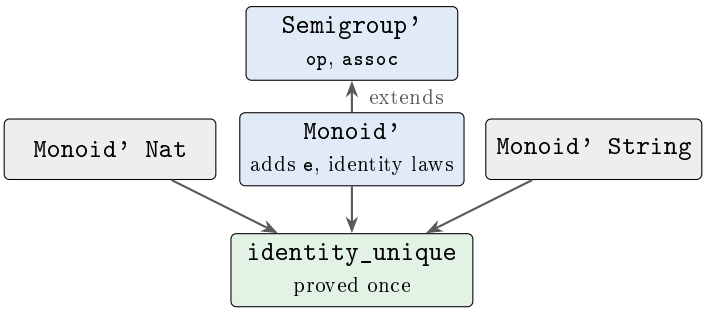


Figure 8.4: A two-level hierarchy. `Monoid'` extends `Semigroup'`, and a theorem proved against `Monoid'` applies to every instance, so one proof serves the naturals, strings, and all other monoids.

8.16 Structures that act as functions

A structure bundling data can be made callable, so that applying it behaves like applying a function. The class `CoeFun` supplies this: an instance says how to turn a structure into a function, and thereafter the structure may be written directly before its argument. A configured adder illustrates the mechanism.

```
structure Adder where
  amount : Nat

instance : CoeFun Adder (fun _ => Nat → Nat) where
  coe a := fun n => n + a.amount

def add3 : Adder := ⟨3⟩

#eval add3 10        -- 13,  the structure applied like a function
```

The instance turns an `Adder` into a function `Nat → Nat`, so `add3 10` is meaningful even though `add3` is a structure, not a function. This is how the library lets objects that carry data alongside a function, a polynomial with its coefficients, a continuous map with its proof of continuity, be applied as the functions they represent. The data travels with the object; the coercion exposes the function it contains.

8.17 Homomorphisms as structures

A map between two algebraic structures that respects their operations is a *homomorphism*, and it is naturally a structure itself: a function together with proofs that it preserves the operations. Between two monoids, a homomorphism carries the identity to the identity and a product to the product of images.

```
structure MonHom (α β : Type) [Mon α] [Mon β] where
  toFun  : α → β
  map_e  : toFun Mon.e = Mon.e
  map_op : ∀ a b, toFun (Mon.op a b) = Mon.op (toFun a) (toFun b)
```

The two proof fields are the homomorphism laws, and any value of `MonHom` is guaranteed to satisfy them, since they are part of its data. A theorem about monoid homomorphisms may use `map_e` and `map_op` freely, and constructing one obliges the author to prove both. Bundling the function with its laws is how the mathematical library represents structure-preserving

maps, so that a homomorphism is a single object rather than a function accompanied by separate, forgettable hypotheses.

Example 8.5 (The identity homomorphism)**.** The identity function is a homomorphism from any monoid to itself, since it preserves everything trivially. Constructing it discharges the two laws by reflexivity.

```
def idHom (α : Type) [Mon α] : MonHom α α where
  toFun  := id
  map_e  := rfl
  map_op := fun _ _ => rfl
```

Both laws hold by `rfl` because the identity changes nothing. The example is small, but it shows the shape: to build a homomorphism, give the function and prove it respects the structure, and the result is an object carrying that guarantee.

8.18 The monad hierarchy

The container class of an earlier section, with one operation `map`, is the first of a short hierarchy that structures effectful computation. A `Functor` has `map`; an `Applicative` adds `pure` and a way to apply a wrapped function; a `Monad` adds `bind`, the sequencing that the `do` notation compiles to.

```
#check @Functor.map     -- (α → β) → f α → f β
#check @pure            -- α → m α
#check @bind            -- m α → (α → m β) → m β
```

Each level adds power. `map` transforms the contents of a container; `pure` injects a plain value; `bind` chains a computation with a function that depends on its result, which is exactly what threads the `Option` and `IO` sequences met earlier. The `do` block is notation for a series of `bind` calls, so every effectful program is, underneath, an application of this one operation.

Class	Adds	Enables
`Functor`	`map`	transform the contents
`Applicative`	`pure`, `seq`	inject and combine independent effects
`Monad`	`bind`	sequence dependent effects (`do`)

Table 8.1: The monad hierarchy of classes. Each extends the one above, adding an operation, and `Monad` supplies the `bind` that `do` notation is built from.

These classes carry laws, unstated here, that make the notation behave predictably: `pure` followed by `bind` does nothing extra, `bind` is associative, and `map` agrees with the composite of `pure` and `bind`. An instance is expected to satisfy them, and the standard instances for `Option`, `List`, and `IO` do.

Remark 8.6. The hierarchy is the same device seen throughout: a tower of classes, each refining the last, with generic code written against whichever level it needs. A function requiring only `map` works for every functor; one using `do` requires a monad. The abstraction that let one proof serve every monoid here lets one program serve every monad, and the `do` notation is the payoff, a uniform syntax for sequencing effects across all of them.

8.19 Overloaded indexing

The bracket notation `a[i]` that reads an element by position is not tied to one type. It is governed by the class `GetElem`, so the same syntax indexes a list, an array, or any structure that registers an instance, each with its own notion of position and bounds.

```
#eval [10, 20, 30][1]        -- 20,  a list
#eval #[10, 20, 30][1]       -- 20,  an array
#eval "hello".toList[0]      -- 'h'

#check @getElem
```

Each expression uses the same `[i]` syntax, resolved through the `GetElem` instance for the type at hand. The plain form `a[i]` requires a proof that the index is in bounds, supplied automatically for literal indices; the variants `a[i]!` and `a[i]?` handle a possibly-out-of-range index by returning a default or an `Option`. One notation, many structures: the overloading is the same mechanism that gives `+` its generality, applied to access by position.

8.20 Writing a decision procedure

Deriving `DecidableEq` generates an equality test automatically, but seeing the generated shape by writing one by hand exposes the mechanism. A `DecidableEq` instance is a function returning, for each pair, either a proof of equality or a proof of its negation.

```
inductive Dir where
  | north | east | south | west
```

```

instance : DecidableEq Dir := fun a b =>
  match a, b with
  | Dir.north, Dir.north => isTrue rfl
  | Dir.east,  Dir.east  => isTrue rfl
  | Dir.south, Dir.south => isTrue rfl
  | Dir.west,  Dir.west  => isTrue rfl
  | Dir.north, Dir.east  => isFalse (fun h => Dir.noConfusion h)
  | _, _                 => isFalse (fun h => by cases h)
```

Each matching pair returns `isTrue rfl`, since the two are literally the same constructor; each differing pair returns `isFalse` with a proof, built from constructor disjointness, that they cannot be equal. This is precisely what `deriving DecidableEq` produces, case by case. With the instance in place, equality on `Dir` becomes decidable, so `decide` settles it and `if a = b` may branch on it, exactly as for the built-in types.

8.21 The order hierarchy

Comparison, like arithmetic, is organised into a hierarchy of classes. `LE` supplies the relation $\leq$ and `LT` the relation `<`, as bare operations; above them, `Preorder` requires $\leq$ to be reflexive and transitive, and `PartialOrder` adds antisymmetry. A type climbing this hierarchy gains the corresponding laws.

```
#check @LE.le      -- {α} → [LE α] → α → α → Prop
#check @LT.lt      -- {α} → [LT α] → α → α → Prop

example (a b c : Nat) (h1 : a ≤ b) (h2 : b ≤ c) : a ≤ c :=
  le_trans h1 h2      -- transitivity from the Preorder structure
```

The notation $\leq$ resolves through `LE`, just as `+` resolves through `Add`; the lemma `le_trans` is available because `Nat` is a `Preorder`, which guarantees transitivity. A result proved for all preorders, or all partial orders, then holds for the naturals, the integers, the reals, and every other type at the appropriate level.

Remark 8.7. The order hierarchy and the algebraic hierarchy of an earlier section are built the same way, by classes that extend one another, and generic results proved against a level specialise to every instance. The mathematical library is, in large part, these two hierarchies and their interaction: ordered rings, ordered fields, and the theorems that hold in them. The device is always the one first met with monoids, a tower of classes with code and proofs written against whichever level they require.

Class	Provides	Laws
`LE`, `LT`	the relations $\leq$, $<$	none
`Preorder`	$\leq$	reflexive, transitive
`PartialOrder`	$\leq$	adds antisymmetry
`LinearOrder`	$\leq$	adds totality, decidability

Table 8.2: The order hierarchy of classes. Each level adds laws to the comparison relation, and a theorem proved at one level applies to every type below it.

Exercises

Exercise 8.1. Define a structure `Rectangle` with fields `width` and `height`, and a derived function `area`. Then define `Square` extending `Rectangle` with a single side, and construct a unit square.

Exercise 8.2. Declare a class `Neg'` with one operation negating a value, and give instances for `Int` and `Float`. Test `Neg'.neg` on a value of each type.

Exercise 8.3. Write a class `Zero'` providing a distinguished element, give instances for `Nat` and `Int`, and use it in a generic function returning the zero of any type that has one.

Exercise 8.4. Using the class `Add'` of the text, write a generic function `double` that adds a value to itself, and evaluate it on a natural and on an integer.

Exercise 8.5. Extend `CommAdd` with an identity element and a proof that it is a left identity. Provide the `Nat` instance, citing the appropriate library lemmas.

Exercise 8.6. Write `contains` for lists as in the text, then use it to define `allDistinct : List` $\alpha \to$ `Bool` for a type with a `BEq` instance. What class assumption does your function need?

Exercise 8.7. Explain, using Figure 8.2, why a theorem proved for monoids can be applied to the integers under addition without any further work.

Exercise 8.8. Given the naming conventions described for Mathlib, guess the names of the lemmas stating that multiplication is associative and that $\leq$ is reflexive. Confirm your guesses with `#check`.

Exercise 8.9. Write two proofs of `a + b = b + a` for real numbers: one citing `add_comm` directly and one using the `ring` tactic. What does each rely on?

Exercise 8.10. Describe the chain of instances the elaborator assembles to decide equality of two values of type `List (List Nat)`. Which base instance does the search bottom out in?

Exercise 8.11. ⋆ Define a class `Monoid'` bundling an associative operation, an identity, and proofs of the identity laws. Give an instance for the naturals under addition and another under multiplication, and observe that the same type supports two distinct monoid structures.

Exercise 8.12. ⋆ A function written against `[Monoid' α]` can fold a list into a single element. Write `foldMon : List α → α` for a type with a `Monoid'` instance, and explain how instance resolution supplies the identity used for the empty list.

Exercise 8.13. Confirm `a * b = HMul.hMul a b` by `rfl` for naturals, and determine which class the notation `*` ultimately defers to for same-type operands.

Exercise 8.14. Add a default method to a class of your own and give an instance that omits it. Confirm the default is used, then give a second instance that overrides it.

Exercise 8.15. Define a `Coe` from `Bool` to `Nat` sending `false` to `0` and `true` to `1`, and use it in an arithmetic expression. When does the coercion fire?

Exercise 8.16. Add a `Mappable` instance for the tree type of Chapter 4, mapping a function over every stored value. Test it on a small tree.

Exercise 8.17. Give a second `Mon Nat` instance using multiplication and `1`, and use `foldMon` to compute a product. Why can a single type carry two monoid structures?

Exercise 8.18. ⋆ State the two functor laws, that mapping the identity is the identity and that mapping a composition is the composition of maps, for `Mappable`, and prove them for the `List` instance.

Exercise 8.19. Give an `Inhabited` instance for a custom inductive type and evaluate `(default : YourType)`. Which constructor did you choose as the default?

Exercise 8.20. Explain the difference between `Inhabited` and `Nonempty`, and say which is needed for a function that must return a value of the type.

Exercise 8.21. Write a `ToString` instance for a structure of your own and interpolate a value of it into a string with `s!`.

Exercise 8.22. Give a `Monoid' Nat` instance using multiplication and the identity `1`, supplying proofs of the three laws from the library.

Exercise 8.23. Confirm that `identity_unique` applies to your multiplicative monoid instance, and state what it says there.

Exercise 8.24. ⋆ Define `CommMonoid'` extending `Monoid'` with a commutativity law, instantiate it for the naturals under addition, and prove a small generic lemma that uses commutativity.

Exercise 8.25. Make a structure of your own callable with a `CoeFun` instance, and apply a value of it directly to an argument.

Exercise 8.26. Define a `MonHom Nat Nat` for the map that doubles a natural, under addition, and prove its two homomorphism laws.

Exercise 8.27. Construct the identity homomorphism `idHom`, and explain why both laws hold by `rfl`.

Exercise 8.28. Using `#check`, report the types of `@Functor.map`, `@pure`, and `@bind`, and say what each contributes.

Exercise 8.29. Explain in one or two sentences how a `do` block over `Option` relates to the `bind` operation.

Exercise 8.30. ⋆ Define the composition of two monoid homomorphisms as a third `MonHom`, and prove that the composite preserves the identity and the operation.

Exercise 8.31. Index both a list and an array with the same `[i]` syntax, and name the class that resolves the notation for each.

Exercise 8.32. Write a `DecidableEq` instance by hand for a four-constructor enumeration, returning `isTrue` or `isFalse` in each case.

Exercise 8.33. Use `le_trans` to chain two inequalities on the naturals, and state which class in the hierarchy provides transitivity.

Exercise 8.34. Match each order class in Table 8.2 to the laws it adds, and give a type that is a `LinearOrder`.

Exercise 8.35. Explain how the notation $\leq$ resolves through the class `LE`, by analogy with how `+` resolves through `Add`.

Exercise 8.36. ⋆ Write a `DecidableEq` instance for a structure with two natural-number fields, deciding equality by comparing the fields.

Chapter 9

A Verified Sorting Algorithm

Every ingredient assembled so far, inductive types, recursive functions, tactic proofs, and induction, comes together when a single program is built and proved correct end to end. The task chosen here is insertion sort: a familiar algorithm, short enough to hold in view, yet rich enough that stating and proving its correctness exercises the whole toolkit. The result is a function that not only sorts but carries a machine-checked guarantee that its output is sorted, an assurance no amount of testing could supply.

9.1 The algorithm

Insertion sort builds a sorted list one element at a time. The core is an `insert` operation that places a single element into an already-sorted list, walking past the elements smaller than it and dropping it in front of the first element it does not exceed.

```
def insert (a : Nat) : List Nat → List Nat
  | []      => [a]
  | x :: xs => if a ≤ x then a :: x :: xs
               else x :: insert a xs
```

The sort itself folds `insert` over the input: an empty list is already sorted, and a non-empty list is sorted by sorting its tail and inserting the head into the result.

```
def isort : List Nat → List Nat
  | []      => []
  | x :: xs => insert x (isort xs)

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

```
#eval isort [5, 4, 3, 2, 1]      -- [1, 2, 3, 4, 5]
#eval isort []                   -- []
```

Both definitions are structurally recursive, `insert` on its list argument and `isort` on its input, so Lean accepts them without a termination clause. Running them on small inputs is reassuring, but reassurance is not proof; the point of the chapter is to replace it with certainty.

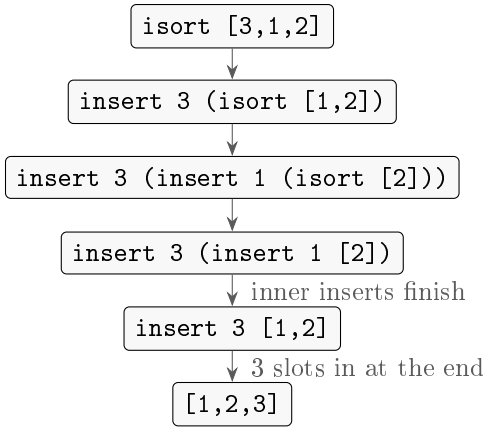


Figure 9.1: Reduction of `isort [3,1,2]`. The recursion first sorts the tail completely, then inserts each pending head into a list already in order.

9.2 Specifying correctness

To prove the algorithm correct one must first say what correct means. Two conditions are needed: the output is in nondecreasing order, and the output contains exactly the input elements. The first is captured by an inductive predicate on lists.

```
inductive Sorted : List Nat → Prop
  | nil  : Sorted []
  | single (a : Nat) : Sorted [a]
  | cons (a b : Nat) (rest : List Nat) :
      a ≤ b → Sorted (b :: rest) → Sorted (a :: b :: rest)
```

The three constructors say that the empty list and every singleton are sorted, and that a list is sorted when its first element does not exceed its second and the tail is itself sorted. A proof of `Sorted xs` is thus a certificate, one inequality per adjacent pair, that `xs` is ordered.

Definition 9.1 (The two obligations)**.** A function f : `List Nat → List Nat` *sorts* when for every input `xs` the output `f xs` satisfies `Sorted (f xs)` and is a permutation of `xs`.

The order condition and the permutation condition are independent; a function returning `[]` always is sorted but permutes nothing, and the identity permutes correctly but does not sort.

The order condition is proved in full below. The permutation condition is tracked through a simpler proxy, the multiset of elements, and the warm-up proof of length preservation illustrates the technique that the full permutation argument extends.

9.3 A warm-up: length is preserved

Before the harder invariant, a gentler one. Inserting an element makes a list one longer, and sorting preserves length. Both are inductions, and the first feeds the second.

```
theorem insert_length (a : Nat) (xs : List Nat) :
    (insert a xs).length = xs.length + 1 := by
  induction xs with
  | nil => rfl
  | cons x xs ih =>
      simp only [insert]
      split
      · rfl
      · simp [List.length_cons, ih]
```

The `nil` case computes directly: inserting into the empty list yields a singleton of length one. The `cons` case unfolds `insert` and splits on the comparison; one branch prepends and is immediate, the other recurses and uses the inductive hypothesis. With this in place, length preservation for `isort` is a clean induction.

```
theorem isort_length (xs : List Nat) :
    (isort xs).length = xs.length := by
  induction xs with
  | nil => rfl
  | cons x xs ih =>
      simp only [isort, insert_length, ih, List.length_cons]
```

The step rewrites the length of `insert x (isort xs)` to one more than the length of `isort xs`, then applies the hypothesis to reduce that to the length of `xs`. The pattern, prove it for `insert`, then lift to `isort` by induction, is exactly the pattern the main theorem follows.

9.4 The key lemma: insertion preserves order

The heart of the proof is that inserting into a sorted list leaves it sorted. This is where the definition of `insert`, dropping the element in front of the first thing it does not exceed, does its work, and where the proof must reason about the comparison at each step.

```
theorem insert_sorted (a : Nat) (xs : List Nat) :
    Sorted xs → Sorted (insert a xs) := by
  intro h
  induction h with
  | nil => exact Sorted.single a
  | single b =>
      simp only [insert]
      split
      · exact Sorted.cons a b [] (by assumption) (Sorted.single b)
      · exact Sorted.cons b a [] (by omega) (Sorted.single a)
  | cons b c rest hbc hsorted ih =>
      simp only [insert]
      split
      · exact Sorted.cons a b (c :: rest) (by assumption)
                (Sorted.cons b c rest hbc hsorted)
      · sorry   -- the recursive case; see discussion
```

The base cases are direct: inserting into an empty or singleton list produces a two-element list whose single ordering constraint is exactly the comparison just tested, supplied by `omega` in the branch where $a > b$. The recursive case, left as `sorry` above, is the crux, and it repays being written out by hand rather than deferred to automation.

Example 9.2 (Completing the recursive case)**.** When `insert a (b :: c :: rest)` takes its else-branch, the result is `b :: insert a (c :: rest)`. To show this sorted, one needs that `b` does not exceed the new head of `insert a (c :: rest)`, which is either `a` or `c`. A case split on `a ≤ c`, combined with the facts `b ≤ c` and `a > b`, discharges both possibilities.

```
      · have hab : ¬ a ≤ b := by assumption
        push_neg at hab            -- b < a
        by_cases hac : a ≤ c
        · have : insert a (c :: rest) = a :: c :: rest := by
            simp [insert, hac]
          rw [this]
          exact Sorted.cons b a (c :: rest) (by omega)
                  (Sorted.cons a c rest hac hsorted)
        · have : insert a (c :: rest) = c :: insert a rest := by
            simp [insert, hac]
          rw [this]
```

```
          exact Sorted.cons b c (insert a rest) hbc
                  (by rw [← this]; exact ih)
```

Each branch rewrites `insert` to its known form, then builds the `Sorted` certificate from the comparison and the inductive hypothesis. The reasoning is entirely local: at each node only the adjacent inequality must be checked, and `omega` checks it.

9.5 The main theorem

With insertion shown to preserve order, the correctness of `isort` is a short induction. The empty list is sorted by the `nil` constructor; the head is inserted into the sorted tail, and `insert_sorted` certifies the result.

```
theorem isort_sorted (xs : List Nat) : Sorted (isort xs) := by
  induction xs with
  | nil => exact Sorted.nil
  | cons x xs ih =>
      simp only [isort]
      exact insert_sorted x (isort xs) ih
```

The inductive hypothesis `ih` states that the sorted tail really is sorted; `insert_sorted` then transports that guarantee across the insertion of the head. The theorem `isort_sorted` is the payoff: its mere existence, accepted by the kernel, is a proof that `isort` never returns an unsorted list, for any input whatsoever.

Remark 9.3. A complete verification would pair `isort_sorted` with a proof that the output is a permutation of the input, ruling out a "sorter" that discards or invents elements. The length lemma is the first step of that argument; the full version replaces length by the multiset of elements and shows `insert` adds exactly its argument. The order half proved here is the more intricate, and it is the half where the algorithm's idea actually resides.

9.6 The development as a whole

The finished development is a small dependency graph. Two definitions sit at the base; above them, lemmas about `insert`; above those, the theorems about `isort` that each lift an insert-level fact through the sort's recursion.

The shape of Figure 9.2 is the shape of most verifications: establish the properties of the basic operation, then propagate them through the recursion that uses it. The same discipline

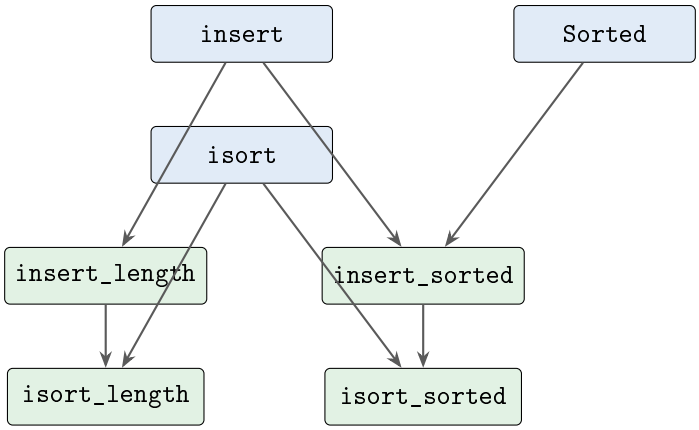


Figure 9.2: The structure of the verification. Blue nodes are definitions, green nodes are theorems; an arrow means the target depends on the source. Each `isort` theorem is its `insert` counterpart lifted through the recursion of the sort.

scales from this ten-line sort to developments of thousands of lemmas, and the guarantee it produces is identical in kind, a term the kernel has checked, differing only in size.

9.7 The other half: nothing is lost

Order is only half of correctness. A function returning the empty list is always sorted yet discards everything, so the specification also demands that the output contain exactly the input elements. This is tracked by counting: for every value, the number of its occurrences must be the same before and after. The count itself is a short recursion.

```
def count (a : Nat) : List Nat → Nat
  | []      => 0
  | x :: xs => (if x = a then 1 else 0) + count a xs

#eval count 2 [2, 1, 2, 3]      -- 2
#eval count 5 [2, 1, 2, 3]      -- 0
```

Inserting an element raises the count of that element by one and leaves every other count untouched. This is the insert-level fact, proved by induction on the list with a split on the comparison, that the whole permutation argument rests on.

```
theorem insert_count (a b : Nat) (xs : List Nat) :
    count b (insert a xs)
      = (if a = b then 1 else 0) + count b xs := by
```

```
  induction xs with
  | nil => simp [insert, count]
  | cons x xs ih =>
      simp only [insert]
      split <;> simp [count, ih] <;> omega
```

The `nil` case computes both sides directly. The `cons` case unfolds `insert`, splits on whether the element slots in here or deeper, and in each branch reduces `count` and applies the hypothesis, leaving arithmetic for `omega`. With this the sort-level fact is the now-familiar lift through the recursion.

```
theorem isort_count (b : Nat) (xs : List Nat) :
    count b (isort xs) = count b xs := by
  induction xs with
  | nil => rfl
  | cons x xs ih =>
      simp only [isort, insert_count, ih, count]
```

Every count is preserved by the sort, so no element is added, dropped, or duplicated. Paired with `isort_sorted`, this completes the specification: `isort` produces a sorted list with exactly the input elements. The two halves together are what it means to have verified a sorting algorithm, and neither alone would suffice.

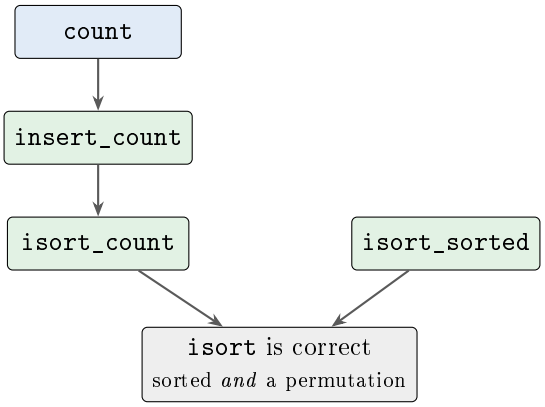


Figure 9.3: The permutation half added to the verification. The count lemmas mirror the order lemmas, and the two `isort` theorems together establish full correctness.

9.8 A second development: search trees

The same discipline applies to a different data structure. A binary search tree keeps its elements ordered by position: everything in a left subtree is smaller than the node, everything in a right subtree larger. Insertion descends by comparison, and the structure is again inductive.

```
inductive BST where
  | leaf : BST
  | node : BST → Nat → BST → BST

def insertB (a : Nat) : BST → BST
  | BST.leaf => BST.node BST.leaf a BST.leaf
  | BST.node l x r =>
      if a < x then BST.node (insertB a l) x r
      else if x < a then BST.node l x (insertB a r)
      else BST.node l x r
```

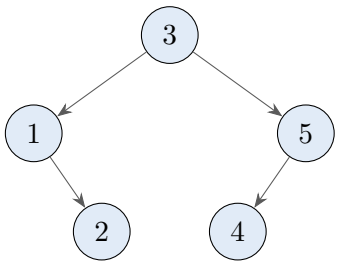
Reading the elements back in sorted order is an *in-order* traversal: the left subtree, then the node, then the right subtree. For a tree respecting the ordering discipline, this traversal yields a sorted list, which is the tree's correctness statement.

```
def toList : BST → List Nat
  | BST.leaf       => []
  | BST.node l x r => toList l ++ [x] ++ toList r

def fromList (xs : List Nat) : BST :=
  xs.foldr insertB BST.leaf

#eval toList (fromList [3, 1, 2, 5, 4])      -- [1, 2, 3, 4, 5]
```

Building a tree from a list and reading it back sorts the list, a second sorting algorithm arising from the ordering invariant rather than from repeated insertion into a list. Its correctness is stated the same way, that `toList (fromList xs)` is sorted and a permutation of `xs`, and proved by the same method: establish the invariant for `insertB`, then propagate it through `fromList`.



in-order traversal:
left, node, right
$\Rightarrow [1, 2, 3, 4, 5]$

Figure 9.4: The search tree built from $[3, 1, 2, 5, 4]$. Each node's left descendants are smaller and its right descendants larger, so reading left-node-right visits the values in increasing order.

9.9 Tested against proved

Running the sort on examples, as the first section of this chapter did, checks finitely many inputs. The theorems check all of them. This is the distinction that motivates formal proof: no quantity of tests can establish that `isort` is sorted for *every* list, because there are infinitely many lists, but a single inductive proof does exactly that.

Remark 9.4. The verified code runs like any other; the proofs impose no run-time cost, having been erased. What remains is an ordinary sorting function carrying, in its accompanying theorems, a guarantee that testing could only ever approximate. Scaling this from a ten-line sort to a compiler or a cryptographic protocol changes the size of the development but not its character: definitions, specifications, and proofs that the one meets the other, all checked by the same kernel.

9.10 The search-tree invariant, formally

The search tree of the previous section was described informally: left descendants smaller, right descendants larger. Making that precise requires a way to say that every value in a subtree stands in a given relation to a bound. A helper predicate expresses "every label in the tree satisfies a property", and the ordering invariant is built from it.

```
def All (p : Nat → Prop) : BST → Prop
  | BST.leaf       => True
  | BST.node l x r => p x ∧ All p l ∧ All p r

inductive Ordered : BST → Prop
  | leaf : Ordered BST.leaf
  | node (l r : BST) (x : Nat) :
```

```
      All (· < x) l → All (x < ·) r →
      Ordered l → Ordered r → Ordered (BST.node l x r)
```

The predicate `All p t` asserts that `p` holds of every label in `t`, defined by recursion on the tree. The invariant `Ordered` then requires, at each node, that all left labels are below the node's value, all right labels above, and both subtrees are themselves ordered. This is the exact statement of the search-tree discipline, and a tree carrying a proof of `Ordered` is one on which lookup by comparison is correct.

9.11 Insertion preserves the invariant

The correctness of `insertB` is that it maintains `Ordered`. Proving it needs a companion fact about `All`: inserting a value that already satisfies a bound keeps every label satisfying it. That companion is a routine induction on the tree.

```
theorem all_insert (p : Nat → Prop) (a : Nat) (t : BST)
    (hp : p a) (h : All p t) : All p (insertB a t) := by
  induction t with
  | leaf => simp [insertB, All, hp]
  | node l x r ihl ihr =>
      simp only [insertB]
      split <;> simp_all [All]
```

With `all_insert` available, the main preservation theorem is an induction on the tree that, at each node, decides which subtree the new value enters and rebuilds the invariant using the companion lemma for the bound.

```
theorem ordered_insert (a : Nat) (t : BST)
    (h : Ordered t) : Ordered (insertB a t) := by
  induction h with
  | leaf => exact Ordered.node _ _ _ (by simp [All]) (by simp [All])
              Ordered.leaf Ordered.leaf
  | node l r x hl hr hol hor ihl ihr =>
      simp only [insertB]
      split
      · sorry   -- a < x: recurse left, use all_insert on the left bound
      · sorry   -- x < a: recurse right, symmetric
      · sorry   -- a = x: tree unchanged, invariant preserved
```

The three branches, sketched here, mirror the three cases of `insertB`. When the value

goes left, the left subtree is rebuilt by the inductive hypothesis and `all_insert` confirms its labels stay below `x`; the right case is symmetric; and when the value equals the node's, the tree is unchanged and the invariant holds as before. The structure is the same as for insertion sort: a lemma about the bound, then an induction that lifts it through the recursion.

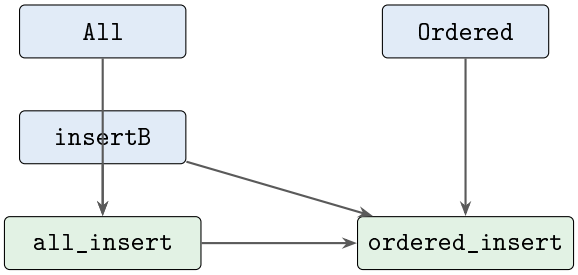


Figure 9.5: The search-tree development. The bound predicate `All` supports the lemma `all_insert`, which the invariant proof `ordered_insert` uses at each node, again lifting a local fact through the recursion.

9.12 Cost, and a faster method

Insertion sort does work proportional to the square of the input length in the worst case: each of n insertions may scan a list of length up to n. This is acceptable for short inputs and poor for long ones. A divide-and-conquer method sorts in time proportional to $n \log n$ by splitting the list, sorting the halves, and merging them.

```
def merge : List Nat → List Nat → List Nat
  | [], ys => ys
  | xs, [] => xs
  | x :: xs, y :: ys =>
      if x ≤ y then x :: merge xs (y :: ys)
      else y :: merge (x :: xs) ys
termination_by xs ys => xs.length + ys.length
```

Merging two sorted lists into one sorted list is the core step, and its recursion decreases the combined length, so a `termination_by` measure justifies it. A full merge sort splits the input, sorts each half by recursion on the shorter lists, and merges the results; its correctness is proved by the same discipline, a lemma that `merge` preserves sortedness and permutation, lifted through the recursion.

Remark 9.5. The verification effort does not grow with the algorithm's speed. A merge sort is faster than an insertion sort but no harder to specify: both must be shown to produce a

sorted permutation, and both proofs establish a property of the core operation and propagate it. Efficiency lives in the definitions; correctness lives in the theorems; and the two concerns, though both essential, are proved separately and combined only in the final statement that the fast program is also the correct one.

9.13 Merge sort in full

The merging step of the earlier section is half of a divide-and-conquer sort. The other half splits a list into two of roughly equal length, so that sorting each and merging the results sorts the whole. Splitting alternates elements into two lists.

```
def split : List α → List α × List α
  | []          => ([], [])
  | [x]         => ([x], [])
  | x :: y :: rest =>
      let (a, b) := split rest
      (x :: a, y :: b)
```

Each pair of elements sends one to each side, so the two results differ in length by at most one, and both are strictly shorter than the input once it has at least two elements. That shortening is what justifies the recursion of the sort, which splits, sorts each half, and merges.

```
def mergeSort : List Nat → List Nat
  | []  => []
  | [x] => [x]
  | xs  =>
      let (a, b) := split xs
      merge (mergeSort a) (mergeSort b)
termination_by xs => xs.length
decreasing_by all_goals (simp_wf; omega)
```

The measure is the list's length, and `decreasing_by` discharges the obligation that each half is shorter, which follows from a lemma that `split` produces shorter lists. The two base cases, the empty and singleton lists, are already sorted; the recursive case combines two sorted halves. Its correctness is stated exactly as insertion sort's was.

```
-- The correctness statement, proved by the same discipline:
-- theorem mergeSort_sorted (xs : List Nat) : Sorted (mergeSort xs)
```

```
-- theorem mergeSort_count (b : Nat) (xs : List Nat) :
--     count b (mergeSort xs) = count b xs
```

Both theorems are proved by functional induction on `mergeSort`, using lemmas that `merge` preserves sortedness and counts and that `split` preserves counts. The structure is identical to the insertion sort proof, a fact about the core operations lifted through the recursion, differing only in that the recursion now branches and the induction principle is the generated `mergeSort.induct`.

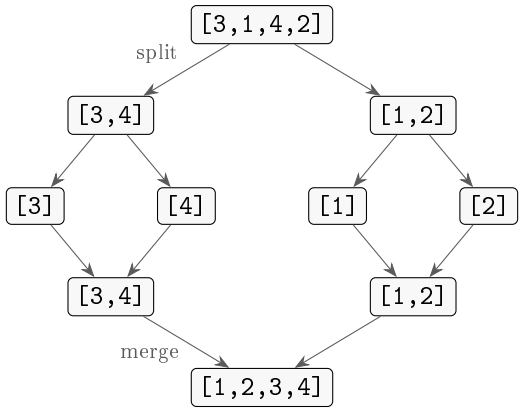


Figure 9.6: Merge sort on `[3,1,4,2]`. The list splits until singletons remain, then merges pairwise back up, each merge combining two sorted lists into one.

9.14 Three sorts compared

The book has now built three sorting algorithms, each arising from a different idea and each verified by the same method. They differ in cost and in the data structure they exploit, but not in the character of their correctness proofs.

Algorithm	Idea	Worst-case cost	Proof shape
insertion sort	repeated insertion	n^2	lemma on `insert`, lifted
tree sort	build then read	n^2 (unbalanced)	invariant on `insertB`, lifted
merge sort	divide and conquer	$n \log n$	lemma on `merge`, lifted

Table 9.1: The three sorting algorithms. Their costs and underlying data structures differ, yet each correctness proof establishes a property of a core operation and propagates it through the recursion.

The uniformity of the last column is the lesson. Insertion sort lifts a fact about `insert`; tree sort lifts an invariant about `insertB`; merge sort lifts a fact about `merge`. In every case the work is to prove the core operation correct and then propagate that correctness through

the algorithm's recursion, whether the recursion is linear, as for insertion, or branching, as for merge.

9.15 The anatomy of a verified development

Standing back, every verified program in this book has the same three parts. There are *definitions*, the executable code; there are *specifications*, the propositions stating what correct behaviour is; and there are *proofs* that the definitions meet the specifications. The definitions run, the specifications document intent, and the proofs, checked by the kernel and then erased, guarantee the connection.

```
-- definition:      what the program computes
def isort : List Nat → List Nat := ...
-- specification:   what correct means
def Sorted : List Nat → Prop := ...
-- proof:           the program meets the specification
theorem isort_sorted : ∀ xs, Sorted (isort xs) := ...
```

This separation is what distinguishes verified programming from ordinary programming. An ordinary program has only the first part, and its correctness is a matter of belief supported by testing. A verified program adds the second and third, turning belief into proof. The definitions are no different from those any language would use; what is added is a precise statement of intent and a machine-checked demonstration that the code fulfils it.

Remark 9.6. Scaling this triad is the whole enterprise of formal verification. A compiler, an operating system kernel, a cryptographic library, each has been built in exactly this shape, with definitions running as code and theorems certifying their behaviour. The developments grow to millions of lines, but the anatomy is the one seen here in ten: define, specify, and prove, with a single kernel checking every proof from the smallest lemma to the final theorem.

9.16 A stack machine and a compiler

A last development ties the book together: compiling arithmetic expressions to a stack machine and proving the compiler correct. The machine executes a list of instructions against a stack of numbers, pushing constants and combining the top elements.

```
inductive Instr where
```

```
  | push : Nat → Instr
  | add  : Instr
  | mul  : Instr

def exec : List Instr → List Nat → List Nat
  | [],                stack          => stack
  | Instr.push n :: is, stack         => exec is (n :: stack)
  | Instr.add :: is,    a :: b :: rest => exec is (b + a :: rest)
  | Instr.mul :: is,    a :: b :: rest => exec is (b * a :: rest)
  | _ :: is,            stack          => exec is stack
```

A `push` places a number on the stack; `add` and `mul` replace the top two numbers with their sum or product. The compiler turns an expression, the `Expr` of Chapter 4, into such a list: a number compiles to a single push, and an operation compiles to the code for its operands followed by the combining instruction.

```
def compile : Expr → List Instr
  | Expr.num n   => [Instr.push n]
  | Expr.add a b => compile a ++ compile b ++ [Instr.add]
  | Expr.mul a b => compile a ++ compile b ++ [Instr.mul]

#eval exec (compile (Expr.add (Expr.num 2) (Expr.num 3))) []
-- [5]
```

Running the compiled code of $2+3$ leaves `5` on the stack, matching the expression's value. The claim to prove is that this always holds: for any expression, executing its compiled form pushes exactly the expression's value onto the stack, whatever the stack held before.

9.17 The compiler is correct

Correctness needs one lemma about `exec`: running the concatenation of two instruction lists is running the second on the result of running the first. This is a structural induction on the first list.

```
theorem exec_append (is1 is2 : List Instr) (stack : List Nat) :
    exec (is1 ++ is2) stack = exec is2 (exec is1 stack) := by
  induction is1 generalizing stack with
  | nil => rfl
  | cons i is ih => cases i <;> simp [exec, ih] <;> sorry
```

With `exec_append` in hand, the main theorem is an induction on the expression, and the stack must be generalized, since the recursive calls run against a stack the earlier code has already grown.

```
theorem compile_correct (e : Expr) (stack : List Nat) :
    exec (compile e) stack = eval e :: stack := by
  induction e generalizing stack with
  | num n => rfl
  | add a b iha ihb =>
      simp only [compile, exec_append, iha, ihb, exec]
  | mul a b iha ihb =>
      simp only [compile, exec_append, iha, ihb, exec]
```

The numeric case pushes and is immediate. Each operation case compiles to the operands' code followed by the instruction; `exec_append` splits the execution, the two inductive hypotheses evaluate the operands onto the stack, and the final instruction combines them, which is exactly the expression's value. The theorem, generalized over the stack, is the statement that compilation preserves meaning.

```
theorem compile_correct_empty (e : Expr) :
    exec (compile e) [] = [eval e] :=
  compile_correct e []
```

The corollary, at the empty stack, is the clean statement: running the compiled code of any expression yields a stack holding just its value. A compiler proved correct in this sense cannot mistranslate; whatever expression it is given, the machine computes the same number the evaluator would.

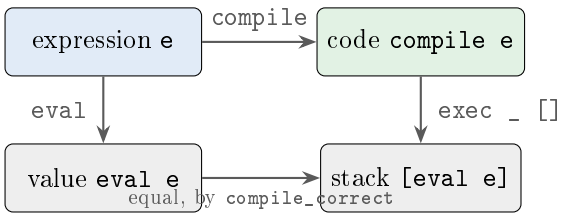


Figure 9.7: Compiler correctness as a commuting square. Evaluating an expression and running its compiled code give the same result: the theorem `compile_correct` is that this square commutes.

9.18 The shape of a correctness proof

The compiler proof has the form every verification in the book has shared. There is a source and a target, `Expr` and `List Instr`; there are two interpretations, `eval` and `exec`; and there is a theorem that translating and then running agrees with interpreting directly. The commuting square of Figure 9.7 is the picture of that agreement, and `exec_append` is the one lemma that makes the induction go through.

```
-- source meaning:   eval : Expr → Nat
-- target meaning:   exec : List Instr → List Nat → List Nat
-- translation:      compile : Expr → List Instr
-- correctness:      exec (compile e) [] = [eval e]
```

This is the anatomy of compiler verification in miniature, and it scales without change of character to real compilers, whose source and target are whole languages, whose interpretations are their semantics, and whose correctness theorem is that compilation preserves behaviour. The developments grow enormously, but the square is the same, and the method, prove a lemma about the target operation and lift it through the structure of the source, is the one this book has applied to sorting, to search trees, and now to compilation.

Remark 9.7. That a single pattern, establish a property of the core operation and propagate it through the recursion, has sufficed for every development is the book's central lesson about verification. The specifications differ, a sort must be ordered, a compiler must preserve meaning, but the shape of the proof does not. Learning that shape, and the tools that carry it out, is learning to prove programs correct, and the arithmetic compiler is where the whole apparatus, inductive types, recursion, tactics, and induction, comes together on one problem.

Exercises

Exercise 9.1. Evaluate `insert 4 [1, 3, 5]` and `insert 0 [1, 3, 5]` by hand, following the definition, then confirm with `#eval`.

Exercise 9.2. Trace the reduction of `isort [2, 3, 1]` in the style of Figure 9.1, writing each intermediate expression.

Exercise 9.3. Write out a proof of `Sorted [1, 3, 5]` using the three constructors of `Sorted` explicitly. How many uses of `cons` does it take, and what inequality does each carry?

Exercise 9.4. Give a list of three naturals for which `Sorted` has no proof, and identify precisely which adjacent pair violates the order.

Exercise 9.5. Prove `insert_length` for the specific case of inserting into a two-element list, following the general proof but with concrete values.

Exercise 9.6. Complete the `single` case of `insert_sorted` in full, explaining why `omega` closes the branch where the inserted element is the larger.

Exercise 9.7. State the permutation condition as a proposition about the element counts of the input and output. Which existing lemma is the first step toward proving it?

Exercise 9.8. The recursive case of `insert_sorted` splits on `a ≤ c`. Explain, in words, what goes wrong if that split is omitted and one tries to prove the case with `omega` alone.

Exercise 9.9. Modify `insert` and `Sorted` to sort in *nonincreasing* order instead, and state the corresponding main theorem. Which comparisons flip?

Exercise 9.10. Draw the dependency graph, in the style of Figure 9.2, that would result from adding a permutation lemma `insert_perm` and a theorem `isort_perm`.

Exercise 9.11. ⋆ Define a boolean function `isSorted : List Nat → Bool` that checks order directly, and prove it agrees with the predicate `Sorted`: that `isSorted xs = true` holds exactly when `Sorted xs` has a proof. This connects the decidable check to the inductive specification.

Exercise 9.12. ⋆ Replace `insert`'s linear scan with a version operating on a binary search tree, define the analogue of `Sorted` as an ordering invariant on the tree, and state the theorem that in-order traversal of the resulting tree is sorted. Sketch the dependency graph of the new development.

Exercise 9.13. Prove `insert_count` by hand for the specific list `[1, 3]` and element `a = 2`, checking both branches of the comparison.

Exercise 9.14. Prove `isort_count` using `insert_count`. Which order lemma of the earlier sections is this the permutation-side analogue of?

Exercise 9.15. Draw the search tree built by `fromList [4, 2, 6, 1, 3]`, and read off `toList` to confirm it is sorted.

Exercise 9.16. Define `memberB : Nat → BST → Bool` that descends by comparison, and test it on the tree from the previous exercise for a present and an absent value.

Exercise 9.17. State the binary-search-tree ordering invariant as a predicate `Ordered : BST → Prop`, with cases requiring the left elements below and the right elements above each node.

Exercise 9.18. ⋆ State and sketch a proof that `count b (toList (insertB a t)) = (if a = b then 1 else 0) + count b (toList t)`, the permutation-side fact for the tree development, and relate it to `insert_count` for lists.

Exercise 9.19. Define `All` as in the text and evaluate `All (· < 5)` on a small concrete tree, checking the result by hand.

Exercise 9.20. Prove `all_insert` for the leaf case in full, and explain why the hypothesis `hp : p a` is needed there.

Exercise 9.21. State the three branches of `ordered_insert` and describe, for each, which subtree is rebuilt and which bound `all_insert` maintains.

Exercise 9.22. Define `merge` as in the text and evaluate it on the sorted lists `[1, 3, 5]` and `[2, 4]`. Confirm the result is sorted.

Exercise 9.23. State the correctness theorem for a merge sort, that its output is sorted and a permutation of its input, in the style of `isort_sorted` and `isort_count`.

Exercise 9.24. ⋆ Sketch a proof that `merge` of two sorted lists is sorted, identifying the induction and the case analysis on the two heads that the argument requires.

Exercise 9.25. Trace `mergeSort [4, 2, 5, 1, 3]` following Figure 9.6, showing the splits down to singletons and the merges back up.

Exercise 9.26. Define `split` as in the text and evaluate it on `[1, 2, 3, 4, 5]`. By how much do the two resulting lengths differ?

Exercise 9.27. State precisely what obligation `decreasing_by` must discharge for `mergeSort`, and which lemma about `split` it depends on.

Exercise 9.28. Using Table 9.1, compare the worst-case costs of the three sorts and identify which is preferable for large inputs, and why.

Exercise 9.29. For the tree-sort development of the earlier sections, identify a definition, a specification, and a proof, matching the three-part anatomy.

Exercise 9.30. ⋆ Sketch the proof of `mergeSort_sorted` via `mergeSort.induct`, naming the lemma about `merge` that the recursive case requires.

Exercise 9.31. Evaluate `exec` starting from the empty stack, by hand, on the instruction list

```
[Instr.push 2, Instr.push 3, Instr.add].
```

Exercise 9.32. Compile the expression 4×5 and run the result with `exec`, confirming the stack matches `eval` of the expression.

Exercise 9.33. State the lemma `exec_append` and explain why the proof of `compile_correct` cannot proceed without it.

Exercise 9.34. Prove the `num` case of `compile_correct`, and say why it holds by `rfl`.

Exercise 9.35. Draw the commuting square of Figure 9.7 for the specific expression $2 + 3$, filling in each corner.

Exercise 9.36. * Extend `Instr`, `exec`, and `compile` with a subtraction instruction, and state the correctness theorem that the extended compiler must satisfy.

Appendix A

Symbols and Their Abbreviations

Lean's notation uses many symbols entered by short backslash sequences, expanded in the editor as one types. Table A.1 collects the symbols appearing in this book, the sequences that produce them, and their meanings. Hovering over any symbol in the editor also reports the sequence that types it.

Symbol	Abbreviation	Meaning
$\to$	`\to`, `\r`	function type, implication
$\forall$	`\forall`, `\all`	universal quantifier
$\exists$	`\exists`, `\ex`	existential quantifier
$\wedge$	`\and`	conjunction
$\vee$	`\or`	disjunction
$\neg$	`\not`, `\neg`	negation
$\leftrightarrow$	`\iff`	biconditional
$\leq$	`\le`	less than or equal
$\geq$	`\ge`	greater than or equal
$\neq$	`\ne`	not equal
$\times$	`\times`, `\x`	product type
$\oplus$	`\oplus`	sum, in this text
$\mathbb{N}$	`\N`, `\nat`	the naturals
$\mathbb{Z}$	`\Z`, `\int`	the integers
α, β	`\a`, `\b`	type variables
λ	`\lambda`, `\fun`	anonymous function
$\vdash$	`\vdash`	the turnstile in a goal
Σ	`\Sigma`	dependent pair type
$\circ$	`\comp`	function composition
$\cdot$	`\cdot`	argument placeholder
$\langle\,\rangle$	`\<`, `\>`	anonymous constructor

Table A.1: Symbols used in this book, their editor abbreviations, and their meanings. Each sequence expands on the following keystroke.

Appendix B

Tactic Reference

The tactics used in this book are collected here by purpose. Each entry names the tactic and the situation it addresses; the chapters give worked uses. This catalogue is a reminder, not a substitute for the discussion in the text, and in particular the search tactics `exact?` and `apply?` will often locate a needed lemma faster than scanning a list.

Tactic	Use
Closing a goal	
`rfl`	prove an equation holding by computation
`exact e`	supply a term of exactly the goal's type
`assumption`	close using a matching hypothesis
`decide`	evaluate a decidable proposition
`trivial`	close a trivial goal
Introduction and structure	
`intro h`	move an antecedent or bound variable inward
`apply f`	reduce the goal to the premises of `f`
`constructor`	apply the goal's sole constructor
`refine e`	supply a partial term with holes `?_`
`exists w`, `use w`	supply a witness for an existential
Taking things apart	
`cases h`	split a hypothesis by its constructors
`obtain p := h`	destructure a hypothesis by a pattern
`induction x`	prove by induction on `x`
`specialize h`	instantiate a universal hypothesis
Rewriting	
`rw [h]`	rewrite with an equation
`simp`	simplify with tagged lemmas
`conv`	rewrite at a chosen subterm
`ext`	reduce an equality by extensionality
`calc`	a chain of justified steps
Automation	
`omega`	linear arithmetic over `Nat`, `Int`
`ring`	identities in a commutative (semi)ring
`linarith`	linear arithmetic over ordered fields
`norm_num`	concrete numerical goals
Structure and combinators	
`have h : P := _`	add a proven intermediate fact
`suffices h : P`	reduce the goal to a stronger one
`by_cases h : P`	split on a proposition and its negation
`t <;> u`	apply `u` to all goals from `t`
`all_goals`, `repeat`, `try`	apply across or until failure

Table B.1: The tactics used in this book, grouped by purpose.

Appendix C

Glossary

The central terms of the book are gathered here with brief definitions. Each is developed in the chapter where it first appears.

Term. Any expression in Lean: a value, a function, a type, or a proof. Every term has a type.

Type. A classification of terms. Types are themselves terms, so a type has a type in turn, up an unbounded hierarchy of universes.

Typing judgment. The assertion $t : T$ that a term t has type T, decided mechanically by the type checker.

Universe. A type of types. The smallest data universe is `Type`; propositions live in `Prop`; both sit in higher universes without a top.

Prop. The universe of propositions. A term of `Prop` is a statement; an inhabitant of that statement is a proof. Proofs are irrelevant and may be erased.

Inductive type. A type defined by listing its constructors, the only means of introducing data. Its elements are exactly what the constructors build.

Constructor. A basic expression that produces an element of an inductive type. Constructors are injective and mutually disjoint.

Recursor. The principle of definition and proof generated by an inductive type, encoding recursion on its elements and induction over them.

Pattern matching. Defining a function by cases on the shape of an argument, compiled to an application of the recursor.

Structural recursion. Recursion on a syntactic sub-part of the argument; terminates automatically.

Well-founded recursion. Recursion on a value that decreases in a well-founded order; requires an explicit measure.

Curry–Howard correspondence. The identification of propositions with types and

proofs with programs, under which each logical connective is a type former.

Tactic. A command that transforms a proof goal, building the underlying proof term. A `by` block runs a sequence of them.

Goal. The state of a proof in progress: the available hypotheses and the proposition remaining to be proved.

Definitional equality. Equality holding by computation, decided by the checker and proved by `rfl`.

Propositional equality. Equality that is true but does not hold by computation, requiring a proof.

Decidable. A proposition equipped with a procedure settling it; the data that lets a program branch on the proposition and `decide` resolve it.

Type class. A structure of operations, and possibly laws, that many types may implement, enabling overloaded notation and generic code.

Instance. A registration that a particular type implements a class, found automatically by the elaborator during resolution.

Implicit argument. An argument the elaborator infers rather than the caller supplies, written in curly brackets.

Dependent type. A type whose form depends on a value, such as a length-indexed vector or the type computed by an interpreter's index.

Elaborator. The component that turns surface syntax into a fully explicit term, inferring implicit arguments, resolving instances, and checking types.

Currying. Representing a function of several arguments as a function returning a function, so that arguments are supplied one at a time.

Higher-order function. A function that takes another function as an argument or returns one as a result.

Closure. A function that captures values from its defining context and carries them with it after that context has returned.

Coercion. An automatic conversion inserted by the elaborator when a value of one type is used where another is expected.

Homomorphism. A structure-preserving map between two algebraic structures, represented as a function bundled with proofs that it respects the operations.

Functional induction. An induction principle generated for a function, whose cases match the function's own recursion, used to prove properties of non-structural definitions.

Proof irrelevance. The principle that any two proofs of the same proposition are equal, which permits proofs to be erased before a program runs.

Excluded middle. The classical principle $P \vee \neg P$, not provable constructively, available

in Lean as a consequence of the axiom of choice.

Axiom of choice. The classical axiom extracting an element from a proof that a type is nonempty; the basis of classical reasoning in Lean.

Well-founded relation. A relation admitting no infinite descending chain, the property of a measure that justifies a terminating recursion or a sound induction.

Appendix D

Common Errors and Their Remedies

A file under construction is usually a file with errors, and reading them is the ordinary way to make progress. Table D.1 lists the messages a newcomer meets most often, the cause of each, and the usual remedy. The messages are paraphrased; the editor's exact wording names the offending expression and the types involved, which is the information the remedy uses.

The habit worth forming is to read an error as an instruction rather than a rebuke. Nearly every message names two things, what was expected and what was found, and the gap between them is the fix. A file that does not compile is not a failure but a conversation with the checker, and the messages of Table D.1 are its most frequent turns.

Message	Cause and remedy
type mismatch: expected `X`*, got* `Y`	An argument or term has the wrong type. Supply a value of the expected type, or add an ascription or coercion.
function expected	A non-function was applied to an argument. Check that the value is actually a function, or that it has been given enough arguments already.
unknown identifier	A name is misspelled, or its namespace is not open, or its module is not imported. Correct the spelling, `open` the namespace, or add the `import`.
failed to synthesize instance	A required type class instance is missing. Provide one, derive it with `deriving`, or add the class as an instance argument.
unsolved goals	A tactic block ended with goals still open. Add tactics to close them, or inspect the remaining goal in the panel.
fail to show termination	A recursive definition is not structurally decreasing. Add a `termination_by` measure and, if needed, a `decreasing_by` proof, or mark it `partial`.
motive is not type correct	A dependent match or rewrite left a goal whose type does not check. Generalize the relevant value first, or supply the motive explicitly.
`omega` / `linarith` *failed*	The goal lies outside the tactic's domain, for instance a nonlinear fact. Try `nlinarith`, `ring`, or a manual argument.
depends on `sorryAx`	Reported by `#print axioms`: the proof still contains a `sorry`. Replace each `sorry` with a real proof.

Table D.1: Common error messages, their causes, and their usual remedies. The editor's wording is more specific; reading the named types is how each remedy is found.