# An introduction to Lean 4

E. Cosme Llópez[*,†]

L. Gong[†]

September 2025

[*]Universitat de València, Departament de Matemàtiques.
[†]Nantong University, School of Mathematics and Statistics.

# Contents

# An introduction to formal verification

## Introduction

Lean 4 is a versatile programming language and interactive theorem prover designed to formalize mathematics, verify software, and explore computational logic. Whether you are a mathematician, computer scientist, or a curious learner, Lean 4 offers powerful tools for rigorous reasoning and proof verification. By combining programming and formal reasoning, Lean 4 serves as an essential tool for both learning and research.

Lean 4 enables us to:

- **Prove Theorems**: Formalize and verify mathematical proofs with precision, eliminating ambiguities and errors.
- **Write Programs**: Develop functional programs with strong type safety and reliability.
- **Verify Systems**: Ensure the correctness of software and hardware through formal verification techniques.
- **Explore Logic**: Study dependent type theory, proof automation, and formal methods in depth.

This manual introduces the fundamentals of Lean 4, covering Basic syntax and types, Theorem proving and verification, and Practical applications in mathematics. Each chapter includes examples, exercises, and practical insights to help us build confidence and proficiency in Lean 4. The content of this manual is based on informal seminar sessions conducted by the author at the Universitat de València and taught to master's students at Nantong University. These sessions focus on foundational topics in mathematics, particularly the universal properties of key constructions.

This manual is available both as a web version and as a PDF. All exercises in this manual are accompanied by solutions available on GitHub. That said, the most effective way to learn is to dive in and tackle them yourself. Mistakes are a natural part of the learning process!

### A Brief History of Lean

Lean was developed by Leonardo de Moura and his team at Microsoft Research in 2013. It was created to provide a robust and scalable framework for formalizing mathematics, verifying software, and exploring type theory.

Over the years, Lean has evolved significantly, with Lean 4 offering improved performance, a redesigned type system, and enhanced support for metaprogramming. Today, it serves as a foundational tool for both theoretical and applied research in mathematics and computer science.

To learn more about Lean 4, visit the official website: lean-lang.org.

### References and Learning Resources

While this manual provides a thorough introduction to Lean 4, there are many other excellent resources available to deepen your understanding. Here are some recommended materials:

1. Functional Programming in Lean: The standard reference for learning how to use Lean as a programming language.
2. Theorem Proving in Lean 4: A comprehensive guide to using Lean as a theorem prover.
3. Mathematics in Lean: A resource focused on using Lean for formalizing mathematics.
4. The Mechanics of Proof: Lecture notes designed for early university-level students on writing rigorous mathematical proofs.

5. Lean Language Reference: A technical document describing the syntax, semantics, and standard library of Lean.
6. Documentation Overview: A collection of examples, developer guides, and other essential documentation.
7. Lean Community Learning Resources: A curated list of tutorials, guides, and documentation sources for Lean 4.
8. Lean Zulip Chat: Join the public chat room to engage with the Lean community and seek guidance.

### Installation and Quickstart Guide

To start using Lean 4, follow the Quickstart Guide from the official documentation. This guide provides step-by-step instructions on installing Lean 4 on our system, setting up a development environment and writing and running our first Lean program

For the best experience, it is recommended to use Lean 4 with **VS Code** and the Lean extension, which provides syntax highlighting, interactive proof support, and an enhanced development workflow.

## Acknowledgements

Writing this textbook has been a personal endeavor, but it would not have been possible without the support and encouragement of many individuals and institutions. First and foremost, we would like to express our deepest gratitude to the Universitat de València and Nantong University for providing an environment conducive to research and teaching, which greatly influenced the development of this book. We are deeply grateful to our students, both past and present, whose curiosity and thoughtful questions have continually inspired me to refine my explanations and enhance the clarity of the material presented here. In particular, we would like to express our heartfelt appreciation to Yan Yan for her keen insight and enthusiasm, which have been a driving force behind this project.

We would also like to acknowledge the contributions of the broader academic and open-source communities, especially the developers and maintainers of Lean and Quarto, whose work has enabled the seamless integration of formal verification and programming into this text.

For this work, the first author held a position as Specially Appointed Professor at the School of Mathematics and Statistics, Nantong University. In addition, the first author is involved in the teaching innovation project *“Beyond the Theorem: Active Strategies for Developing 21st-Century Mathematicians”* (code PIEE-3900548), under the *Vicerectorat de Formació Permanent, Transformació Docent i Ocupació de la Universitat de València.*

Finally, to all who have contributed in ways large and small, whether through direct collaboration or simply by offering words of motivation, thank you.

# 1 Basic Syntax

This chapter introduces the foundational elements of Lean 4s syntax. Well learn how to define variables, write functions, and work with types and expressions, the essential building blocks of Lean 4 programming. By the end, we'll be able to read and write basic Lean 4 programs, preparing us for more advanced topics ahead.

Lets start with a fundamental question.

## 1.1 What is a type?

A type classifies data, defining what values it can hold and what operations can be performed on it. Types are essential in both programming and theorem proving, ensuring correctness and structuring reasoning. In Lean 4, types act as a safety mechanism:

- A value of type `Nat` (natural number) can be `0, 1, 2`, etc.
- A value of type `String` can be `"hello", "Lean 4"`, etc.
- A function of type `Nat → Nat` takes a natural number as input and returns another natural number.

By assigning types to values and functions, Lean 4 prevents errors like adding a number to a string or applying a function to incompatible data.

Next, we'll explore some fundamental Lean 4 commands.

## 1.2 Comment code

In Lean 4, comments help make code more readable and serve as documentation. They are ignored by the compiler and do not affect execution. Here's how to write comments in Lean:

- Single-line comments start with `--` and apply to the rest of the line.
- Multi-line comments are enclosed between `/-` and `-/`.

```
-- This is a comment

/-
This is a multi-line comment.
It can span multiple lines.
Useful for longer explanations.
-/
```

## 1.3 check

We'll begin with the `#check` command, a key tool for exploring Lean's type system. The `#check` command allows us to inspect the type of an expression, definition, or theorem in Lean. It's invaluable for understanding how Lean interprets our code and for troubleshooting type-related issues.

```
#check true
#check 42
#check 'h'
#check ['h', 'e', 'l', 'l', 'o']
#check "hello"
#check Nat
```

In these examples:

1. outputs `Bool.true : Bool` which tells us that `true` is of type `Bool`.

2. outputs `42 : Nat` which tells us that `42` is of type `Nat` (natural number).

3. outputs `"h" : Char` which tells us that `"h"` is of type `Char` (a character).

4. outputs `['h', 'e', 'l', 'l', 'o'] : List Char` which tells us that `['h', 'e', 'l', 'l', 'o']` is a list of characters.

5. outputs `"hello" : String` which tells us that `"hello"` is of type `String`.

6. outputs `Nat : Type` which tells us that `Nat` is of type `Type`.

## 1.4 print

The `#print` command allows us to inspect the definition of a function, theorem, or other named entity in Lean. It provides detailed information, including the type, implementation, and any dependencies. This command is especially useful for understanding how Lean's standard library works or for debugging our own code.

```
#print Bool
#print Nat
#print Char
#print List
#print String
-- We cannot `#print Type` because this is a built-in concept
#print Type
```

In these examples:

- `#print Bool` outputs

```
inductive Bool : Type
number of parameters: 0
constructors:
Bool.false : Bool
Bool.true : Bool
```

This tells us that `Bool` is an inductive type with no parameters and two constructors. Constructors are ways to provide elements of the given type—in this case, `Bool.false` represents the value false, and `Bool.true` represents the value true. Thus, an **inductive type** defines a new type by specifying a set of constructors that generate its elements. Each constructor may take arguments, including recursive references to the type itself. Inductive types are fundamental in Lean, serving as the basis for both **data structures** (such as natural numbers, lists, and trees) and **logical propositions**. We will explore inductive types in more detail later, but for now, understand that constructors define the possible values of a type, and `Bool` is a type with exactly two such values.

- `#print Nat` outputs

```
inductive Nat : Type
number of parameters: 0
constructors:
Nat.zero : Nat
Nat.succ : Nat → Nat
```

As we can see, `Nat` is also an inductive type representing natural numbers, which do not require parameters. It has two constructors:

- `Nat.zero`, which represents the number 0.

- `Nat.succ`, which represents the successor of a natural number (essentially adding 1).

This definition allows natural numbers to be constructed starting from 0 and adding 1 repeatedly. Another way of denoting natural numbers in Lean is using `ℕ`. To type this symbol, you can use the shortcut `\N`.

- `#print Char` outputs

```
structure Char : Type
number of parameters: 0
constructor:
Char.mk : (val : UInt32) → val.isValidChar → Char
fields:
val : UInt32
valid : self.val.isValidChar
```

`Char` is a structure representing a single character and does not have parameters. A *structure* in Lean is a way to define a type that groups together related data. The `Char` structure has a constructor called `Char.mk`, which takes:

- A `UInt32` value (a 32-bit unsigned integer) representing the Unicode code point of the character.
- `val.isValidChar`, a proof that the value is a valid Unicode character.

Moreover, every element of type `Char` has two associated fields:

- `val`: This field returns the Unicode value of the character.
- `valid`: This field returns the proof that the code point is valid.

Thus, the `Char` structure combines the character value and its validity in a structured way. We will also explore structures in more detail in future chapters.

- `#print List` outputs

```
inductive List.{u} : Type u → Type u
number of parameters: 1
constructors:
List.nil : { α : Type u } → List α
List.cons : { α : Type u } → α → List α → List α
```

`List` is an inductive type that represents a sequence of elements of a given type `α`. This type requires one parameter, `α`, where `α` is the type of elements in the list. The universe level `u` is used in the type definition to avoid paradoxes in type theory. For example, when we instantiate `List` with `Nat`, we get the type of lists of natural numbers (`List Nat`), and when instantiated with `Char`, we get the type of lists of characters (`List Char`). The List type has two constructors:

- `List.nil`, which represents the empty list ([]).
- `List.cons`, which adds an element to the front of a list.

Thus, `List` allows us to work with ordered sequences of elements, either empty or with elements added recursively.

- `#print String` outputs

```
structure String : Type
number of parameters: 0
constructor:
String.mk : List Char → String
fields:
data : List Char
```

This indicates that `String` is a structure in Lean, with no parameters, that represents strings. The constructor for `String` is `String.mk`, which takes a `List Char` (a list of characters) as input and returns a `String`. Each element of type `String` has a field called `data`, which returns the list of characters that make up the string.

- `#print Type` outputs an error.

The reason is that `Type` is a fundamental, built-in concept within Lean's logical framework, not a user-defined term or definition. In Lean, `Type` denotes the universe of all types, and it cannot be queried using `#print` because it is not a printable entity like user-defined terms.

## 1.5 def

The `def` keyword is used to define a function or a value in Lean. It is one of the most fundamental constructs, allowing us to create reusable code—whether for simple values, functions, or more complex computations.

```
-- Definition of number pi
def pi : Float := 3.1415926
```

Definitions can have parameters.

```
-- Definition of the sum of two natural numbers
def sum (a b : Nat) : Nat := a + b
```

## 1.6 fun

The `fun` keyword is used to define anonymous functions (also called lambda functions) in Lean. These are functions that don't have a name and are defined inline. The `fun` keyword is a core concept in functional programming. We can also use the λ (lambda) operator to define anonymous functions, which is written as `\lambda` in Lean.

```
-- Two anonymous ways of defining the sum of two natural numbers
#check fun (a b : Nat) => a + b
#check λ (a b : Nat) => a + b
```

The resulting type `Nat → Nat → Nat` represents a curried function, which is a fundamental concept in functional programming. This type can be understood as follows:

- `Nat → Nat → Nat` is a function that takes a `Nat` (first argument) and returns another function.
- The returned function then takes a `Nat` (second argument) and returns a `Nat` (final result).

This curried function type is equivalent to `Nat → (Nat → Nat)`—meaning a function that takes a `Nat` and returns a function from `Nat` to `Nat`. In Lean, function types are right-associative, so the parentheses are often omitted for clarity. Thus, `Nat → Nat → Nat` is interpreted as `Nat → (Nat → Nat)`.

## 1.7 The function type

In general, if `A` and `B` are types, then `A → B` is a type representing all mappings from type `A` to type `B`. This means that any value of type `A → B` is a mapping that takes an element of type `A` and returns an element of type `B`. In Lean, we use `\to` to type the arrow → when writing this in code. We will explore this type in more detail later.

```
-- The type of all mappings from Nat to Nat
#check Nat → Nat
-- An example of an element of the above type
#check sum 3
```

This new construction provides an alternative approach to defining the previous sum mapping. The goal is to construct an element of type `Nat → Nat → Nat`. There are several ways to achieve this: either by using the `fun` keyword, as previously demonstrated, or by explicitly introducing the variables and specifying the expression that the mapping should return.

The latter approach uses the keywords `by`, `intro`, and `exact`. These are used in interactive proof mode, where you construct definitions step by step using **tactics**.

- `by`: This keyword signals that the function definition will be built interactively using tactics.
- `intro`: This tactic introduces the function's arguments as hypotheses in the goal.
- `exact`: This tactic is used to provide the exact value that satisfies the goal.

```
-- Two new ways of defining the sum of two natural numbers
def sum2 : Nat → Nat → Nat := fun a b => a + b
--
def sum3 : Nat → Nat → Nat := by
    intro a b
    exact a + b
```

## 1.8 cases

The `cases` tactic splits the definition into two branches based on the value of the input. We will use it to define the Negation function for Booleans.

```
def BoolNot : Bool → Bool := by
  intro b
  cases b
  -- b = false
  exact true
  -- b = true
  exact false
```

## 1.9 match

The `match` keyword is used for pattern matching. This is another way to handle different cases of the input.

```
def BoolNot2 : Bool → Bool := by
  intro b
  match b with
    | false => exact true
    | true  => exact false
```

## 1.10 let

The `let` keyword is used to define local variables or terms within a proof or expression.

```
def sphereVolume (r : Float) : Float :=
  let pi : Float := 3.1415926
  (4/3) * pi * r^3
```

## 1.11 eval

The `#eval` command is used to evaluate an expression and display its result. It is one of the most commonly used commands for testing and debugging code, as it allows us to see the output of a computation directly.

```
#eval sum 3 4
#eval (sum 3) 4
#eval (fun (a b : Nat) => a + b) 3 4
--
#eval pi
#eval Nat.succ 4
#eval UInt32.isValidChar 104
#eval 'h'.val
#eval String.mk ['h', 'e', 'l', 'l', 'o']
#eval "hello".data
```

The first evaluations result in `7` because they compute the sum `3 + 4`. After that, the following evaluations make use of previously defined values. For example, `#eval pi` returns the value of pi that was defined earlier. `#eval Nat.succ 4` evaluates the successor function for natural numbers, applied to 4, returning 5 as the result. `#eval UInt32.isValidChar 104` returns `true`, indicating that `104` is a valid Unicode character. `#eval 'h'.val` shows that the character `'h'` has the Unicode value `104`.

Additionally, `#eval String.mk ['h', 'e', 'l', 'l', 'o']` transforms the list of characters `['h', 'e', 'l', 'l', 'o']` into a string, in this case `"hello"`. Finally, `#eval "hello".data` returns the list of characters that make up the string `"hello"`, in this case `['h', 'e', 'l', 'l', 'o']`.

## 1.12 variable

The keyword `variable` is used to declare variables that can be used later in the code. These variables are implicitly available in the context of any theorem, definition, or proof that follows. They allow us to introduce general assumptions or placeholders for types or values without needing to explicitly define them at each step.

```
variable (m n : Nat)
#check m
```

## 1.13 namespaces

Namespaces in Lean are used to organize code, helping with structure and readability. We can define variables, functions, or theorems within a `namespace`, and these definitions are scoped to that `namespace`, meaning they are only accessible inside it. When we exit a `namespace`, the variables defined inside it are no longer recognized. To access a variable defined within a `namespace`, we must reference it using the `namespace` name.

```
namespace WorkSpace
-- Define a natural number `r` with the value 27
def r : Nat := 27
-- The variable `r` is perfectly defined within the namespace
#eval r
end WorkSpace

-- Evaluating `r` outside the namespace will result in an error
#eval r  -- Error: unknown identifier 'r'

-- To access `r`, we must reference it using its namespace
#eval WorkSpace.r  -- Output: 27
```

## 1.14 open

The `open` keyword is used in Lean to bring definitions, theorems, or namespaces into the current scope, allowing us to reference them without needing to use their full qualified names. This helps make our code more concise and easier to read by reducing the need for repetitive namespace prefixes.

```
open Workspace
#eval r
```

# 2 Propositions

This chapter introduces the type of propositions in Lean, along with fundamental logical connectives (∧, ∨, →, ¬, ↔). We will learn how to construct basic proofs using these concepts. By the end of this chapter, we will be able to write and prove simple logical statements in Lean.

### What is a Proposition?

Propositions are statements that express a definite claim. In Lean, propositions belong to the built-in type `Prop`, which is fundamental to Lean's logical system.

Examples:

- `2 + 2 = 4` is a proposition (true).
- `3 < 1` is also a proposition (false).

```
-- Prop is the type of all propositions in Lean
#check Prop
#print Prop
-- Examples
#check 2 + 2 = 4
#check 3 < 1
```

In Lean, a proposition is a type that represents a logical statement. This means that propositions themselves are types, and proving a proposition is equivalent to constructing a term of that type. If we declare a variable `P` of type `Prop`, we can later define another variable `h` of type `P`. In Lean's type-theoretic framework, we interpret `h` as a proof of `P`.

```
variable (P : Prop)
variable (h : P)
-- We need to understand h as a proof of P
#check h
```

Declaring a variable `P : Prop` does not mean that `P` is immediately true. It simply introduces `P` as a proposition. A proposition `P` is true if and only if there exists a term of type `P`—that is, a proof of P. If we can construct a term `h : P`, then `P` is true. Conversely, if no such term exists, then `P` is false. For example

- `2 + 2 = 4` is true because Lean can construct a proof of this proposition.
- `3 < 1` is false because no proof (no term of this type) can be constructed.

This perspective is central to constructive logic, where truth means having an explicit proof.

## 2.1 First proofs

The following Lean code defines a theorem called `Th1`. Let's examine its components step by step to understand its purpose and functionality.

```
theorem Th1 (h : P) : P := by
  exact h
```

This code defines a theorem named `Th1`. In Lean, a theorem is a proposition that has been or will be proven.

- `(h : P)` introduces a hypothesis `h` of type `P`, meaning `h` serves as a proof of the proposition `P`.
- `: P` represents the conclusion, stating that the theorem will establish the truth of `P`.

- `:= by` signals that the proof will be constructed interactively using tactics, entering proof mode through indentation.
- The `exact h` tactic completes the proof by instructing Lean to use `h` (a proof of `P`) to establish `P`.

Essentially, `Th1` asserts a fundamental logical principle: if a proof of `P` exists (`h : P`), then `P` is true. While seemingly obvious, this concept underlies the foundations of formal reasoning in Lean.

The theorem `Th1` is now a reusable component that can be referenced and applied wherever needed, allowing us to build on previous results.

When checking its type, Lean shows `∀ (P : Prop), P → P`. This indicates that `P` is an arbitrary proposition, and `Th1` is a function that takes a proof of `P` and returns a proof of `P`. In other words, `Th1 P` belongs to the function type `P → P`.

Since we previously defined `h : P`, applying `Th1` to `P` and `h` —written as `Th1 P h`— yields an element of type `P`. We will explore this concept further in the next sections.

```
-- Th1 has type ∀ (P : Prop), P → P
#check Th1
-- Th1 P has type P → P
#check Th1 P
-- Th1 P h has type P
#check Th1 P h
```

Notice that `Th1` is adaptable to any proposition it is applied to. For example, if we introduce a new variable `Q : Prop`, then `Th1 Q` becomes an element of type `Q → Q`. This means `Th1` can be used with any proposition, reinforcing its generality in formal reasoning.

```
variable (Q : Prop)
-- Th1 Q has type Q → Q
#check Th1 Q
```

It is worth noting that when we use `#print Th1`, Lean returns the following code:

```
theorem Th1 : ∀ (P : Prop), P → P := fun P h => h
```

The difference occurs because Lean automatically generalizes `Th1` to its most abstract form.

However, this notation can sometimes be cumbersome since applying a theorem requires explicitly providing all necessary hypotheses. To simplify this, we can define implicit variables. For example, let's introduce our second theorem:

```
theorem Th2 {P : Prop} (h : P) : P := by
  exact h
```

The curly braces around `P` indicate that `P` is an **implicit variable**. This means that Lean will automatically infer the value of `P` based on the context when `Th2` is used, so we don't need to manually provide `P` as an argument each time. This makes the code cleaner and more concise, as Lean handles the inference for us.

```
-- Th2 has type ∀ {P : Prop}, P → P
#check Th2
-- Th2 h has type P, infered from h
#check Th2 h
```

We can also prove this theorem using a third method, where we employ the `apply` tactic. The `apply` tactic allows us to use a previously defined theorem—in this case, `Th2`—to progress towards the current goal.

For `apply` to work, the conclusion of the theorem we want to apply must match or be unifiable with the current goal, which is true in this case. Once we apply `Th2`, Lean will prompt us to prove the necessary hypotheses required by the theorem to complete the proof. This approach leverages the power of previously established results to build new proofs more efficiently.

```
theorem Th3 {P : Prop} (h : P) : P := by
  apply Th2
  exact h
```

### 2.1.1 have

The have keyword in Lean is a powerful tool used in proofs to introduce intermediate results or hypotheses. It helps break down complex proofs into smaller, more manageable steps by allowing us to prove and name intermediate statements. These intermediate results can be referenced later in the proof, making the overall structure clearer and easier to follow. This approach is particularly useful when working through multi-step arguments, as it enables us to focus on individual pieces of the proof before combining them to reach the final conclusion.

```
theorem Th4 {P : Prop} (h : P) : P := by
  have h2 : P := by
    exact h
  exact h2
```

### 2.1.2 apply? exact?

One of the most powerful features of Lean is the ability to use the apply? command within a proof. The apply? tactic automatically searches through the available theorems in the current context and suggests relevant ones that could be applied to prove the current goal. For example, in the theorem Th3 above, if we write apply?, Lean responds with

```
Try this: exact h
```

Additionally, we can use the exact? tactic to prompt Lean to suggest the hypothesis needed to conclude a theorem. When invoked, exact? analyzes the current goal and offers possible hypotheses or terms that could directly satisfy the goal.

These tactics are especially useful when working with numerous results, as they save time by automatically searching for and suggesting relevant theorems or hypotheses. This eliminates the need to manually search for the specific result needed, streamlining the proof process significantly.

### 2.1.3 example

We can use the example keyword to define an anonymous theorem. This allows us to demonstrate a proof without giving it a specific name. The structure is similar to a regular theorem, but the difference is that it's not assigned a name, making it ideal for quick demonstrations or illustrating concepts. Here's an example that follows the structure from above:

```
example (h : P) : P := by
  exact h
```

Since examples are anonymous, they cannot be referenced or reused later in the code.

### 2.1.4 sorry

The sorry command in Lean is a placeholder that allows us to temporarily skip the proof of a theorem or definition. When we use sorry, Lean assumes that the proof is correct without actually verifying it. This can be useful during the development process when we want to focus on the structure of our code or test parts of our work without completing all the proofs. However, it's important to note that sorry does not provide a valid proof. If left in the code, Lean cannot guarantee the correctness of the theorem or definition, as the proof is incomplete. It's a useful tool for incremental development, but should be removed or replaced with a valid proof before finalizing the code.

```
theorem Th3 (h : P) : P := by
  sorry
```

## 2.2 Logical connectives

In this section, we will introduce logical connectives, explore their implementation in Lean, and demonstrate how to prove statements involving them.

### 2.2.1 Conjunction

The logical And connective, represented by the symbol ∧ (which can be typed in Lean using \and), is used to combine two propositions, asserting that both are true simultaneously. In Lean, the And connective is implemented as a built-in logical construct. Specifically, if P and Q are propositions, then P ∧ Q is also a proposition. This is read as "P and Q," meaning that P is true and Q is true.

```
#check And P Q
#check P ∧ Q
```

If we #print And, Lean returns

```
structure And : Prop → Prop → Prop
number of parameters: 2
constructor:
And.intro : ∀ {a b : Prop}, a → b → a ∧ b
fields:
left : a
right : b
```

This declares And as a structure in Lean. And takes two arguments of type Prop and returns a new Prop. In other words, And is a binary logical connective, requiring two parameters, both of type Prop. The constructor for And is And.intro, which is a function that takes two propositions a and b (implicitly defined) and two proofs —one proof of a (of type a) and one proof of b (of type b)— and returns a proof of a ∧ b. Here's an example of how the constructor works:

```
-- To prove a proposition of the form P ∧ Q we need a proof of P and a proof of Q
theorem ThAndIn (hP : P) (hQ : Q) : P ∧ Q := by
  exact And.intro hP hQ
```

The And structure has two fields: left and right. The left field stores the proof of the first proposition a, and the right field stores the proof of the second proposition b. These fields are essential for constructing a proof of a ∧ b, as they hold the individual proofs required to establish the truth of both propositions simultaneously. Here's an example of how the fields work:

```
-- From a proof of P ∧ Q, we can obtain a proof of P
theorem ThAndOutl (h : P ∧ Q) : P := by
  exact h.left
-- We can also obtain a proof for Q
theorem ThAndOutr (h : P ∧ Q) : Q := by
  exact h.right
```

### 2.2.2 Disjunction

The logical Or connective, represented by the symbol  (which can be typed in Lean using \or), is used to combine two propositions, asserting that at least one of them is true. In Lean, the Or connective is implemented as a built-in logical construct. Specifically, if P and Q are propositions, then P  Q is also a proposition. This is read as "P or Q," meaning that P is true or Q is true.

```
#check Or P Q
#check P ∨ Q
```

If we #print Or, Lean returns

```
inductive Or : Prop → Prop → Prop
number of parameters: 2
constructors:
Or.inl : ∀ {a b : Prop}, a → a ∨ b
Or.inr : ∀ {a b : Prop}, b → a ∨ b
```

This declares Or as an inductive type in Lean. Or takes two arguments of type Prop and returns a new Prop. In other words, Or is a binary logical connective that requires two parameters, both of which are of type Prop. There are two constructors for Or: Or.inl and Or.inr. These constructors are functions that take two propositions, a and b, implicitly defined, and a proof—either a proof of a (of type a) or a proof of b (of type b)—and return a proof of a ∨ b. The two constructors Or.inl and Or.inr correspond to the two possible ways a disjunction can be true: either by proving a or by proving b. Here's an example of how the constructors work:

```
-- From a proof of P, we can obtain a proof of P ∨ Q
theorem ThOrInl (h : P) : P ∨ Q := by
  exact Or.inl h
-- From a proof of Q, we can obtain a proof of P ∨ Q
theorem ThOrInr (h : Q) : P ∨ Q := by
  exact Or.inr h
```

Unlike the `And` type, `Or` does not have fields associated with it. However, the absence of fields does not mean we cannot reason with elements of this type. In cases where we have a hypothesis of type `P ∨ Q`, we can reason by cases. In Lean, the `cases` keyword is used for pattern matching and case analysis on inductive types. It allows us to break down a hypothesis or term into its possible constructors and handle each case separately.

For example, if we have a hypothesis of type `P ∨ Q`, we know that there are two possible cases to consider: either we have a proof of `P` (using `Or.inl`), or we have a proof of `Q` (using `Or.inr`). The `cases` tactic will break down the goal into two branches, one for each case, allowing us to reason about each case individually. Let's look at how we can use `cases` to handle such a scenario:

```
-- From a proof of P ∨ Q, we can obtain a proof of Q ∨ P
theorem ThOrCases (h : P ∨ Q) : Q ∨ P := by
  cases h
  -- Case 1
  rename_i hP
  exact Or.inr hP
  -- Case 2
  rename_i hQ
  exact Or.inl hQ
```

This code defines a theorem in Lean that demonstrates how to reason by cases using the `cases` tactic. The theorem proves that if you have a proof of `P ∨ Q`, you can derive a proof of `Q ∨ P`. Here's how the proof works:

1. Hypothesis: We start with the hypothesis `h : P ∨ Q`, which asserts that at least one of the propositions `P` or `Q` is true.

2. Case Analysis with `cases`: The `cases` tactic is applied to `h` to break it into two subgoals, each corresponding to a possible way the disjunction could have been constructed:

   - Case 1: If `h` is constructed using `Or.inl`, it means we have a proof of `P` (denoted `hP` using `rename_i`), so `hP : P`. To prove `Q ∨ P`, we use `Or.inr` to inject `P` into the right side of the disjunction, giving us the proof `Q ∨ P`.

   - Case 2: If `h` is constructed using `Or.inr`, it means we have a proof of `Q` (denoted `hQ` using `rename_i`), so `hQ : Q`. To prove `Q ∨ P`, we use `Or.inl` to inject `Q` into the left side of the disjunction, completing the proof.

In both cases, we construct a valid proof of `Q ∨ P` by appropriately using the constructors `Or.inl` and `Or.inr`. The `cases` tactic allows us to handle each scenario separately and derive the desired result.

We can alternatively use the keyword `Or.elim` to provide an alternative proof by cases.

```
theorem ThOrCases2 (h : P ∨ Q) : Q ∨ P := by
  apply Or.elim h
  -- Case P
  intro hP
  exact Or.inr hP
  -- Case Q
  intro hQ
  exact Or.inl hQ
```

### 2.2.3 Implication

The logical implication connective, represented by the symbol → (which can be typed in Lean using `\to`), is used to combine two propositions and describe a conditional relationship between them. Specifically, if `P` and `Q` are propositions, then `P → Q` is also a proposition, which reads as "if `P`, then `Q`", or "`P` implies `Q`." This means that if `P` is true, then `Q` must also be true. If `P` is false, the implication `P → Q` is considered true regardless of the truth value of `Q`. This is known as a *vacuous truth*.

In Lean, implication is treated as a function type: a proof of `P → Q` is a function that takes a proof of `P` and produces a proof of `Q`. To prove an implication `P → Q`, you assume that `P` is true and then show that `Q` must also be true under this assumption. This is typically done using the `intro` tactic, which introduces the assumption `P` into the proof context. Let's look at an example to illustrate this.

```
-- From a proof of Q, we can obtain a proof of P → Q
theorem ThImpIn (hQ : Q) : P → Q := by
  intro hP
  exact hQ
```

Additionally, if we have a proof of `P → Q` and a proof of `P`, we can derive a proof of `Q`. This is an application of *modus ponens*, a fundamental rule of inference in logic. The process involves applying the proof of `P → Q` to the proof of `P`, which allows us to conclude `Q`.

Conceptually, this process is similar to how a function operates: just as a function takes an input and transforms it into an output, the implication `P → Q` takes the proof of `P` (the input) and transforms it into a proof of `Q` (the output).

```
-- From a proof P → Q and a proof of P, we can obtain a proof of Q
theorem ThModusPonens (h : P → Q) (hP : P) : Q := by
  exact h hP
```

### 2.2.4 Double implication

The double implication connective, `Iff` (represented by the symbol `↔`, which can be typed in Lean using `\iff`), is used to combine two propositions, expressing a biconditional relationship between them. Specifically, if `P` and `Q` are propositions, then `P ↔ Q` is also a proposition. This is read as "`P` if, and only if, `Q`," meaning that `P` is true if `Q` is true, and `Q` is true if `P` is true.

In other words, `P ↔ Q` asserts that `P` and `Q` are logically equivalent: if one is true, the other must also be true, and if one is false, the other must also be false. This biconditional relationship combines two implications: `P → Q` and `Q → P`. Both directions must hold for `P ↔ Q` to be true, meaning that `P` and `Q` are interchangeable in terms of truth values.

```
#check Iff P Q
#check P ↔ Q
```

If we `#print Iff`, Lean returns

```
structure Iff : Prop → Prop → Prop
number of parameters: 2
constructor:
Iff.intro : ∀ {a b : Prop}, (a → b) → (b → a) → (a ↔ b)
fields:
mp : a → b
mpr : b → a
```

Thus, `Iff` is a structure that takes two propositions (`a` and `b`) as inputs and returns a new proposition (`a ↔ b`). The constructor for `Iff` is named `Iff.intro`. It requires two proofs: one of `a → b` and one of `b → a`. Using these two proofs, it constructs a proof of `a ↔ b`.

Additionally, the `Iff` structure has two fields: - `mp` (short for modus ponens), which is a proof of `a → b` - `mpr` (short for modus ponens reverse), which is a proof of `b → a`.

These fields store the two implications that together prove the equivalence `a ↔ b`. Essentially, the double implication `a ↔ b` is shorthand for the conjunction `(a → b) ∧ (b → a)`, as we can see below:

```
-- From P ↔ Q we can derive (P → Q) ∧ (Q → P)
theorem ThIffOut (h : P ↔ Q) : (P → Q) ∧ (Q → P) := by
  apply And.intro
  -- Left
  exact h.mp
  -- Right
  exact h.mpr
```

In the previous proof, from `P ↔ Q`, we can derive `(P → Q) ∧ (Q → P)`. To do this, we use the `And.intro` constructor To obtain the left hand side of the desired proposition we use the `mp` field and to obtain the right hand side we use the `mpr` field.

```
-- From (P → Q) ∧ (Q → P) we can derive P ↔ Q
theorem ThIffIn (h1 : P → Q) (h2 : Q → P) : P ↔ Q := by
  exact Iff.intro h1 h2
```

In the previous proof, from `P → Q` and `Q → P` we can derive `P ↔ Q`. To do this, we use the `Iff.intro` constructor.

### 2.2.5 True

The logical constant `True` is a proposition that is always true.

```
#check True
```

If we `#print True`, Lean returns

```
inductive True : Prop
number of parameters: 0
constructors:
True.intro : True
```

The `True` type in Lean represents the logical proposition true. It is a proposition and has no parameters. The only constructor for `True` is `True.intro`. This constructor is the canonical proof of the proposition `True`. When we use `True.intro`, we are essentially providing a proof that `True` is true, which completes any proof that requires a `True` proposition.

```
-- True can always be obtained
theorem ThTrueIn : True := by
  exact True.intro
```

An alternative way to obtain a proof of `True` is to write `trivial`, which is an element of type `True`.

```
-- Trivial is an element of type True
theorem ThTrivial : True := by
  exact trivial
```

### 2.2.6 False

The logical constant `False` is a proposition that is always false.

```
#check False
```

If we `#print False`, Lean returns

```
inductive False : Prop
number of parameters: 0
constructors:
```

The `False` type in Lean represents the logical proposition false. It is an inductive type, but unlike `True`, it has no constructors. This means that no terms or proofs of type `False` can exist. The absence of constructors implies that the type is uninhabited—there is no way to construct a proof of `False`. Given that `False` has no constructors, the principle of *ex falso quodlibet* (from falsehood, anything follows) holds: if we can derive a proof of `False`, we can derive any other proposition. This is the logical principle that allows us to infer arbitrary conclusions from a contradiction. To apply this principle, Lean provides the tactic `False.elim`. This tactic allows us to derive any proposition from a proof of `False`.

```
-- False implies any proposition
theorem ThExFalso : False → P := by
  intro h
  exact False.elim h
```

### 2.2.7 Negation

In Lean, the negation connective `Not`, represented by the symbol `¬`, is used to express the negation of a proposition. Specifically, if `P` is a proposition, then `¬P` is another proposition that reads as *not P*. In logical terms, `¬P` means that `P` is false.

```
#check Not P
#check ¬P
```

If we `#print Not`, Lean returns

```
def Not : Prop → Prop :=
fun a => a → False
```

That is, `¬P` is an abbreviation for the implication `P → False`. Therefore, to prove `¬P`, we need to show that assuming `P` leads to a contradiction. Let's see an example.

```
theorem ThModusTollens (h1 : P → Q) (h2 : ¬Q) : ¬P := by
  -- Assume P is true (to prove ¬P, which is P → False).
  intro h3
  -- Derive Q from P → Q and P.
  have h4 : Q := by
    exact h1 h3
  -- Use ¬Q (Q → False) and Q to derive False.
  exact h2 h4
```

In the above theorem, we are given the hypotheses `h1 : P → Q` and `h2 : ¬Q`, and our goal is to prove `¬P`. To do this, we begin by assuming `P` and aim to derive `False`. From the assumption `P`, we can use `h1 : P → Q` to derive `Q`. Then, since we also have `h2 : ¬Q`, which asserts that `Q` is false, we reach a contradiction. This contradiction allows us to conclude `False`, which completes the proof of `¬P`. The theorem demonstrates the well-known logical principle of *Modus Tollens*, a fundamental rule in classical logic.

## 2.3 Decidable propositions

A proposition is **decidable** if we can constructively determine whether it is true or false. That is, we have either a proof of the proposition or a proof of its negation. In Lean, **decidability** of a proposition is captured by the inductive type `Decidable`.

If we `#print Decidable`, Lean returns

```
inductive Decidable : Prop → Type
number of parameters: 1
constructors:
Decidable.isFalse : {p : Prop} → ¬p → Decidable p
Decidable.isTrue : {p : Prop} → p → Decidable p
```

`Decidable` takes a proposition `p : Prop` as a parameter. This type expresses the idea that we can *constructively decide* whether `p` holds or not—that is, we can either prove `p` or prove its negation `¬p`. The `Decidable` type has two constructors: `Decidable.isTrue` and `Decidable.isFalse`. The constructor `isTrue` takes a proof of `p` and yields a value of type `Decidable p`, indicating that `p` is provably true. Conversely, `isFalse` takes a proof of `¬p` and returns a value of type `Decidable p`, indicating that `p` is provably false.

In the code below, we prove that `True` and `False` are decidable, and that each logical connective is decidable—provided that the propositions they operate on are themselves decidable.

```
-- True is decidable
def DecidableTrue : Decidable True := by
  exact isTrue trivial

-- False is decidable
def DecidableFalse : Decidable False := by
  exact isFalse id

-- If `P` is decidable, then `¬ P` is decidable
def DecidableNot {P : Prop} : Decidable P → Decidable (¬ P) := by
  intro hP
  match hP with
    | isFalse hP => exact isTrue  (fun h => False.elim (hP h))
    | isTrue hP  => exact isFalse (fun h => False.elim (h hP))

-- If `P` and `Q` are decidable, then `P ∧ Q` is decidable
def DecidableAnd {P Q : Prop} : Decidable P → Decidable Q → Decidable (P ∧ Q) := by
  intro hP hQ
```

```
  match hP, hQ with
    | isFalse hP, _           => exact isFalse (fun h => hP h.left)
    | _         , isFalse hQ  => exact isFalse (fun h => hQ h.right)
    | isTrue hP , isTrue hQ   => exact isTrue (And.intro hP hQ)

-- If `P` and `Q` are decidable, then `P ∨ Q` is decidable
def DecidableOr {P Q : Prop} : Decidable P → Decidable Q → Decidable (P ∨ Q) := by
  intro hP hQ
  match hP, hQ with
    | isTrue hP , _            => exact isTrue (Or.inl hP)
    | _         , isTrue hQ    => exact isTrue (Or.inr hQ)
    | isFalse hP, isFalse hQ   => exact isFalse (fun h => h.elim hP hQ)

-- If `P` and `Q` are decidable, then `P → Q` is decidable
def DecidableImplies {P Q : Prop} : Decidable P → Decidable Q → Decidable (P → Q) := by
  intro hP hQ
  match hP, hQ with
    | isFalse hP , _          => exact isTrue  (fun h => False.elim (hP h))
    | _          , isTrue hQ  => exact isTrue  (fun _ => hQ)
    | isTrue hP  , isFalse hQ => exact isFalse (fun h => hQ (h hP))

-- If `P` and `Q` are decidable, then `P ↔ Q` is decidable
def DecidableIff {P Q : Prop} : Decidable P → Decidable Q → Decidable (P ↔ Q) := by
  intro hP hQ
  have hPtoQ : Decidable (P → Q) := DecidableImplies hP hQ
  have hQtoP : Decidable (Q → P) := DecidableImplies hQ hP
  match hPtoQ, hQtoP with
    | isFalse hPtoQ, _ => exact isFalse (fun h => hPtoQ h.mp)
    | _, isFalse hQtoP => exact isFalse (fun h => hQtoP h.mpr)
    | isTrue  hPtoQ, isTrue  hQtoP => exact isTrue (Iff.intro hPtoQ hQtoP)
```

## 2.4 Classical Logic

The statement `P ∨ ¬P` is a classic example of a proposition that cannot be proven in general without additional assumptions. This is because Lean's logic is based on intuitionistic logic by default, which does not assume that every proposition must be either true or false. In intuitionistic logic, to prove `P ∨ ¬P`, we would need to provide a constructive proof for either `P` or `¬P`, but such a proof does not always exist. Without more information about `P`, there is no general method to construct a proof of either `P` or `¬P`. However, if we wish to work within classical logic in Lean, we can explicitly assume the law of excluded middle as an axiom. Lean provides a mechanism for doing this through the `Classical` namespace. Here's how we can prove `P ∨ ¬P` using classical logic.

```
-- We open the `Classical` namespace
open Classical
-- We use `Classical.em` to prove the excluded middle
theorem ThExcludedMiddle : P ∨ ¬P := by
  exact em P
```

Another important classical equivalence is between `P` and `¬¬P`. In classical logic, this equivalence allows for proving propositions by contradiction. To prove a proposition `P` by contradiction, we assume `¬P` and derive a contradiction. This gives us `¬¬P`, and by the equivalence between `P` and `¬¬P`, we can conclude that `P` is true. This form of reasoning, known as *proof by contradiction*, can be reproduced in Lean by using the `byContradiction` tactic, as demonstrated below.

```
-- Classical Logic allows proofs by contradiction
theorem ThDoubNeg : P ↔ ¬¬P := by
  apply Iff.intro
  -- Implication P → ¬¬P
  intro hP
  intro hNP
  exact hNP hP
  -- Implication ¬¬P → P
  intro hNNP
  have hF : ¬P → False := by
    intro hNP
    exact hNNP hNP
  apply byContradiction hF
```

We observe that in the equivalence between P and ¬¬P, the implication P → ¬¬P holds in intuitionistic logic. However, the converse, ¬¬P → P, is the key result that the `byContradiction` tactic relies on. In intuitionistic logic, we cannot conclude P simply because assuming ¬P leads to a contradiction. Instead, intuitionistic logic only allows us to derive ¬¬P from such a contradiction, meaning that we can assert it is not the case that P is false, but we cannot constructively prove P itself. Therefore, the step from ¬¬P to P (double negation elimination) is not valid in intuitionistic logic, as it goes beyond what is constructively derivable.

An alternative is to use `false_or_by_contra`, which transforms the goal into `False`, switching to classical reasoning if the goal is not decidable.

## 2.5 Exercises

The following exercises are sourced from Daniel Clemente's website.

```
variable (A B C D I L M P Q R : Prop)

theorem T51 (h1 : P) (h2 : P → Q) : P ∧ Q := by sorry

theorem T52 (h1 : P ∧ Q → R) (h2 : Q → P) (h3 : Q) : R := by sorry

theorem T53 (h1 : P → Q) (h2 : Q → R) : P → (Q ∧ R) := by sorry

theorem T54 (h1 : P) : Q → P := by sorry

theorem T55 (h1 : P → Q) (h2 : ¬Q) : ¬P := by sorry

theorem T56 (h1 : P → (Q → R)) : Q → (P → R) := by sorry

theorem T57 (h1 : P ∨ (Q ∧ R)) : P ∨ Q := by sorry

theorem T58 (h1 : (L ∧ M) → ¬P) (h2 : I → P) (h3 : M) (h4 : I) : ¬L := by sorry

theorem T59 : P → P := by sorry

theorem T510 : ¬ (P ∧ ¬P) := by sorry

theorem T511 : P ∨ ¬P := by sorry

theorem T512 (h1 : P ∨ Q) (h2 : ¬P) : Q := by sorry

theorem T513 (h1 : A ∨ B) (h2 : A → C) (h3 : ¬D → ¬B) : C ∨ D := by sorry

theorem T514 (h1 : A ↔ B) : (A ∧ B) ∨ (¬A ∧ ¬B) := by sorry
```

# 3 Quantifiers

This chapter introduces the core concepts of quantifiers in Lean, which are pivotal in expressing logical statements. Quantifiers allow us to make general statements about elements of a type. The universal quantifier (∀) asserts that a property holds for all elements of a type, while the existential quantifier (∃) states that there exists at least one element of the type for which the property holds. Through examples and exercises, this chapter will help us understand how to use these quantifiers effectively in Lean.

## 3.1 Predicates

For simplicity, we will assume that `A` is an arbitrary type and `P` is a predicate on `P`, i.e., `P : A → Prop`, which is a function mapping elements of `A` to logical propositions.

```
variable (A : Type)
variable (P Q : A → Prop)
```

### 3.1.1 Examples of predicates

Given a type `A`, we can define various predicates on it. One trivial example is the predicate that always evaluates to `False`, meaning it never holds for any element of `A`. Similarly, we can define a predicate that always evaluates to `True`, meaning it holds for every element of `A`. These can be expressed as follows:

```
-- False predicate
def PFalse {A : Type} : A → Prop := fun _ => False
-- True predicate
def PTrue {A : Type} : A → Prop := fun _ => True
```

In the definitions above, the underscore `_` is a placeholder for an arbitrary input of type `A`, indicating that the function ignores its argument and always returns a constant value—either `False` or `True`. This underscores the fact that `PFalse` and `PTrue` do not depend on any particular element of `A`, but rather define predicates that are uniformly false or true for all elements of `A`.

### 3.1.2 Operations on predicates

Given two predicates `P, Q : A → Prop`, we can define their **conjunction**, a new predicate that holds for an element `a : A` if and only if both `P a` and `Q a` are true. This is captured by the following definition:

```
-- Conjunction of two predicates
def PAnd {A : Type} (P Q : A → Prop) : A → Prop := by
  intro a
  exact P a ∧ Q a
```

Here, the function `PAnd` takes two predicates `P` and `Q` and returns a new predicate `PAnd P Q` on `A`. This new predicate holds at `a : A` if and only if both `P a` and `Q a` hold.

In Lean, the `notation` keyword allows us to define custom symbolic representations for functions and expressions, improving readability and aligning with standard mathematical conventions. It introduces shorthand notation for existing definitions, making logical and algebraic expressions more intuitive.

For example, we can define a custom infix operator `∧` for the conjunction of two predicates:

```
notation : 65 lhs:65 " ∧ " rhs:66 => PAnd lhs rhs
```

Here, `notation` specifies that `P ∧ Q` should be interpreted as `PAnd P Q`. The numbers `65` and `66` indicate precedence levels, ensuring that expressions involving `∧` are parsed correctly relative to other operators. The `lhs` and `rhs` keywords designate the left-hand side and right-hand side of the notation, ensuring proper binding behavior.

By using `notation`, we can write logical expressions in a way that closely resembles traditional mathematical notation, making proofs and definitions more readable. To verify the notation, we can check the type of `P ∧ Q`:

```
#check P ∧ Q
```

Lean confirms that `P ∧ Q` is a predicate on `A`, reinforcing that this notation correctly represents the conjunction of two predicates.

Building on the previous example, we can similarly define other fundamental logical operations on predicates. These include the disjunction `P ∨ Q`; the implication `P → Q`; the biconditional `P ↔ Q`; and the negation `¬P`. Each of these operations extends our ability to reason about predicates.

## 3.2 Universal Quantifier

The ∀ command (typed as `\forall`) represents the universal quantifier. It is used to express statements of the form `∀ (a : A), P a`, which reads as "for every `a` of type `A`, the proposition `P a` holds." This enables us to make general statements about all elements of a given type. Specifically, if `P` is a predicate on `A`, then `∀ (a : A), P a` is of type `Prop`. The proposition `∀ (a : A), P a` is true if `P a` is true for every element `a` of type `A`. The following three forms serve to denote the universal quantifier in Lean.

```
#check ∀ (a : A), P a
#check ∀ a, P a
#check ∀ {a : A}, P a
```

In the second form, the type of the variable is not explicitly stated, as Lean can infer it from context. In the third form, the quantifier binding is implicit, indicated by curly braces `{}`. This allows Lean to automatically infer the value of `a` whenever possible, reducing the need for explicit annotations.

To prove a statement of the form `∀ (a : A), P a`, we typically use the `intro` tactic (or just write a `lambda` function directly in term mode). This introduces an arbitrary element `a` of type `A` and requires us to prove `P a` for that arbitrary `a`.

```
theorem T1 : ∀ (a : A), P a := by
  intro a
  sorry
```

On the other hand, if we have a hypothesis `h : ∀ (a : A), P a` and we want to use it for a specific value `a : A`, we can apply `h` on `a` to get `P a`.

```
variable (a : X)
variable (h : ∀ (a : A), P a)
#check h a
```

The `specialize` tactic is used to apply a hypothesis that is a universally quantified statement to specific arguments. This allows us to instantiate a general hypothesis with particular values, making it easier to work with in our proof. When we have a hypothesis of the form `h : ∀ (a : A), P a`, we can use specialize to apply `h` to a specific value `a : A`, resulting in a new hypothesis `h : P a`. This is particularly useful when we want to focus on a specific instance of a general statement.

```
theorem T2 (a : A) (h : ∀ (a : A), P a) : P a := by
  specialize h a
  exact h
```

## 3.3 Existential Quantifier

The ∃ command (typed as `\exists`) represents the existential quantifier. It is used to express statements of the form `∃ (a : A), P a`, which reads as "for some `a` of type `A`, the proposition `P a` holds." This enables us to make particular statements about elements of a given type. Specifically, if `P` is a predicate on `A`, then `∃ (a : A), P a` is of type `Prop`. The proposition `∃ (a : A), P a` is true if `P a` is true for some element `a` of type `A`. The following three forms serve to denote the existential quantifier in Lean.

```
#check ∃ (a : A), P a
#check ∃ a, P a
#check Exists P
```

Unlike the universal quantifier, the existential quantifier does not support implicit binding. Attempting to write `∃ {a : A}, P a` results in an error because Lean requires the bound variable `a` to be explicitly declared.

If we `#print Exists`, Lean returns

```
inductive Exists.{u} : α{ : Sort u} → α( → Prop) → Prop
number of parameters: 2
constructors:
Exists.intro : ∀ α{ : Sort u} {p : α → Prop} (w : α), p w → Exists p
```

This code defines the existential quantifier as an inductive type, `Exists`. It has two parameters: `α : Sort u`, the type of the witness, and `p : α → Prop`, the predicate that the witness must satisfy. The function type `(α → Prop) → Prop` ensures that `Exists` takes a predicate `p : α → Prop` and returns a proposition asserting the existence of an element of `α` that satisfies `p`.

The single constructor, `Exists.intro`, constructs a proof of `Exists p` given a witness `a : α` and a proof that `a` satisfies `p`. Lean infers `α` and `p` from context, so to obtain an element of type `Exists p`, it suffices to provide `a` and a proof of `p a`. Here's an example of how the constructor works:

```
theorem T3 (a : A) (h : P a) : ∃ (a : A), P a := by
  exact Exists.intro a h
```

Since `Exists` is an inductive type, we can use `cases` on a proof of this type to extract both the witness and the proof that it satisfies the predicate `P`.

```
variable (Q : Prop)
theorem T4 (h1 : ∃ (a : A), P a) (h2 : ∀ (a : A), P a → Q) : Q := by
  cases h1
  rename_i a h3
  specialize h2 a
  exact h2 h3
```

Another alternative is to use `Exists.elim` the eliminator for the `Exists` type, allowing us to use the witness and the proof of predicate on the witness.

```
theorem T5 (h1 : ∃ (a : A), P a) (h2 : ∀ (a : A), P a → Q) : Q := by
  apply Exists.elim h1
  exact h2
```

## 3.4 Exercises

The following propositions are common identities involving quantifiers.

```
open Classical
variable (a b c : A)
variable (R : Prop)

theorem E1 : ∃( (a : A), R) → R := by sorry

theorem E2 (a : A) : R → ∃( (a : A), R) := by sorry

theorem E3 : ∃( (a : A), P a ∧ R) ↔ ∃( (a : A), P a) ∧ R := by sorry

theorem E4 : ∃( (a : A), (P ∨ Q) a) ↔ ∃( (a : A), P a) ∨ ∃( (a : A), Q a) :=  by sorry

theorem E5 : ∀( (a : A), P a) ↔ ¬∃( (a : A), (¬P) a) := by sorry

theorem E6 : ∃( (a : A), P a) ↔ ¬ ∀( (a : A), (¬P) a) := by sorry

theorem E7 : (∃¬ (a : A), P a) ↔ ∀( (a : A), (¬P) a) := by sorry

theorem E8 : (∀¬ (a : A), P a) ↔ ∃( (a : A), (¬P) a) := by sorry

theorem E9 : ∀( (a : A), P a → R) ↔ ∃( (a : A), P a) → R := by sorry

theorem E10 (a : A) : ∃( (a : A), P a → R) → ∀( (a : A), P a) → R := by sorry

theorem E11 (a : A) : ∃( (a : A), R  →  P a) → (R → ∃ (a : A), P a) :=  by sorry
```

# 4 Equalities

This chapter provides an introduction to the concept of equality in the Lean theorem prover. It explores how equality is defined and utilized in Lean's type theory.

Throughout this chapter, we assume that `X` is an arbitrary type, `x` and `y` are terms of type `X`, as are `a`, `b`, `c`, and `d`. Additionally, `P` is a predicate on `X`, meaning `P : X → Prop`.

```
variable (X : Type)
variable (x y a b c d : X)
variable (P : X → Prop)
```

## 4.1 Equality

Given two terms of any given type we can consider the type of their equality (`Eq`), which is a term of type `Prop`.

```
#check Eq
#check Eq x y
#check x = y
```

If we `#print Eq`, Lean returns

```
inductive Eq.{u_1} : { α : Sort u_1 } → α → α → Prop
number of parameters: 2
constructors:
Eq.refl : ∀ { α : Sort u_1 } (a : α), a = a
```

`Eq` is an inductive type that takes two implicit parameters: a universe level `u_1` and `{α : Sort u_1}`, a type at this universe level. Given any two values of type `α`—the elements being compared for equality—it returns a proposition in `Prop`, asserting their equality.

The negation of an equality `a = b` is expressed in Lean using `Ne` or the symbol `≠` (written as `\neq`). This is simply the negation of the equality proposition, meaning `¬ (a = b)`.

```
#check Ne
#check Ne x y
#check x ≠ y
```

### 4.1.1 Reflexivity

The `Eq` type has a single constructor, `Eq.refl`, which captures the principle of reflexivity: every element is equal to itself. This constructor takes an implicit type `α : Sort u_1` and an element `a : α`, producing a proof of the proposition `a = a`. In other words, it establishes that any element is identical to itself. This is a more powerful theorem than it may appear at first, because although the statement of the theorem is `a = a`, Lean will allow anything that is definitionally equal to that type. So, for instance, `2 + 2 = 4` is proven in Lean by reflexivity.

As a shorthand, we can use `rfl` instead of `Eq.refl`. The key difference is that `rfl` infers `a` implicitly rather than requiring it explicitly. Here's an example demonstrating how the constructor works:

```
theorem TEqRfl (a : X) : a = a := by
  exact rfl

theorem T1 : 2 + 2 = 4 := by
  exact rfl
```

### 4.1.2 Symmetry

If we have `h : a = b` as a hypothesis, we can derive `b = a` using the symmetric property of equality. This is achieved by applying `Eq.symm` to `h`. Alternatively, the shorthand `h.symm` can be used in place of `Eq.symm h` to provide a proof of `b = a`.

```
theorem TEqSymm (h : a = b) : (b = a) := by
  exact Eq.symm h -- also (exact h.symm)
```

### 4.1.3 Transitivity

If we have `h1 : a = b` and `h2 : b = c` as hypotheses, we can derive `a = c` using the transitive property of equality. This is achieved by applying `Eq.trans` to `h1` and `h2`. Alternatively, the shorthand `h1.trans h2` can be used in place of `Eq.trans h1 h2` to provide a proof of `a = c`.

```
theorem TEqTrans (h1 : a = b) (h2 : b = c) : (a = c) := by
  exact Eq.trans h1 h2 -- also (exact h1.trans h2)
```

### 4.1.4 Rewrite

The `rewrite [e]` tactic applies the identity `e` as a rewrite rule to the target of the main goal.

- The rewrite [$e_1$, ..., $e_n$] tactic applies the given rewrite rules sequentially.
- The `rewrite [e] at l` variant applies the rewrite at specific locations `l`, which can be either `*` (indicating all applicable places) or a list of hypotheses in the local context.

We can also use `rw`, which automatically attempts to close the goal by applying `rfl` after performing the rewrite.

```
theorem TEqRw (h1 : a = b) : P b ↔ P a := by
  apply Iff.intro
  -- P b → P a
  intro h2
  rewrite [h1] -- rewrites the goal using h1
  exact h2
  -- P a → P b
  intro h2
  rw [h1] at h2 -- rewrites h2 using h1
  exact h2
```

### 4.1.5 calc

The `calc` command allows for structured reasoning by chaining a sequence of equalities or inequalities. This approach makes multi-step proofs clearer and easier to follow. The general syntax is:

```
calc
  expr_{1}  = expr_{2}      := justification_{1}
  _         = expr_{3}      := justification_{2}
  _         = expr_{4}      := justification_{3}

  _         = expr_{n}      := justification_{n-1}
```

Here:

- `expr_1, expr_2, expr_3, ...` are expressions.
- `justification_1, justification_2, ...` are proofs or explanations that establish each equality or inequality.
- The `_` syntax connects the steps, ensuring a logical flow.

Here's an example demonstrating the use of the `calc` command.

```
theorem TCalc (h1 : a = b) (h2 : b = c) (h3: c = d) : (a = d) := by
  calc
    a = b := by rw [h1]
    _ = c := by rw [h2]
    _ = d := by rw [h3]
```

## 4.2 Types with meaningful equality

Equality (=) is a fundamental concept in Lean, defined for all types. However, its interpretation and behavior depend on the structure of the specific type. While equality is always available, its computational properties—such as whether it is decidable—vary depending on the type.

```
#eval x = y -- Returns error
#eval 2 + 2 = 4 -- Returns true
```

### 4.2.1 Decidable Equality

A type has *decidable equality* if there exists an algorithm to determine whether any two elements of that type are equal. In Lean, this is captured by the `DecidableEq` type class.

If we `#print DecidableEq` Lean returns

```
@[reducible] def DecidableEq.{u} : Sort u → Sort (max 1 u) :=
fun α => (a b : α) → Decidable (a = b)
```

An example of a type with decidable equality is `Bool`. The following proof defines an instance of `DecidableEq` for booleans:

```
def DecidableEqBool : DecidableEq Bool := by
  intro a b
  match a, b with
    | false, false => exact isTrue rfl
    | false, true  => exact isFalse (fun h => Bool.noConfusion h)
    | true , false => exact isFalse (fun h => Bool.noConfusion h)
    | true , true  => exact isTrue rfl
```

In Lean, the `noConfusion` principle is a powerful tool for reasoning about inductive types. It captures two essential properties of constructors: they are *disjoint* (no two distinct constructors can produce equal values) and *injective* (equal constructor applications imply equal arguments). For the type `Bool`, which has exactly two constructors—`true` and `false`—these properties mean that `true ≠ false` and `false ≠ true`. The expression `Bool.noConfusion h` exploits this fact: when we assume a contradictory equality like `true = false`, `noConfusion` produces a logical contradiction, allowing us to conclude that such an assumption is invalid. More generally, `noConfusion` can be used to eliminate impossible equalities between constructors or to extract equalities of their arguments when constructors match.

Some other examples of types with decidable equality include:

- Basic types such as `Nat`, `Int`, and `String`, which all have decidable equality.
- Inductive types and structures, provided that their components also have decidable equality.

```
#check Nat.decEq
#check Int.decEq
#check String.decEq
```

Consider the following functions.

```
-- Function Charp
def Charp : Nat → Nat → Bool := by
  intro n m
  by_cases n = m
  -- Case n = m
  exact true
  -- Case n ≠ m
  exact false

-- Function Charp2
def Charp2 : Nat → Nat → Bool := fun n m => if n = m then true else false

-- Function Charpoint
noncomputable def Charpoint {A : Type} : A → A → Bool := by
  intro a b
  by_cases a = b
  -- Case a = b
  exact true
```

```
  -- Case a ≠ b
  exact false

-- Function Charpoint2
def Charpoint2 {A : Type} [DecidableEq A] : A → A → Bool :=
fun n m => if n = m then true else false
```

The function `Charp` takes two natural numbers, `n` and `m`, and determines whether they are equal. It returns `true` if `n = m` and `false` otherwise. This is achieved using the `by_cases` tactic, which performs a case distinction: if `n = m`, the function returns `true`, and if `n ≠ m`, it returns `false`. An alternative implementation, `Charp2`, expresses the same function using an `if ... then ... else` expression.

In general, equality in an arbitrary type is not necessarily computable. For instance, in the function `Charpoint`, which generalizes `Charp` to an arbitrary type `A`, the use of `by_cases a = b` introduces a logical case distinction that may not be computable. Consequently, the function is marked as `noncomputable`, indicating that it relies on classical reasoning rather than constructive computation. To ensure computability, the alternative function `Charpoint2` explicitly assumes that `A` has decidable equality by requiring the type class instance `[DecidableEq A]`. This assumption allows Lean to treat equality on `A` as a computable procedure, ensuring that the function remains fully computable.

### 4.2.2 Equality in `Prop`

In `Prop`, the type of propositions, equality is defined in terms of logical equivalence, as stated by the axiom of *propositional extensionality*, written `propext`. An *axiom* is a fundamental assumption accepted without proof.

If we `#print propext`, Lean returns:

```
axiom propext : ∀ {a b : Prop}, (a ↔ b) → a = b
```

This means that if two propositions `a` and `b` are logically equivalent (`a ↔ b`), then they are considered equal. Note that `propext` is an axiom. Axioms are accepted by definition, rather than being derived from existing theorems. This axiom enables the substitution of equivalent propositions within any context. However, unlike equality for concrete types such as `Nat` or `Int`, logical equivalence is generally undecidable—determining whether two arbitrary propositions are equivalent is, in general, an undecidable problem.

```
theorem TEqProp {Q : Prop} : (Q ∧ True) = Q := by
  apply propext
  apply Iff.intro
  -- Q ∧ True → Q
  intro h2
  exact h2.left
  -- Q → Q ∧ True
  intro h2
  apply And.intro
  exact h2
  trivial
```

We can always inspect the axioms upon which a theorem relies by using the `#print axioms` command. For instance, to check the axioms involved in the `TEqProp` theorem, we can run:

```
#print axioms TEqProp
```

The above code returns `'TEqProp' depends on axioms: [propext]`.

# 5 Functions

In Lean, a **function** is a relation that associates each element of one type (the domain) with a unique element of another type (the codomain). This concept is foundational in both mathematics and programming. In this chapter, we delve into how functions are represented and utilized in Lean. Key topics include:

- The definition and notation of functions in Lean.
- Function composition and application.
- Special types of functions, such as injective (one-to-one), surjective (onto), and bijective (one-to-one and onto) functions, and their significance in mathematical reasoning.

By the end of this chapter, we will have a basic foundation for defining, manipulating, and formally reasoning about functions in Lean.

Throughout this chapter, we assume that `A`, `B`, `C` and `D` are arbitrary types

```
variable (A B C D : Type)
```

Given two types, `A` and `B`, the expression `A → B` denotes the type of all functions from `A` to `B`. The arrow `→` is written in Lean using `\to`. In this context, `A` is called the *domain*, and `B` the *codomain*. Each element `f` of `A → B` is a function. If `a : A` is an element of the domain, then `f a` represents its image under `f` and belongs to `B`. In Lean, function application does not require parentheses, making the syntax more natural and readable. Instead of writing `f(a)`, as in many programming languages, Lean uses `f a`.

```
-- The type of all functions from A to B
#check A → B

-- Declare functions f and g
variable (f g : A → B)

-- Declare an element a of type A
variable (a : A)

-- This is an element of type B
#check f a
```

### 5.0.1 Equality

The function type `A → B` comes with a natural notion of equality. *Function extensionality*, expressed by `funext`, states that if two functions with the same domain and codomain produce the same output for every input, then they are equal:
`(∀ (a : A), f a = g a) → f = g`. In many dependent type theory systems, function extensionality is an axiom, as it cannot be derived from the core logic alone. Conversely, if two functions are equal, then they yield the same result for every input:
`f = g → ∀ (a : A), f a = g a`. This follows from the *congruence property* of functions, implemented in Lean as `congrFun`. The following example illustrates both principles:

```
theorem TEqApl : f = g ↔ ∀ (a : A), f a = g a := by
  apply Iff.intro
  -- f = g → ∀ (a : A), f a = g a
  intro h a
  exact congrFun h a
  -- ∀( (a : X), f a = g a) → f = g
  intro h
  exact funext h
```

### 5.0.2 Composition

If `f : A → B` and `h : B → C` are functions, their composition, written as `h ∘ f`, is a function of type `A → C`. In Lean, the composition operator `∘` is written using `\comp` or `\circ`.

```
variable (h : B → C)
#check h ∘ f
```

Composition of functions is associative.

```
theorem TCompAss {A B : Type} {f : A → B} {g : B → C} {h : C → D} : h ∘ (g ∘ f) = (h ∘ g) ∘ f := by
  funext a
  exact rfl
```

### 5.0.3 Identity function

For any type `A`, we can define the *identity function* `id`, which has type `A → A`. This function simply returns its input unchanged, meaning that for every `a : A`, we have `id a = a`.

If we `#print id`, Lean returns:

```
def id.{u} : { α : Sort u } → α → α := fun { α } a => a
```

This definition shows that `id` takes an implicit argument `{α : Sort u}`—a type in any universe—and an explicit argument `a : α`, returning `a` unchanged.

The identity function serves as a neutral element for function composition, meaning that composing any function with `id`, whether on the left or the right, leaves the function unchanged.

```
theorem TIdNeutral : (f ∘ id = f) ∧ (id ∘ f = f) := by
  apply And.intro
  -- f ∘ id = f
  funext a
  exact rfl
  -- id ∘ f = f
  funext a
  exact rfl
```

Observe that in the proof above, the equality `f ∘ id = f` uses `id` to denote the identity function on `A`, while in the equality `id ∘ f = f`, `id` refers to the identity function on `B`. This proof showcases Lean's capabilities for type inference.

If we want to explicitly specify the domain of the identity function, we can disable the automatic insertion of implicit parameters by using the `@` symbol before `id`. For example, `@id A` returns the identity function defined on the type `A`.

```
#check @id A
```

## 5.1 Injections

In this section, we introduce the concepts of injective function, monomorphism, and left inverse of a function, and we examine their relationships and key properties.

#### Injective

We say that a function `f : A → B` is *injective* or *one-to-one* if, for all pairs of elements `a1, a2 : A`, the equality `f a1 = f a2` implies `a1 = a2`.

```
def injective {A B : Type} (f : A → B) : Prop := ∀{a1 a2 : A}, (f a1 = f a2) → (a1 = a2)
```

#### Monomorphism

We say that a function `f : A → B` is a *monomorphism* if, for every other type `C` and every pair of functions `g h : C → A`, the equality `f ∘ g = f ∘ h` implies that `g = h`.

```
def monomorphism {A B : Type} (f : A → B) : Prop := ∀{C : Type}, ∀{g h : C → A}, f ∘ g = f ∘ h → g = h
```

#### Left inverse

We say that a function `f : A → B` has a *left inverse* if there exists a function `g : B → A` such that `g ∘ f = id`.

```
def hasleftinv {A B : Type} (f : A → B) : Prop := ∃(g : B → A), g ∘ f = id
```

### 5.1.1 An example: The identity

Every identity is injective, a monomorphism and has a left inverse (the identity itself).

```
-- The identity is injective
theorem TIdInj : injective (@id A) := by
  -- rw [injective] -- rw to recover the definition
  intro a1 a2 h
  calc
    a1 = id a1 := by exact rfl
    _  = id a2 := by exact h
    _  = a2    := by exact rfl

-- The identity is a monomorphism
theorem TIdMon : monomorphism (@id A) := by
  -- rw [monomorphism] -- rw to recover the definition
  intro C g h h1
  calc
    g = id ∘ g := by exact rfl
    _ = id ∘ h := by exact h1
    _ = h      := by exact rfl

-- The identity has a left inverse
theorem TIdHasLeftInv : hasleftinv (@id A) := by
  -- rw [hasleftinv] -- rw to recover the definition
  apply Exists.intro id
  exact rfl
```

### 5.1.2 Exercises

```
-- Negation of injective
theorem TNegInj {A B : Type} {f : A → B} : ¬ (injective f) ↔ ∃(a1 a2 : A), f a1 = f a2 ∧ a1 ≠ a2 := by
    sorry

-- The composition of injective functions is injective
theorem TCompInj {A B : Type}  {f : A → B} {g : B → C} (h1 : injective f) (h2 : injective g) : injective
      (g ∘ f) := by sorry

-- If the composition (g ∘ f) is injective, then f is injective
theorem TCompRInj {A B : Type} {f : A → B} {g : B → C} (h1 : injective (g ∘ f)) : (injective f) := by
    sorry

-- Injective and Monomorphism are equivalent concepts
theorem TCarMonoInj {A B : Type} {f : A → B} : injective f ↔ monomorphism f := by sorry

-- If a function has a left inverse then it is injective
theorem THasLeftInvtoInj {A B : Type} {f : A → B} : hasleftinv f → injective f := by sorry
```

## 5.2 Surjections

In this section, we introduce the concepts of surjective function, epimorphism, and right inverse of a function, and we examine their relationships and key properties.

#### Surjective

We say that a function `f : A → B` is *surjective* or *onto* if, for every element `b : B`, there exists an element `a : A` such that `f a = b`.

```
def surjective {A B : Type} (f : A → B) : Prop := ∀{b : B}, ∃(a : A), f a = b
```

### Epimorphism

We say that a function `f : A → B` is an *epimorphism* if, for every other type `C` and every pair of functions `g h : B → C`, the equality `g ∘ f = h ∘ f` implies that `g = h`.

```
def epimorphism {A B : Type} (f : A → B) : Prop := ∀{C : Type}, ∀{g h : B → C}, g ∘ f = h ∘ f → g = h
```

### Right inverse

We say that a function `f : A → B` has a *right inverse* if there exists a function `g : B → A` such that `f ∘ g = id`.

```
def hasrightinv {A B : Type} (f : A → B) : Prop := ∃(g : B → A), f ∘ g = id
```

### 5.2.1 An example: The identity

Every identity is surjective, an epimorphism and has a right inverse (the identity itself).

```
-- The identity is surjective
theorem TIdSurj : surjective (@id A) := by
  -- rw [surjective] -- rw to recover the definition
  intro a
  apply Exists.intro a
  exact rfl

-- The identity is an epimorphism
theorem TIdMon : epimorphism (@id A) := by
  -- rw [epimorphism] -- rw to recover the definition
  intro C g h h1
  calc
    g = g ∘ id := by exact rfl
    _ = h ∘ id := by exact h1
    _ = h      := by exact rfl

-- The identity has a right inverse
theorem TIdHasRightInv : hasrightinv (@id A) := by
  -- rw [hasrightinv] -- rw to recover the definition
  apply Exists.intro id
  exact rfl
```

### 5.2.2 Exercises

```
-- Negation of surjective
theorem TNegSurj {A B : Type} {f: A → B} : ¬ (surjective f) ↔ ∃(b : B), ∀ (a : A), f a ≠ b := by sorry

-- The composition of surjective functions is surjective
theorem TCompSurj {A B : Type} {f : A → B} {g : B → C} (h1 : surjective f) (h2 : surjective g) :
    surjective (g ∘ f) := by sorry

-- If the composition (g ∘ f) is surjective, then g is surjective
theorem TCompLSurj {A B : Type} {f : A → B} {g : B → C} (h1 : surjective (g ∘ f)) : (surjective g) := by
    sorry

-- Surjective and Epimorphism are equivalent concepts
theorem TCarEpiSurj {A B : Type} {f : A → B} : surjective f ↔ epimorphism f := by sorry

-- If a function has a right inverse then it is surjective
theorem THasRightInvtoInj {A B : Type} {f : A → B} : hasrightinv f → surjective f := by sorry
```

## 5.3 Bijections

In this section, we introduce the concepts of bijective function and isomorphism and we examine their relationships and key properties.

#### Bijective

We say that a function `f : A → B` is *bijective* if it is injective and surjective.

```
def bijective {A B : Type} (f : A → B) : Prop := injective f ∧ surjective f
```

#### Isomorphism

We say that a function `f : A → B` is an *isomorphism* if there exists a function `g : B → A` such that `g ∘ f = id ∧ f ∘ g = id`.

```
def isomorphism {A B : Type} (f : A → B) : Prop := ∃ (g : B → A), g ∘ f = id ∧ f ∘ g = id
```

### 5.3.1 An example: The identity

Every identity is bijective and an isomorphism.

```
-- The identity is bijective
theorem TIdBij : bijective (@id A) := by
  -- rw [bijective] -- rw to recover the definition
  apply And.intro
  exact TIdInj A
  exact TIdSurj A

-- The identity is an isomorphism
theorem TIdMon : isomorphism (@id A) := by
  rw [isomorphism] -- rw to recover the definition
  apply Exists.intro id
  apply And.intro
  exact rfl
  exact rfl
```

### 5.3.2 Exercises

```
-- The composition of bijective functions is bijective
theorem TCompBij {A B : Type} {f : A → B} {g : B → C} (h1 : bijective f) (h2 : bijective g) : bijective
    (g ∘ f) := by sorry

-- A function is an isomorphism if and only if it has left and right inverse
theorem TCarIso {A B : Type} {f : A → B} : isomorphism f ↔ (hasleftinv f ∧ hasrightinv f) := by sorry

-- Every isomorphism is bijective
theorem TCarIsotoBij {A B : Type} {f : A → B} : isomorphism f → bijective f := by sorry
```

# 6 Natural numbers

The natural numbers—0, 1, 2, and so on—form the foundation of mathematics. In Lean and other proof assistants, natural numbers aren't taken for granted; instead, they are built from the ground up using *inductive types*. This approach not only mirrors their intuitive construction but also unlocks powerful tools for reasoning about them formally.

In this chapter, we'll explore how the natural numbers are defined inductively in Lean, and how such a definition allows us to reason about them using *case analysis* and *mathematical induction*. We'll also examine how to define functions on natural numbers using *recursion*, and use this technique to construct the familiar operations of *maximum* and *minimum*, *addition* and *multiplication*. The chapter concludes with exercises to reinforce our understanding and help us apply these concepts.

## 6.1 Definition

The natural numbers `Nat` are defined inductively in Lean using two constructors: `zero`, which represents the base case, and `succ`, which takes a natural number and returns its successor. This closely follows the Peano axioms, where `0` is a natural number and, if `n` is a natural number, then so is `n + 1`.

If we `#print Nat`, Lean returns:

```
inductive Nat : Type
number of parameters: 0
constructors:
Nat.zero : Nat
Nat.succ : Nat → Nat
```

Inductive types like `Nat` not only specify how values are built, but also provide fundamental principles of recursion and induction. These principles allow functions to be defined by pattern matching on constructors, and proofs to be carried out using structural induction.

Lean comes with the `Nat` type already implemented, along with many theorems related to natural numbers. However, to gain a deeper understanding of how natural numbers can be constructed and reasoned about, we will define our own custom type, which we will simply call `N`.

```
inductive N : Type where
  | z : N
  | s : N →  N
deriving Repr
```

The `deriving Repr` clause is used to automatically generate an instance of the `Repr` type class for a user-defined type. The `Repr` class defines how values of a type can be converted into a human-readable format, primarily for the purpose of displaying them during evaluation or debugging. When a type derives `Repr`, Lean synthesizes the necessary code to produce a structured string representation of any value of that type. This is particularly useful when using commands like `#eval`, where Lean attempts to evaluate an expression and display the result. Without a `Repr` instance, Lean would not know how to present the value, resulting in an error. By including `deriving Repr` in a type declaration, users enable Lean to show values automatically, making it easier to inspect the behaviour of programs and proofs.

If we `#print N`, Lean returns:

```
inductive N : Type
number of parameters: 0
constructors:
N.z : N
N.s : N → N
```

We can access the two constructors of `N` with `N.z` and `N.s`. To work with `N` without needing to prefix everything with `N.`, we can open the namespace to bring its notation into scope.

```
open N
#check z
#check s
```

## 6.2 Cases

Assuming `x` is a variable in the local context with an inductive type, `cases x` splits the main goal, producing one goal for each constructor of the inductive type, in which the target is replaced by a general instance of that constructor.

The code below defines a function `Eqzero` in Lean, which takes a natural number `n` of type `N` and returns a boolean value in `Bool`. The purpose of this function is to compare the given number `n` with zero. The function is defined using `cases` over the structure of a natural number `n`, used to perform a case analysis. This splits the proof into two cases:

1. **Case Zero**: When `n` is zero, represented by `z`, the function returns `true`, indicating that `n` is equal to zero.

2. **Case Successor**: When `n` is the successor `s` of some natural number `m` the function returns `false`, indicating that `n` is not equal to zero.

```
def Eqzero : N → Bool := by
  intro n
  cases n
  -- Case zero
  exact true
  -- Case successor
  exact false
```

## 6.3 Match

Let us recall that an alternative to using `cases` is the `match` expression, which enables us to perform pattern matching directly within a definition. In what follows, we will define an alternative version of `Eqzero` using this approach.

```
def Eqzero2 : N → Bool := by
  intro n
  match n with
    | z    => exact true
    | s _  => exact false
```

## 6.4 Dedekind-Peano

### 6.4.1 Cases

Note that the type `N` gives us a Dedekind–Peano algebra. We can think of this type as the free algebra generated by a constant `z` and a unary operation `s`. In this setup, `z` can never be equal to `s n` for any `n : N`.

```
theorem TZInj : ∀ (n : N), z ≠ s n := by
  intro n
  intro h
  cases h
```

In the proof of the theorem `NInj`, the first step `intro n` introduces an arbitrary element `n` of type `N`. This sets the stage for proving that `z` is not equal to `s n` for any such `n`. The next step, `intro h`, assumes the contrary—that is, it introduces a hypothesis `h : z = s n`. To analyze this equality, we apply the tactic `cases h`, which attempts to decompose the equation. However, since `z` and `s n` are built using different constructors of the inductive type `N`, Lean can immediately determine that this equality is impossible. This is a consequence of the fact that constructors of an inductive type are *disjoint*—they produce values that can never be equal. As a result, Lean closes the goal automatically, completing the proof.

### 6.4.2 Injection

Similarly, the successor function `s` is injective. For this we use the tactic `injection` which states that constructors of inductive data types are injective.

```
theorem TSuccInj : injective s := by
  intro n m
  intro h
  injection h
```

### 6.4.3 noConfusion

The `noConfusion` principle formalizes the fact that the different constructors of an inductive type are distinct and that they are injective when applied to arguments. Here's how it works in practice:

```
theorem TSuccInjAlt : injective s := by
  intro n m h
  exact N.noConfusion h id
```

In this proof, we're saying: if `s n = s m`, then—by the injectivity of the `s` constructor—we must have `n = m`. The `noConfusion` principle takes the equality `h : s n = s m` and safely removes the constructors, handing us the equality `n = m` underneath. This is especially useful when we want to avoid manual pattern matching with `cases` or `match`, and instead reason abstractly about the structure of our inductive values.

## 6.5 Induction

Assuming `x` is a variable of inductive type in the local context, the tactic `induction x` applies induction on `x` to the main goal. This results in one subgoal for each constructor of the inductive type, where the target is replaced by a general instance of that constructor. For each recursive argument of the constructor, an inductive hypothesis is introduced. If any element in the local context depends on `x`, it is reverted and then reintroduced after the induction, ensuring that the inductive hypothesis properly incorporates these dependencies.

Next theorem proves that a predicate holds for every natural number.

```
theorem TInd {P : N → Prop} (h0 : P z) (hi : ∀ (n : N), P n → P (s n)) : ∀ (n : N), P n := by
  intro n
  induction n
  -- Base case: `z`
  exact h0
  -- Inductive step: assume the property holds for `n`, and prove it for `s n`
  rename_i n hn
  exact (hi n) hn
```

The proof proceeds by induction on `n`:

1. **Base case**: We begin with the case `z`. Here, we need to show that `P z` holds. But this is precisely what `h0` provides, so the base case is established.

2. **Inductive step**: For the inductive case, we assume a natural number `n` and the inductive hypothesis `hn : P n`, which states that the property holds for `n`. Our goal is to show that `P (s n)` holds. This follows directly from the hypothesis `hi`.

## 6.6 Recursion

Recursion is a fundamental concept in both mathematics and computer science, allowing us to define functions in terms of simpler instances of themselves. In the context of natural numbers, recursive definitions mirror the inductive structure of the numbers themselves. This structure lends itself naturally to recursive functions, where we specify the result for the base case and describe how to compute the result for a successor in terms of the result for its predecessor. In this subsection, we will explore how recursion works in Lean, with simple examples like the definition of the maximum and the minimum, the addition and the multiplication.

### 6.6.1 Maximum

Recursive definition of the maximum of two natural numbers.

```
def max : ℕ → ℕ → ℕ := by
  intro n m
  match n, m with
    | z, m  => exact m
    | n, z  => exact n
    | s n', s m' => exact s (max n' m')
```

### 6.6.2 Minimum

Recursive definition of the minimum of two natural numbers.

```
def min : ℕ → ℕ → ℕ := by
  intro n m
  match n, m with
    | z, _  => exact z
    | _, z  => exact z
    | s n', s m' => exact s (min n' m')
```

### 6.6.3 Addition

We now define the function `Addition`, which recursively specifies the addition of natural numbers by recursion on the first argument. For clarity and readability, we will use the shorthand notation `n + m` in place of `Addition n m`.

```
def Addition : ℕ → ℕ → ℕ := by
  intro n m
  cases n with
  | z   => exact m
  | s n => exact s (Addition n m)
-- Notation for Addition
notation : 65 lhs:65 " + " rhs:66 => Addition lhs rhs
```

`Addition` takes two natural numbers `n` and `m` as input and computes their addition by recursion on `n`. The base case handles the situation when `n` is zero in which case the result is simply `m`, since adding zero to any number yields that number. In the inductive case, where we consider `s n`, the successor of `n`, the function returns the successor of the recursive addition of `n` and `m`, that is, `s (Addition n m)`. This reflects the intuitive idea that to compute `(s n) + m`, we first compute `n + m` and then take its successor.

#### Fibonacci

Thanks to `Addition` we can define the Fibonacci function recursively.

```
def Fib : ℕ → ℕ := by
  intro n
  match n with
    | z       => exact z
    | s z     => exact (s z)
    | s (s n) => exact n + (s n)
```

### 6.6.4 Multiplication

With a similar idea, we can define the function `Multiplication`, which recursively specifies the multiplication of natural numbers by recursion on the first argument. For clarity and readability, we will use the shorthand notation `n * m` in place of `Multiplication n m`.

```
def Multiplication : ℕ → ℕ → ℕ := by
  intro n m
  cases n with
  | z   => exact z
  | s n => exact (Multiplication n m) + m
-- Notation for Multiplication
notation : 70 lhs:70 " * " rhs:71 => Multiplication lhs rhs
```

Multiplication takes two natural numbers n and m as input and computes their addition by recursion on n. The base case handles the situation when n is zero in which case the result is simply z, since multiplying zero to any number yields zero. In the inductive case, where we consider s n, the successor of n, the function returns the sum of the recursive multiplication of n and m and m, that is, (Multiplication n m) + m. This reflects the intuitive idea that to compute (s n) * m, we first compute n * m and then add m.

#### Factorial

Thanks to Multiplication we can define the factorial function recursively.

```
def Fact : N → N := by
  intro n
  cases n with
    | z   => exact (s z)
    | s n => exact (s n) * (Fact n)
```

## 6.7 Decidable Equality

Thanks to the inductive structure of N, we can define a recursive procedure to determine whether two values of type N are equal. This means providing an instance of the DecidableEq type class for N. The idea is simple: we compare two values by structurally analyzing their form—whether they are both zero, both successors, or mismatched. In the case of successors, we reduce the problem to their predecessors and apply the same logic recursively. Here's how this can be implemented:

```
def instDecidableEqN : DecidableEq N := by
  intro n m
  match n, m with
    | z, z        => exact isTrue  rfl
    | z, s _      => exact isFalse (by intro h; cases h)
    | s _, z      => exact isFalse (by intro h; cases h)
    | s n', s m'  =>
      match instDecidableEqN n' m' with
      | isTrue h  => exact isTrue (congrArg s h)
      | isFalse h => exact isFalse (fun h' => N.noConfusion h' (fun h'' => h h''))
```

## 6.8 Exercises

### 6.8.1 Injection

```
-- Prove that no natural number is equal to its own successor
theorem TInjSucc {n : N} : ¬ (n = s n) := by sorry
```

### 6.8.2 Maximum

```
-- Max (z, n) = n
theorem TMaxzL : ∀ {n : N}, (maxi z n) = n := by sorry

-- Max (n, z) = n
theorem TMaxzR : ∀ {n : N}, (maxi n z) = n := by sorry

-- Max (n, m) = Max (m, n)
theorem TMaxComm : ∀ {n m : N}, (maxi n m) = (maxi m n) := by sorry

-- Max (n, m) = n ∨ Max (n, m) = m
theorem TMaxOut : ∀ {n m : N}, ((maxi n m) = n) ∨ ((maxi n m) = m) := by sorry

-- Max (n, n) = n
theorem TMaxIdpt : ∀ {n : N}, maxi n n = n := by sorry
```

### 6.8.3 Minimum

```
-- Min (z, n) = z
theorem TMinzL : ∀ {n : N}, (mini z n) = z := by sorry

-- Min (n, z) = z
theorem TMinzR : ∀ {n : N}, (mini n z) = z := by sorry

-- Min (n, m) = Min (m, n)
theorem TMinComm : ∀ {n m : N}, (mini n m) = (mini m n) := by sorry

-- Min (n, m) = n ∨ Min (n, m) = m
theorem TMinOut : ∀ {n m : N}, ((mini n m) = n) ∨ ((mini n m) = m) := by sorry

-- Min (n, n) = n
theorem TMinIdpt : ∀ {n : N}, mini n n = n := by sorry

-- Min (n, m) = Max (n, m) → n = m
theorem TMinMaxEq : ∀ {n m : N}, mini n m = maxi n m → n = m := by sorry

-- Min (n, m) = n ↔ Max (n, m) = m
theorem TMinMax : ∀ {n m : N}, mini n m = n ↔ maxi n m = m := by sorry
```

### 6.8.4 Addition

```
-- z is a left identity for addition
theorem TAdd0L : ∀ {n : N}, z + n = n := by sorry

-- z is a right identity for addition
theorem TAdd0R : ∀ {n : N}, n + z = n := by sorry

-- Addition of natural numbers is commutative up to a successor
theorem TAddOne : ∀ {n m : N}, (s n) + m = n + (s m) := by sorry

-- Addition is commutative
theorem TAddComm : ∀ {n m : N}, n + m = m + n := by sorry

-- If the sum of two natural numbers is zero, then the first number must be zero
theorem TAddZ : ∀ {n m : N}, n + m = z → n = z := by sorry

-- If the sum of two natural numbers is zero, then both numbers are zero
theorem TAddZ2 : ∀ {n m : N}, n + m = z → (n = z) ∧ (m = z) := by sorry

-- Addition is associative
theorem TAddAss : ∀{n m p : N}, (n + m) + p = n + (m + p) := by sorry

-- n can never be equal to n + s k
theorem TAddSucc : ∀ {n k : N}, n = n + (s k) → False := by sorry

-- A number cannot be both ahead of and behind another number by a positive amount
theorem TIncAdd : ∀ {n m k : N}, m = n + (s k) →  n = m + (s k) → False := by sorry

-- Right congruence of addition
theorem TAddCongR : ∀ {n m k : N}, m = k → n + m = n + k := by sorry

-- Left congruence of addition
theorem TAddCongL : ∀ {n m k : N}, m = k → m + n = k + n := by sorry

-- Addition on the left is cancellative
theorem TAddCancL : ∀ {n m k : N}, n + m = n + k → m = k := by sorry

-- Addition on the right is cancellative
theorem TAddCancR : ∀ {n m k : N}, m + n = k + n → m = k := by sorry

-- Left cancellation property of addition with zero
theorem TAddCancLZ : ∀ {n m : N}, n + m = n → m = z := by sorry

-- Right cancellation property of addition with zero
theorem TAddCancRZ : ∀ {n m : N}, m + n = n → m = z := by sorry
```

### 6.8.5 Multiplication

```
-- z is a left zero for multiplication
theorem TMult0L : ∀ {n : N}, z ∗ n = z := by sorry

-- z is a right zero for multiplication
theorem TMult0R : ∀ {n : N}, n ∗ z = z := by sorry

-- We introduce `one`
def one : N := s z

-- one + n = s n
theorem TOneAddR : ∀ {n : N}, one + n = s n := by sorry

-- n + one = s n
theorem TOneAddL : ∀ {n : N}, n + one = s n := by sorry

-- The different cases for two numbers adding to one
theorem TAddOneCases : ∀ {n m : N}, n + m = one → (n = z ∧ m = one) ∨
(n = one ∧ m = z) := by sorry

-- one is a left identity for multiplication
theorem TMult1L : ∀ {n : N}, one ∗ n = n := by sorry

-- one is a right identity for multiplication
theorem TMult1R : ∀ {n : N}, n ∗ one = n := by sorry

-- Multiplication is left distributive over addition
theorem TMultDistL : ∀ {n m k : N}, (n + m) ∗ k = (n ∗ k) + (m ∗ k) := by sorry

-- Multiplication is right distributive over addition
theorem TMultDistR : ∀ {n m k : N}, n ∗ (m + k) = (n ∗ m) + (n ∗ k) := by sorry

-- Multiplication is commutative
theorem TMultComm : ∀ {n m : N}, n ∗ m = m ∗ n := by sorry

-- If the product of two natural numbers is zero, then one of them must be zero
theorem TMultZ : ∀ {n m : N}, n ∗ m = z → (n = z) ∨ (m = z) := by sorry

-- Right congruence of multiplication
theorem TMultCongR : ∀ {n m k : N}, m = k → n ∗ m = n ∗ k := by sorry

-- Left congruence of addition
theorem TMultCongL : ∀ {n m k : N}, m = k → m ∗ n = k ∗ n := by sorry

-- Multiplication is associative
theorem TMultAss : ∀{n m p : N}, (n ∗ m) ∗ p = n ∗ (m ∗ p) := by sorry

-- Fix points for multiplication
theorem TMultFix : ∀{n m : N}, n ∗ m = n → n = z ∨ m = one := by sorry

-- One is the unique idempotent for multiplication
theorem TMultOne : ∀ {n m : N}, n ∗ m = one ↔ (n = one ∧ m = one) := by sorry
```

# 7 Choice

Reasoning about types often requires distinguishing between those that are merely **nonempty** and those that are **inhabited**. While both concepts assert the existence of elements in a type, they differ in logical strength and computational implications. A type is **nonempty** if it contains at least one element, but this existence is not necessarily constructive—it does not provide an explicit example. In contrast, a type is **inhabited** if we can specify a concrete default element, making it more useful in computational contexts. This distinction becomes particularly significant when comparing **constructive** and **classical** reasoning. In constructive logic, knowing that a type is nonempty does not guarantee that we can extract an element from it, whereas in classical logic, the axiom `Classical.choice` allows us to select an element from a nonempty type, thereby making it inhabited.

In this chapter, we will explore the definitions of `Inhabited` and `Nonempty`, examine their properties, and analyze their relationship. We will also discuss the role of `Classical.choice` in bridging the gap between nonemptiness and inhabitation, along with the implications of relying on nonconstructive principles in formal proofs.

## 7.1 Inhabited types

The `Inhabited α` typeclass ensures that the type `α` has a designated element, known as `default : α`. This property is sometimes referred to as making `α` a “pointed type.”

If we `#print Inhabited`, Lean returns:

```
class Inhabited.{u} : Sort u → Sort (max 1 u)
number of parameters: 1
constructor:
Inhabited.mk : { α : Sort u } → α → Inhabited α
fields:
default : α
```

A *typeclass* is a special kind of structure that defines a set of properties or operations that a type can possess. We will explore typeclasses further in later chapters. The definition above shows that `Inhabited` is a typeclass with a single parameter, `α`, and a constructor, `Inhabited.mk`, which takes an element of `α` and produces an instance of `Inhabited α`. The `default` field provides access to this designated element.

To define a specific instance of a type class, we use the `instance` or `def` keywords. For the `Inhabited` typeclass, we only need to specify a default element. In the example below, we declare an instance of `Inhabited Bool` where the default value is `true`.

```
instance InBool : Inhabited Bool := { default := true }

#check InBool -- returns InBool : Inhabited Bool
#print InBool -- returns def InBool : Inhabited Bool := { default := true }
#eval InBool.default -- returns true
```

Lean provides predefined instances of the `Inhabited` type class for several types. These instances specify a default value for each type.

```
#print instInhabitedBool -- default := false
#print instInhabitedProp -- default := True
#print instInhabitedNat  -- default := Nat.zero
```

In these cases, `Bool` has `false` as its default value, `Prop` has `True`, and `Nat` has `0`. These defaults ensure that each type has at least one canonical element available.

## 7.2 Nonempty

The `Nonempty` type is an inductive proposition that asserts the existence of at least one element in a given type.

If we `#print Nonempty`, Lean returns

```
inductive Nonempty.{u} : Sort u → Prop
number of parameters: 1
constructors:
Nonempty.intro : ∀ { α : Sort u }, α → Nonempty α
```

This means that `Nonempty` is an inductive type that requires a type `α : Sort u` as a parameter. The only constructor of `Nonempty` is `Nonempty.intro`. This states that for any type `α`, if we have an element `a : α`, then we can construct a proof of `Nonempty α`. In other words, `Nonempty α` is true if there exists at least one instance of `α`. Note that `Nonempty α` asserts that `α` has at least one element but does not specify this element. Unlike `Inhabited α`, which requires an explicit `default` value, `Nonempty α` only requires an existence proof.

For example, we can prove that `Bool` is nonempty by constructing a `Nonempty Bool` instance using the `intro` constructor.

```
theorem TNEBool : Nonempty Bool := Nonempty.intro true
```

This proof shows that `Bool` is nonempty by providing `true` as a witness. Since `Nonempty α` only requires the existence of at least one element in `α`, choosing `true` suffices to establish the proof.

#### Inhabited implies Nonempty

We have a straightforward implication: if a type is inhabited, then it is also nonempty. The following code defines a function that converts an `Inhabited A` instance into a `Nonempty A` proof.

```
def InhabitedToNonempty {A : Type} : Inhabited A → Nonempty A := by
  intro h
  exact Nonempty.intro h.default
```

## 7.3 Choice

The reverse implication, `Nonempty A → Inhabited A`, does not always hold in constructive logic. A proof of `Nonempty A` only asserts the existence of an element without providing a specific one, whereas `Inhabited A` requires a concrete, predefined default value. In classical logic, we can recover `Inhabited A` from `Nonempty A` using the **axiom of choice** (`Classical.choice`), but this is **noncomputable**, meaning we cannot explicitly construct the default element. Consequently, while nonemptiness implies the mere existence of an element, inhabitation requires an explicit and fixed representative, making the two concepts distinct in constructive mathematics.

If we `#print Classical.choice`, Lean returns

```
axiom Classical.choice.{u} : { α : Sort u } → Nonempty α → α
```

This states that for any type `α : Sort u`, if `α` is nonempty, then we can obtain an actual element of `α`. Note that `Classical.choice` is an `axiom`. Many axioms, such as `Classical.choice`, are nonconstructive in nature, meaning they assert the existence of certain objects without providing explicit constructions. As a result, these axioms are often noncomputable and cannot be used in computational contexts.

#### Nonempty implies Inhabited in Classical Logic

Consider the following function.

```
noncomputable def NonemptyToInhabited {A : Type} : Nonempty A → Inhabited A := by
  intro h
  have a : A := Classical.choice h
  exact Inhabited.mk a
```

The function `NonemptyToInhabited` demonstrates how a proof of nonemptiness can be converted into a proof of inhabitation. Given that `Nonempty A` asserts the existence of at least one element in the type `A`, the function uses `Classical.choice` to extract `a : A`, a specific element of type `A` from the existence proof `h`. The extracted element is then used to construct a proof of inhabitation using `Inhabited.mk`, which asserts that `A` has a predefined default element. This function is marked as **noncomputable** because the choice of an element is nonconstructive: while the existence of an element is guaranteed, the method of selecting it cannot be explicitly computed.

### 7.3.1 Choose

The command `Classical.choose` is a function from classical logic that enables the selection of an element satisfying a given predicate. Specifically, given the existence of an element `x : X` such that a proposition `P x` holds (i.e., `∃ x : X, P x`), `Classical.choose` returns one such element `x` for which `P x` is true. `Classical.choose` is a direct consequence of `Classical.choice`; in fact, the two concepts are interderivable. Additionally, the command `Classical.choose_spec` guarantees that the element extracted indeed satisfies the predicate, providing a formal guarantee that the selected element meets the required condition.

In most situations, we can also use the alternative commands `Exists.choose` and `Exists.choose_spec` instead of `Classical.choose` and `Classical.choose_spec`. Here is an example of how it is used.

```
theorem TCarNonempty {A : Type} : Nonempty A ↔ ∃ (a : A), a = a := by
  apply Iff.intro
  -- Nonempty A → ∃ a, a = a
  intro h
  apply Exists.intro (Classical.choice h)
  exact rfl
  -- ∃( a, a = a) → Nonempty A
  intro h
  have a : A := Exists.choose h
  exact Nonempty.intro a
```

### 7.3.2 Exercises

```
-- Under Classical.Choice, if a function is injective and the domain is Nonempty then the function has a
     left inverse
theorem TInjtoHasLeftInv {A B : Type} {f : A → B} : injective f →  Nonempty A → hasleftinv f := by sorry

-- Under Classical.Choice, every surjective function has a right inverse
noncomputable def Inverse {A B : Type} (f : A → B) (h : surjective f) : B → A := by sorry

-- Under Classical.Choice, the inverse of a surjective function is a right inverse
theorem InvR {A B : Type} (f : A → B) (h : surjective f) : f ∘ (Inverse f h)  = id := by sorry

-- Under Classical.Choice, every surjective function has a right inverse
theorem TSurjtoHasRightInv {A B : Type} {f : A → B} : surjective f  → hasrightinv f := by sorry

-- Under Classical.Choice, the inverse of a bijective function is a left inverse
theorem InvL {A B : Type} (f : A → B) (h : bijective f) :  (Inverse f h.right) ∘ f = id := by sorry

-- Under Classical.Choice bijective and isomorphism are equivalent concepts
theorem TCarBijIso {A B : Type} {f : A → B} : bijective f ↔ isomorphism f :=  by sorry
```

# 8 Subtypes

This chapter explores the definition and usage of subtypes in Lean. We will introduce their basic properties, discuss common operations, and demonstrate their application in mathematical reasoning and program verification.

A **subtype** is a way to define a restricted entity of a given type by specifying a condition that the elements of this type must satisfy. A subtype of a type `A` is typically defined using a predicate `P : A → Prop`, which assigns a proposition to each element of `A`. The corresponding subtype, denoted as `Subtype P` or `{a : A // P a}`, consists of all elements `a : A` that satisfy `P`.

```
-- We define variables A : Type and P : A → Prop, a predicate on A
variable (A : Type)
variable (P : A → Prop)
-- With this information we can obtain `Subtype P`
#check Subtype P
-- An alternative notation is
#check { a : A // P a }
```

Subtypes play a crucial role in formal verification, as they allow us to encode mathematical objects with additional properties. For example, we can define the subtype of even natural numbers, the positive real numbers, or the set of invertible matrices. By doing so, we ensure that any element of the subtype inherently satisfies the given condition, reducing the need for repetitive proof obligations.

If we `#print Subtype`, Lean returns

```
structure Subtype.{u} : { α : Sort u } → α( → Prop) → Sort (max 1 u)
number of parameters: 2
constructor:
Subtype.mk : { α : Sort u } → {p : α → Prop} → (val : α) → p val → Subtype p
fields:
val : α
property : p self.val
```

The `Subtype` structure in Lean is parameterized by a type `α` and a predicate `p : α → Prop`, which defines adscription to the subtype. It includes a constructor, `Subtype.mk`, that takes a value `val : α` along with a proof of `p val`, ensuring that `val` satisfies the predicate. An instance of `Subtype p` has two fields: `val`, which holds the underlying value, and `property`, which provides the proof that `val` satisfies `p`.

### 8.0.1 Examples of subtypes

#### The False subtype

Given a type `A`, we can consider the `False` subtype on `A`.

```
def SFalse {A : Type} := { a : A // PFalse a}
```

#### The True subtype

Given a type `A`, we can consider the `True` subtype on `A`.

```
def STrue {A : Type} := { a : A // PTrue a}
```

#### The image of a function

We introduce the *image* of a function. Given two types `A` and `B` and a function `f : A → B`, we define the image of `f`, denoted as `Im f`, as the subtype of `B` consisting of all elements that are mapped from some `a : A` under `f`.

```
def Im {A B : Type} (f : A → B) : Type := { b : B // ∃ (a : A), f a = b}
```

### 8.0.2 Elements of a subtype

To create an element of `Subtype P`, we use the `Subtype.mk` function, which maps an element `a : A` and a proof `h : P a` to an element of type `Subtype P`.

```
variable (a : A)
variable (h : P a)
#check Subtype.mk a h
```

Two elements of type `Subtype P` are equal if and only if their corresponding values in `A` are also equal. For this we have the the theorems `Subtype.eq` and `Subtype.eq_iff`.

```
#check Subtype.eq     -- `a1.val = a2.val → a1 = a2`
#check Subtype.eq_iff -- `a1.val = a2.val ↔ a1 = a2`
```

### 8.0.3 The inclusion function

The *inclusion* function is a function of a subtype into its underlying type: it simply extracts the value of a `Subtype P` element, discarding its proof.

```
def inc {A : Type} {P : A → Prop} : Subtype P → A := by
  intro a
  exact a.val
```

Thanks to `Subtype.eq` we can prove that the inclusion function is always injective.

```
theorem Tincinj {A : Type} {P : A → Prop} : injective (@inc A P) := by
  intro a1 a2 h1
  exact Subtype.eq h1
```

## 8.1 Functions and Subtypes

### 8.1.1 Restriction

We can formalize the notion of restricting functions to subtypes. Any function `f : A → B` can be *restricted* to a subtype by applying `f` only to the underlying values of the subtype elements. Restriction provides a way to transform elements of type `A → B` into elements of type `Subtype P → B`.

```
def rest {A B : Type} {P : A → Prop} (f : A → B) : Subtype P → B := by
  intro a
  exact f a.val
```

### 8.1.2 Correstriction

Given a function `f : A → B` and a predicate `Q` on `B`, we can *correstrict* `f` to `Subtype P`, provided that every `b : Im f` satisfies `Q`. If the above condition holds, correstriction provides a way to transform elements of type `A → B` into elements of type `A → Subtype Q`.

```
def correst {A B : Type} {Q : B → Prop} (f : A → B) (h : ∀ (b : Im f), Q b.val)
: A → Subtype Q := by
  intro a
  have ha : ∃ (a1 : A), f a1 = f a := by
    apply Exists.intro a
    exact rfl
  apply Subtype.mk (f a) (h ⟨f a, ⟩ha)
```

### 8.1.3 Birrestriction

Given a function `f : A → B`, a predicate `P` on `A` and a predicate `Q` on `B`, we can *birrestrict* `f` to the respective subtypes, provided that `f a` satisfies `Q` for every `a : Subtype P`. If the above condition holds, birrestriction provides a way to transform elements of type `A → B` into elements of type `Subtype P → Subtype Q`.

```
def birrest {A B : Type} {P : A → Prop} {Q : B → Prop} (f : A → B) (h : ∀ (a : A), P a → Q (f a)) :
    Subtype P → Subtype Q := by
  apply correst (rest f)
  intro ⟨b, ⟨a, ⟩⟩ha
  simp
  specialize h a.val
  have hb : f a.val = b := ha
  rw [hb] at h
  exact h a.property
```

In particular, given two predicates `P1` and `P2` on a type `A`, the following function establishes a transformation from the subtype corresponding to `P1` to the subtype corresponding to `P2`, provided that `P1` implies `P2`. This is achieved by birrestricting the identity function.

```
def SubtoSub {A : Type} {P1 P2 : A → Prop} (h : ∀ (a : A), P1 a → P2 a) : Subtype P1 → Subtype P2 :=
    birrest id h
```

## 8.2 Equalizers

In this section, we introduce the concept of the **equalizer** of two functions, a construction that identifies the subtype of a domain where the two functions agree. Beyond its definition as a subtype, the equalizer is also characterized by a **universal property**: it serves as the most general type equipped with a map into `A` on which the functions agree.

Given two functions `f, g : A → B`, the **equalizer** of `f` and `g` is the subtype of `A` consisting on all elements `a : A` such that `f a = g a`.

```
def Eq {A B : Type} (f g : A → B) : Type := { a : A // f a = g a}
```

It commes equipped with the inclusion function, from the equalizer to `A`.

```
def incEq {A B : Type} (f g : A → B) : Eq f g → A := @inc A (fun a => f a = g a)
```

This inclusion satisfies that `f ∘ (incEq f g) = g ∘ (incEq f g)`.

```
theorem TEqInc {A B : Type} (f g : A → B) : f ∘ (incEq f g) = g ∘ (incEq f g) := by
  apply funext
  intro a
  calc
    (f ∘ (incEq f g)) a = f a.val               := rfl
    _                   = g a.val               := a.property
    _                   = (g ∘ (incEq f g)) a := rfl
```

### 8.2.1 Universal property of the equalizer

The **universal property** of the equalizer characterizes it not merely as a subtype, but as a *universal solution* to the problem of mediating between `f` and `g`. The universal property states that the pair `(Eq f g, incEq f g)` is initial among all pairs `(C, h)`, where `C` is a type and `h : C → A` is a function satisfying `f ∘ h = g ∘ h`.

That is, if `C` is a type and `h : C → A` is a function satisfying `f ∘ h = g ∘ h`, then there exists a unique function `u : C → (Eq f g)` such that `(incEq f g) ∘ u = h`.

```
-- If there is another function h : C → A satisfying f ∘ h = g ∘ h, then there exists a function u : C
    → Eq f g
def u {A B C : Type} {f g : A → B} {h : C → A} (h1 : f ∘ h = g ∘ h) : C → Eq f g := by
  intro c
  exact Subtype.mk (h c) (congrFun h1 c)

-- The function u satisfies that incEq f g ∘ u = h
theorem TEqIncEq {A B C : Type} {f g : A → B} {h : C → A} (h1 : f ∘ h = g ∘ h) :
(incEq f g) ∘ (u h1) = h := by
  apply funext
  intro c
  exact rfl

-- The function u is unique in the sense that, if there is another function v : C → Eq f g satisfying
    incEq f g ∘ v = h, then v = u.
```

```
theorem TEqUni {A B C : Type} {f g : A → B} {h : C → A} (h1 : f ∘ h = g ∘ h)
(v : C → Eq f g) (h2 : (incEq f g) ∘ v = h) : v = u h1 := by
  apply funext
  intro c
  apply Subtype.eq
  calc
    (v c).val = ((incEq f g) ∘ v) c := rfl
    _         = h c                 := congrFun h2 c
    _         = (u h1 c).val        := rfl
```

In other words, any function into A that "equalizes" f and g factors uniquely through the equalizer. This property ensures that the equalizer is the most general and canonical way to capture the elements where two functions agree.

## 8.3 Exercises

### 8.3.1 Subtypes

```
-- If two subtypes are equivalent, the corresponding subtypes are equal.
theorem TEqSubtype {A : Type} {P1 P2 : A → Prop} (h : ∀ (a : A), P1 a ↔ P2 a) : Subtype P1 = Subtype P2
    := by sorry
```

### 8.3.2 Restriction

```
-- Im (inc) = Subtype
theorem TUPSub {A : Type} {P : A → Prop} : Im (@inc A P) = Subtype P := by sorry

-- rest f = f ∘ inc
theorem TRest {A B : Type} {f : A → B} {P : A → Prop}: (@rest A B P f) = f ∘ (@inc A P) := by sorry
```

### 8.3.3 Correstriction

Theorems `TUPCorrest` and `TUPCorrestUn` establish the *universal property* of the correstriction of a function. The first result, `TUPCorrest`, states that for any function `f : A → B` that respects a predicate `Q` on `B` (i.e., `Q b` holds for all `b : Im f`)), the function `f` can be expressed as the composition of the inclusion function `inc` and its correstriction `correst f h`, that is `f = inc ∘ (correst f h)`. The second result, `TUPCorrestUn`, establishes the *uniqueness* of the correstriction. If there exists another function `g : A → Subtype P` such that `f = inc ∘ g`, then `g` must be exactly `correst f h`.

```
-- f = inc ∘ correst
theorem TUPCorrest {A B : Type} {Q : B → Prop} {f : A → B}
(h : ∀ (b : Im f), Q b.val) : f = (@inc B Q) ∘ (correst f h) := by sorry

-- Unicity
theorem TUPCorrestUn {A B : Type} {Q : B → Prop} {f : A → B}
(h : ∀ (b : Im f), Q b.val) (g : A → Subtype Q) (h1 : f = (@inc B Q) ∘ g) : (correst f h) = g := by
    sorry
```

### 8.3.4 Equalizers

```
-- The function incEq is a monomorphism
theorem TincEqMono {A B : Type} {f g : A → B} : monomorphism (incEq f g) := by sorry

-- An epic incEq is an isomorphism
theorem TincEqEpi {A B : Type} {f g : A → B} : epimorphism (incEq f g) → isomorphism (incEq f g) := by
    sorry
```

# 9 Relations

A **relation** on a type `A` is a predicate that takes two elements of `A` and returns a proposition, indicating whether the elements are related. In Lean, a relation is represented as a function of type `A → A → Prop`. Given a relation `R : A → A → Prop` and elements `a1, a2 : A`, the expression `R a1 a2` asserts that `a1` and `a2` are related under `R`. Relations play a fundamental role in mathematics, capturing concepts such as order, equivalence, or divisibility, among others. In this chapter, we explore key properties that relations can satisfy, laying the groundwork for their formal use in Lean.

Since relations are predicates on two input variables, two relations are equal if and only if they relate the same elements. This can be shown by applying `funext` twice. Furthermore, this is equivalent to the predicates being logically equivalent on the same elements, which follows from `propext`.

```
theorem TEqRel {A : Type} {R S : A → A → Prop} : R = S ↔ ∀ (a1 a2 : A),
R a1 a2 ↔ S a1 a2 := by
  apply Iff.intro
  -- R = S → ∀ (a1 a2 : A), R a1 a2 ↔ S a1 a2
  intro h a1 a2
  apply Iff.intro
  -- R a1 a2 → S a1 a2
  intro hR
  rw [h.symm]
  exact hR
  -- S a1 a2 → R a1 a2
  intro hS
  rw [h]
  exact hS
  -- ∀( (a1 a2 : A), R a1 a2 ↔ S a1 a2) → R = S
  intro h
  apply funext
  intro a1
  apply funext
  intro a2
  apply propext
  exact h a1 a2
```

### 9.0.1 Examples of relations

Given a type `A`, we can define various relations on it. For instance, the `emptyRelation` relation relates no elements at all. At the other extreme, the `total` relation relates every element to every other element. Another important example is the diagonal relation, `diag`, where each element is related only to itself.

```
-- The empty relation on A
def empty {A : Type} : A → A → Prop := fun x y => False

-- The total relation on A
def total {A : Type} : A → A → Prop := fun x y => True

-- The diag (diagonal) relation on A
def diag {A : Type} : A → A → Prop := fun x y => (x = y)
```

## 9.1 Types of relations

In this section, we explore various properties that a relation can satisfy.

#### Reflexive

A relation `R : A → A → Prop` is *reflexive* if, for every `a : A`, the proposition `R a a` holds, meaning that `a` is `R`-related to itself.

```
def Reflexive {A : Type} (R : A →  A → Prop) : Prop := ∀{a : A}, R a a
```

#### Symmetric

A relation `R : A → A → Prop` is *symmetric* if, for all `a1, a2 : A`, whenever `R a1 a2` holds, `R a2 a1` must also hold.

```
def Symmetric {A : Type} (R : A → A → Prop) : Prop := ∀{a1 a2 : A}, R a1 a2 → R a2 a1
```

#### Antisymmetric

A relation `R : A → A → Prop` is *antisymmetric* if, for all `a1, a2 : A`, whenever `R a1 a2` and `R a2 a1` hold, it follows that `a1 = a2`.

```
def Antisymmetric {A : Type} (R : A → A → Prop) : Prop := ∀{a1 a2 : A}, R a1 a2 → R a2 a1 → (a1 = a2)
```

#### Transitive

A relation `R : A → A → Prop` is *transitive* if, for all `a1, a2 a3 : A`, whenever `R a1 a2` and `R a2 a3` hold, it follows that `R a1 a3`.

```
def Transitive {A : Type} (R : A → A → Prop) : Prop := ∀{a1 a2 a3 : A}, R a1 a2 → R a2 a3 → R a1 a3
```

#### Serial

A relation `R : A → A → Prop` is *serial* if, for every `a1 : A`, there exists an element `a2 : A` such that `R a1 a2`

```
def Serial {A : Type} (R : A → A → Prop) : Prop := ∀{a1 : A}, ∃(a2 : A), R a1 a2
```

#### Euclidean

A relation `R : A → A → Prop` is *Euclidean* if, for every `a1, a2 ,a3 : A`, whenever `R a1 a2` and `R a1 a3` hold, it follows that `R a2 a3`.

```
def Euclidean {A : Type} (R : A → A → Prop) : Prop := ∀{a1 a2 a3 : A}, R a1 a2 → R a1 a3 → R a2 a3
```

#### Partially functional

A relation `R : A → A → Prop` is *partially functional* if, for every `a1, a2, a3 : A`, whenever `R a1 a2` and `R a1 a3` hold, it follows that `a2 = a3`.

```
def PartiallyFunctional {A : Type} (R : A → A → Prop) : Prop := ∀{a1 a2 a3 : A}, R a1 a2 → R a1 a3 → a2
    = a3
```

#### Functional

A relation `R : A → A → Prop` is *functional* if, for every `a1 : A`, there exists a unique `a2 : A` such that `R a1 a2`.

```
def Functional {A : Type} (R : A → A → Prop) : Prop := ∀(a1 : A), ∃(a2 : A), ∀(a3 : A), R a1 a3 ↔ a2 =
    a3
```

#### Weakly dense

A relation `R : A → A → Prop` is *weakly dense* if, for every `a1 a2 : A`, if `R a1 a2` holds then there exists `a3 : A` such that `R a1 a3` and `R a3 a2`.

```
def WeaklyDense {A : Type} (R : A → A → Prop) : Prop := ∀{a1 a2 : A}, (R a1 a2 → ∃(a3 : A), (R a1 a3) ∧
    (R a3 a2))
```

### Weakly connected

A relation `R : A → A → Prop` is *weakly connected* if, for all `a1, a2, a3 : A`, whenever `R a1 a2` and `R a1 a3` hold, at least one of the following three conditions must be satisfied: `R a2 a3`, `a2 = a3`, or `R a3 a2`.

```
def WeaklyConnected {A : Type} (R : A → A → Prop) : Prop := ∀{a1 a2 a3 : A}, R a1 a2 → R a1 a3 → ((R a2
    a3) ∨ (a2 = a3) ∨ (R a3 a2))
```

### Weakly directed

A relation `R : A → A → Prop` is *weakly directed* if, for all `a1, a2, a3 : A`, whenever `R a1 a2` and `R a1 a3` then there exists `a4 : A` satisfying that `R a2 a4` and `R a3 a4`.

```
def WeaklyDirected {A : Type} (R : A → A → Prop) : Prop := ∀{a1 a2 a3 : A}, R a1 a2 → R a1 a3 → ∃(a4 : A
    ), ((R a2 a4) ∧ (R a3 a4))
```

## 9.1.1 An example: The diagonal

The diagonal relation is a relation that satisfies all the properties above.

```
-- The diagonal is reflexive
theorem TDiagRefl {A : Type} : Reflexive (@diag A) := by
  intro a
  exact rfl

-- The diagonal is symmetric
theorem TDiagSymm {A : Type} : Symmetric (@diag A) := by
  intro a1 a2 h
  exact h.symm

-- The diagonal is antisymmetric
theorem TDiagASymm {A : Type} : Antisymmetric (@diag A) := by
  intro a1 a2 h1 h2
  exact h1

-- The diagonal is transitive
theorem TDiagTrans {A : Type} : Transitive (@diag A) := by
  intro a1 a2 a3 h1 h2
  exact h1.trans h2

-- The diagonal is serial
theorem TDiagSer {A : Type} : Serial (@diag A) := by
  intro a
  apply Exists.intro a
  exact rfl

-- The diagonal is Euclidean
theorem TDiagEucl {A : Type} : Euclidean (@diag A) := by
  intro a1 a2 a3 h1 h2
  exact h1.symm.trans h2

-- The diagonal is partially functional
theorem TDiagPFunc {A : Type} : PartiallyFunctional (@diag A) := by
  intro a1 a2 a3 h1 h2
  exact h1.symm.trans h2

-- The diagonal is functional
theorem TDiagFunc {A : Type} : Functional (@diag A) := by
  intro a
  apply Exists.intro a
  intro z
  apply Iff.intro
  -- diag a z → a = z
  intro h
  exact h
  -- a = z → diag a z
  intro h
  exact h

```

```
-- The diagonal is weakly dense
theorem TDiagWDense {A : Type} : WeaklyDense (@diag A) := by
  intro a1 a2 h1
  apply Exists.intro a1
  apply And.intro
  -- Left
  exact rfl
  -- Right
  exact h1

-- The diagonal is weakly connected
theorem TDiagWConn {A : Type} : WeaklyConnected (@diag A) := by
  intro a1 a2 a3 h1 h2
  exact Or.inl (h1.symm.trans h2)

-- The diagonal is weakly directed
theorem TDiagWDir {A : Type} : WeaklyDirected (@diag A) := by
  intro a1 a2 a3 h1 h2
  apply Exists.intro a1
  apply And.intro
  -- Left
  exact h1.symm
  -- Right
  exact h2.symm
```

### 9.1.2 Exercises

The following exercises were extracted from Zach, R. (2019) *Boxes and Diamonds: An Open Introduction to Modal Logic.*

```
variable (A : Type)
variable (R : A → A → Prop)

-- Reflexive implies serial
theorem TRefltoSerial : Reflexive R → Serial R := by sorry

-- For a symmetric relation, transitive and Euclidean are equivalent
theorem TSymmTransIffSer (hS : Symmetric R) : Transitive R ↔ Euclidean R := by sorry

-- If a relation is symmetric then it is weakly directed
theorem TSymmtoWDir : Symmetric R → WeaklyDirected R := by sorry

-- If a relation is Euclidean and antisymmetric, then it is weakly directed
theorem TEuclASymmtoWDir : Euclidean R → Antisymmetric R → WeaklyDirected R := by sorry

-- If a relation is Euclidean, then it is weakly connected
theorem TEucltoWConn : Euclidean R → WeaklyConnected R := by sorry

-- If a relation is functional, then it is serial
theorem TFunctoSer : Functional R → Serial R := by sorry

-- If a relation is symmetric and transitive, then it is Euclidean
theorem TSymmTranstoEucl : Symmetric R → Transitive R → Euclidean R := by sorry

-- If a relation is reflexive and Euclidean, then it is symmetric
theorem TReflEucltoSymm : Reflexive R → Euclidean R → Symmetric R := by sorry

-- If a relation is symmetric and Euclidean, then it is transitive
theorem TSymmEucltoTrans : Symmetric R → Euclidean R → Transitive R := by sorry

-- If a relation is serial, symmetric and transitive, then it is reflexive
theorem TSerSymmTranstoRefl : Serial R → Symmetric R → Transitive R → Reflexive R := by sorry
```

## 9.2 Operations on relations

#### Composition

The composition of two binary relations captures the idea of chaining relations through an intermediate element. Given binary relations `R` and `S` on a type `A`, its **composition**, denoted as `R ∘ S`, is a new relation on A that holds between two elements a1 and a3 if there exists an intermediate element a2 such that R relates a1 to a2 and S relates a2 to a3.

```
def composition {A : Type} (R S : A → A → Prop) : A → A → Prop := by
  intro a1 a3
  exact ∃ (a2 : A), (R a1 a2) ∧ (S a2 a3)

-- Notation for the composition (\circ)
notation : 65 lhs:65 " ∘ " rhs:66 => composition lhs rhs
#check R ∘ S
```

#### Inverse

Given a relation `R : A → A → Prop` on a type A, its **inverse**, denoted R^, swaps the order of its arguments. The definition of `inverse` takes a relation R and returns a new relation R^ where 'R^ a1 a2 holds if and only if R a2 a1 holds in the original relation.

```
def inverse {A : Type} (R : A → A → Prop) : A → A → Prop := by
  intro a1 a2
  exact R a2 a1

-- Notation for the inverse (^)
notation : 65 lhs:65 "^" => inverse lhs
#check R^
```

#### Meet

Given binary relations R and S on a type A, its **meet**, denoted as R ∧ S, is a new relation on A that holds between two elements a1 and a2 if both R a1 a2 and S a1 a2 hold.

```
def meet {A : Type} (R S : A → A → Prop) : A → A → Prop := by
  intro a1 a2
  exact (R a1 a2) ∧ (S a1 a2)

-- Notation for the meet  (`\and`)
notation : 65 lhs:65 " ∧ " rhs:66 => meet lhs rhs
#check R ∧ S
```

#### Join

Given binary relations R and S on a type A, its **join**, denoted as R ∨ S, is a new relation on A that holds between two elements a1 and a2 if either R a1 a2 or S a1 a2 hold.

```
def join {A : Type} (R S : A → A → Prop) : A → A → Prop := by
  intro a1 a2
  exact (R a1 a2) ∨ (S a1 a2)

-- Notation for the join (`\or`)
notation : 65 lhs:65 " ∨ " rhs:66 => join lhs rhs
#check R ∨ S
```

### 9.2.1 Exercises

The following propositions are common identities involving binary relations.

```
variable {A : Type}
variable (R S T : A → A → Prop)

-- Associativity of composition
theorem TAssComp : R ∘ (S ∘ T) = (R ∘ S) ∘ T := by sorry

-- The diagonal is a left neutral element for the composition
theorem TDiagL : R ∘ (@diag A) = R := by sorry

-- The diagonal is a right neutral element for the composition
theorem TDiagR : (@diag A) ∘ R = R := by sorry

```

```
-- The inverse relation of the inverse relation is the original relation
theorem TInvInv : (R^)^ = R := by sorry

-- The inverse of the composition
theorem TInvComp : (R ∘ S)^ = (S^) ∘ (R^) := by sorry

-- The inverse of the meet
theorem TInvMeet : (R ∧ S)^ = (S^) ∧ (R^) := by sorry

-- The inverse of the join
theorem TInvJoin : (R ∨ S)^ = (S^) ∨ (R^) := by sorry

-- Distributivity of composition on the left over join
theorem TDisL : R ∘ (S ∨ T) = (R ∘ S) ∨ (R ∘ T) := by sorry

-- Distributivity of composition on the right over join
theorem TDisR : (R ∨ S) ∘ T = (R ∘ T) ∨ (S ∘ T) := by sorry

-- Empty is a left zero for the composition
theorem TEmptLZ : (@empty A) ∘ R = (@empty A) := by sorry

-- Empty is a right zero for the composition
theorem TEmptRZ : R ∘ (@empty A) = (@empty A) := by sorry
```

# 10 Quotients

In mathematics, we often want to consider objects *modulo* some form of identification—treating different representatives as essentially the same. This practice leads us naturally to the notion of *equivalence relations*, which formally define when two elements should be considered indistinguishable for our purposes.

Once we have an equivalence relation, we can group elements into *equivalence classes*—collections of elements that are all equivalent to each other. The process of forming a new structure out of these equivalence classes is called taking a *quotient*. Quotients allow us to construct new types that "forget" unnecessary distinctions while preserving the structure we care about.

This chapter introduces the foundations of working with quotients in a formal setting. We begin by reviewing equivalence relations and exploring concrete examples. We then move to the notion of *setoids*, which package a type together with an equivalence relation—an essential concept in Lean.

Next, we delve into the construction of quotients themselves. We examine how to reason about their elements, understand the *projection* function from a type to its quotient, and work through illustrative examples.

We then explore how functions behave in the presence of quotient types. Specifically, we study how functions defined on a type can be *astricted*, *coastricted*, and *biastricted*. These ideas are key when reasoning about quotient structures in a type-theoretic setting.

Finally, we explore the notion of the *coequalizer* of two functions.

The chapter concludes with exercises to reinforce our understanding and help us apply these concepts.

## 10.1 Equivalence relations

A relation `R : A → A → Prop` is an *equivalence relation* if it is `Reflexive`, `Symmetric` and `Transitive`. This is already implemented as `Equivalence`.

If we `#print Equivalence`, Lean returns:

```
structure Equivalence.{u} : { α : Sort u } → ( α → α → Prop ) → Prop
number of parameters: 2
constructor:
Equivalence.mk : ∀ { α : Sort u } { r : α → α → Prop },
  ∀( ( x : α ), r x x) → ∀( { x y : α }, r x y → r y x) → ∀( { x y z : α }, r x y → r y z → r x z) →
    Equivalence r
fields:
refl : ∀ ( x : α ), r x x
symm : ∀ { x y : α }, r x y → r y x
trans : ∀ { x y z : α }, r x y → r y z → r x z
```

This structure takes two parameters: a type `α : Sort u` and a binary relation `r : α → α → Prop`. It encapsulates the three fundamental properties that define an equivalence relation: reflexivity, symmetry, and transitivity. The `Equivalence.mk` constructor allows us to create an instance of `Equivalence r` by providing proofs for these three properties. Specifically, the `refl` field ensures that every element is related to itself `∀ (x : α), r x x`, the `symm` field guarantees that the relation is symmetric `∀ {x y : α}, r x y → r y x`, and the `trans` field enforces transitivity `∀ {x y z : α}, r x y → r y z → r x z`.

### 10.1.1 Examples of equivalence relations

#### The diagonal relation

Given a type `A`, the diagonal relation on `A` is an equivalence relation on `A`.

```
theorem TDiagEqv {A : Type} : Equivalence (@diag A) := {
  refl  := by
    intro a
    exact TDiagRefl
  symm  := by
```

```
    intro a1 a2
    exact TDiagSymm
  trans := by
    intro a1 a2 a3
    exact TDiagTrans
```

### The total relation

Given a type A, the total relation on A is an equivalence relation on A.

```
theorem TTotalEqv {A : Type} : Equivalence (@total A) := {
  refl  := by
    intro _
    exact trivial
  symm  := by
    intro a1 a2 _
    exact trivial
  trans := by
    intro a1 a2 a3 _ _
    exact trivial
```

### The kernel of a function

We introduce the *kernel* of a function and demonstrate that it defines an equivalence relation. Given two types A and B and a function `f : A → B`, we define the kernel of f, denoted as `Ker f`, as a binary relation on A where two elements `a1, a2 : A` are related if and only if they have the same image under f, i.e., `f a1 = f a2`. We then establish that `Ker f` satisfies the properties of an equivalence relation.

```
def Ker {A B : Type} (f : A → B) : A → A → Prop := by
  intro a1 a2
  exact f a1 = f a2

-- The kernel of a function is an equivalence relation
theorem TKerEqv {A B : Type} {f : A → B} : Equivalence (Ker f) := {
  refl  := by
    intro a
    exact rfl
  symm  := by
    intro a1 a2 h1
    exact h1.symm
  trans := by
    intro a1 a2 a3 h1 h2
    exact h1.trans h2
}
```

## 10.2 Equivalence relation generated by a relation

The following Lean code defines the equivalence closure of a relation as an inductive type and proves that it forms an equivalence relation. Specifically, the inductive type `Eqvgen` constructs the smallest equivalence relation generated by a given relation `R : A → A → Prop` on a type A. The `Eqvgen` type is defined with four constructors: `base`, which includes the original relation R; and `refl`, `symm`, and `trans`, ensuring that `Eqvgen R` satisfies the properties for reflexivity, symmetry, and transitivity, respectively.

```
inductive Eqvgen {A : Type} (R : A → A → Prop) : A → A → Prop where
  | base : ∀ {a1 a2 : A}, (R a1 a2 → Eqvgen R a1 a2)
  | refl : ∀ (a : A), Eqvgen R a a
  | symm : ∀ {a1 a2 : A}, Eqvgen R a1 a2 → Eqvgen R a2 a1
  | trans : ∀ {a1 a2 a3 : A}, (Eqvgen R a1 a2) → (Eqvgen R a2 a3) → (Eqvgen R a1 a3)

-- The equivalence generated by a relation is an equivalence relation
theorem TEqvgen {A : Type} (R : A → A → Prop) : Equivalence (Eqvgen R) := {
  refl  := Eqvgen.refl
  symm  := Eqvgen.symm
  trans := Eqvgen.trans
}
```

## 10.3 Setoids

A **Setoid** is a type class that encapsulates an equivalence relation on a given type.

If we `#print Setoid`, Lean returns

```
class Setoid.{u} : Sort u → Sort (max 1 u)
number of parameters: 1
constructor:
Setoid.mk : { α : Sort u } → (r : α → α → Prop) → Equivalence r → Setoid α
fields:
r : α → α → Prop
iseqv : Equivalence Setoid.r
```

Given a type `α`, a `Setoid α` consists of a binary relation `r : α → α → Prop` and a proof that `r` is an equivalence relation. The constructor `Setoid.mk` allows defining such structures by providing both the relation and its proof of equivalence. An element of type A `Setoid α` has two fields, `r : α → α → Prop` the equivalence relation on `α` and `iseqv : Equivalence Setoid.r`, a proof that `r` is an equivalence relation.

### 10.3.1 Examples of setoids

Since both the diagonal relation and the total relation are equivalence relations, we can define their corresponding setoids. Additionally, we can construct the setoid associated with the kernel of a function.

#### The diagonal setoid

```
instance DiagSetoid {A : Type} : Setoid A := {
  r     := @diag A
  iseqv := TDiagEqv
}
```

#### The total setoid

```
instance TotalSetoid {A : Type} : Setoid A := {
  r     := @total A
  iseqv := TTotalEqv
}
```

#### The kernel setoid

```
instance KerSetoid {A B : Type} (f : A → B) : Setoid A := {
  r     := Ker f
  iseqv := TKerEqv
}
```

## 10.4 Quotients

The significance of `Setoid` lies in its role in quotienting, where elements of `α` can be grouped into equivalence classes based on `r`, leading to the `Quotient` construction. The **`Quotient`** type provides a structured way to form quotient types based on equivalence relations.

If we `#print Quotient`, Lean returns

```
def Quotient.{u} : α{ : Sort u} → Setoid α → Sort u :=
fun α{} s => Quot Setoid.r
```

The `Quotient` type constructs the quotient of a given *setoid*, encapsulating an equivalence relation on a type. It takes an implicit type `α` along with an instance of `Setoid α`, which defines an equivalence relation `r` on `α`. By applying `Quot Setoid.r`, it forms a new type in which elements of `α` are identified according to `r`. The resulting type remains in the same universe level `Sort u`, preserving the type hierarchy of `α`. Unlike `Quot`, which can be defined for arbitrary binary relations, `Quotient` is specifically tailored for equivalence relations, ensuring a structured and well-behaved quotienting mechanism in Lean.

Note that for a type `A` and an element `S` of type `Setoid Am`, the term `Quotient S` is a type.

```
variable (A : Type)
variable (S : Setoid A)
#check Quotient S
```

### 10.4.1 Examples of quotients

With the setoids defined above, we can construct the quotient of a type A by its diagonal relation, the quotient of A by the total relation and the quotient of A by the kernel of a function.

#### The diagonal quotient

```
def QDiag {A : Type} := Quotient (@DiagSetoid A)
```

#### The total quotient

```
def QTotal {A : Type} := Quotient (@TotalSetoid A)
```

#### The kernel quotient

```
def QKer {A B : Type} (f : A → B) := Quotient (KerSetoid f)
```

### 10.4.2 Elements of a quotient

To create an element of `Quotient S`, we use the `Quotient.mk S a` function, which maps an element `a : A` to its **equivalence class**, that is, the elements of type A that are related with a.

```
variable (a : A)
#check Quotient.mk S a
```

Two equivalence classes are equal if and only if their representatives are related by the underlying equivalence relation. First, we use `Quotient.exact`, which states that if two classes are equal, then their representatives must be related. Conversely, we apply `Quotient.sound`, which ensures that if two elements are related, then their classes are equal. This result confirms that quotient types faithfully represent equivalence classes, ensuring that equality in the quotient type corresponds precisely to the equivalence relation in the original type.

```
theorem TEqQuotient {A : Type} {S : Setoid A} {a1 a2 : A} :
Quotient.mk S a1 = Quotient.mk S a2 ↔ S.r a1 a2 := by
  apply Iff.intro
  -- Quotient.mk S a1 = Quotient.mk S a2 → Setoid.r a1 a2
  apply Quotient.exact
  -- Setoid.r a1 a2 → Quotient.mk S a1 = Quotient.mk S a2
  apply Quotient.sound
```

### 10.4.3 The projection function

The *projection* function is a function of a type into its quotient: it simply constructs the class of a given element.

```
def pr {A : Type} (S : Setoid A) : A → Quotient S := by
  intro a
  exact Quotient.mk S a
```

An essential property of quotient types is that every element `q` of type `Quotient S` has a **representative** in A, meaning there exists some `a : A` such that `Quotient.mk S a` produces the element q. This property is formalized by `Quotient.exists_rep`. This proposition implies, in particular, that the projection function is always surjective.

```
-- The projection function is surjective
theorem TprSurj {A : Type} (S : Setoid A) : surjective (pr S) := by
  intro q
  apply Quotient.exists_rep
```

## 10.5 Functions and Quotient types

### 10.5.1 Astriction

We can formalize the notion of astricting functions to quotients. Any function `f : A → B` can be *astricted* to a quotient by a setoid `S` on `B` by considering classes on the images under `f`. Astriction provides a way to transform elements of type `A → B` into elements of type `A → Quotient S`.

```
def ast {A B : Type} {S : Setoid B} (f : A → B) : A → Quotient S := by
  intro a
  exact Quotient.mk S (f a)
```

### 10.5.2 Coastriction

Given a function `f : A → B` and a setoid `R` on `A`, we can *coastrict* `f` to `Quotient R`, provided that every pair related according to the underlying relation of `R` is also related according to `Ker f`. For this we will apply the keyword `Quotient.lift`. If the above condition holds, coastriction provides a way to transform elements of type `A → B` into elements of type `Quotient R → B`.

```
def coast {A B : Type} {R : Setoid A} (f : A → B) (h : ∀ (a1 a2 : A),
R.r a1 a2 → (Ker f) a1 a2) : Quotient R → B := by
  apply Quotient.lift f
  intro a1 a2 h1
  exact (h a1 a2) h1
```

### 10.5.3 Biastriction

Given a function `f : A → B`, a setoid `R` on `A` and a setoid `S` on `B`, we can *biastrict* `f` to the respective quotient types, provided that if a pair `a1 a2 : A` is related according to the underlying relation of `R` then the pair `(f a1), (f a2) : B` is related according to the underlying relation of `S`. If the above condition holds, biastriction provides a way to transform elements of type `A → B` into elements of type `Quotient R → Quotient S`.

```
def biast {A B : Type} {R : Setoid A} {S : Setoid B} (f : A → B)
(h : ∀ (a1 a2 : A), R.r a1 a2 → S.r (f a1) (f a2)) : Quotient R → Quotient S := by
  apply coast (ast f)
  intro a1 a2 hR
  specialize h a1 a2
  apply Quotient.sound
  exact h hR
```

In particular, given two setoids `R1` and `R2` on a type `A`, the following function establishes a transformation from the type `Quotient R1` to the type `Quotient R2`, provided that the underlying relation of `R1` implies the underlying relation of `R2`. This is achieved by biastricting the identity function.

```
def QuottoQuot {A : Type} {R1 R2 : Setoid A} (h : ∀ (a1 a2 : A), R1.r a1 a2 → R2.r a1 a2) : Quotient R1
    → Quotient R2 := biast id h
```

## 10.6 Coequalizer

In this section, we introduce the concept of the **coequalizer** of two functions, a construction that identifies elements in the codomain where the two functions produce equivalent outputs. Beyond its definition as a quotient type, the coequalizer is characterized by a **universal property**: it serves as the most general type equipped with a map from `B` that makes the two functions agree.

Given two functions `f, g : A → B`, the **coequalizer** of `f` and `g` is the quotient of `B` by the equivalence relation generated by identifying `f a` with `g a` for all `a : A`.

```
-- We define the relation on B relating elements of the form f a and g a for some a : A
def CoeqRel {A B : Type} (f g : A → B) : B → B → Prop := by
  intro b1 b2
  exact ∃ (a : A), (f a = b1) ∧ (g a = b2)

-- We next consider the equivalence relation generated by the previous relation
def CoeqEqv {A B : Type} (f g : A → B) := Eqvgen (CoeqRel f g)

-- The Coequalizer Setoid
instance CoeqSetoid {A B : Type} (f g : A → B): Setoid B := {
  r     := CoeqEqv f g
  iseqv := TEqvgen (CoeqRel f g)
}

-- The coequalizer Coeq f g
def Coeq {A B : Type} (f g : A → B) : Type := Quotient (CoeqSetoid f g)
```

It comes equipped with the quotient map from `B` to the coequalizer.

```
def prCoeq {A B : Type} (f g : A → B) : B → Coeq f g := @pr B (CoeqSetoid f g)
```

This projection satisfies that `(prCoeq f g) ∘ f = (prCoeq f g) ∘ g`.

```
theorem TCoeqPr {A B : Type} (f g : A → B) : (prCoeq f g) ∘ f = (prCoeq f g) ∘ g := by
  apply funext
  intro a
  apply Quotient.sound
  apply Eqvgen.base
  apply Exists.intro a
  apply And.intro
  exact rfl
  exact rfl
```

### 10.6.1 Universal property of the coequalizer

The **universal property** of the coequalizer characterizes it not merely as a quotient type, but as a universal solution to the problem of mediating between `f` and `g`. The universal property states that the pair `(Coeq f g, prCoeq f g)` is final among all pairs `(C, h)`, where `C` is a type and `h : B → C` is a function satisfying `h ∘ f = h ∘ g`.

That is, if `C` is a type and `h : B → C` is a function satisfying `h ∘ f = h ∘ g`, then there exists a unique function `u : (Coeq f g) → C` such that `u ∘ (prCoeq f g) = h`.

```
-- If there is another function h : B → C satisfying h ∘ f = h ∘ g, then there exists a function u :
    Coeq f g → C
def u {A B C : Type} {f g : A → B} {h : B → C} (h1 : h ∘ f = h ∘ g) : Coeq f g → C := by
  apply Quotient.lift h
  intro b1 b2 h2
  induction h2
  -- Base case
  rename_i c1 c2 h2
  apply Exists.elim h2
  intro a ⟨h2, ⟩h3
  calc
    h c1  = h (f a)   := congrArg h (h2.symm)
    _     = (h ∘ f) a := rfl
    _     = (h ∘ g) a := congrFun h1 a
    _     = h (g a)   := rfl
    _     = h c2      := congrArg h h3
  -- Rfl Case
  rename_i c
  exact rfl
  -- Symm Case
  rename_i c1 c2 _ h3
  exact h3.symm
  -- Trans Case
  rename_i c1 c2 c3 _ _ h4 h5
  exact h4.trans h5

-- The function u satisfies that u ∘ prCoeq f g = h
```

```
theorem TCoeqPrEq {A B C : Type} {f g : A → B} {h : B → C} (h1 : h ∘ f = h ∘ g) : (u h1) ∘ (prCoeq f g) = h := by
  apply funext
  intro b
  exact rfl

-- The function u is unique in the sense that if there is another function
-- v : Coeq f g → C satisfying v ∘ (prCoeq f g) = h, then v = u
theorem TCoeqUni {A B C : Type} {f g : A → B} {h : B → C} (h1 : h ∘ f = h ∘ g) (v : Coeq f g → C) (h2 : v ∘ (prCoeq f g) = h) : v = u h1 := by
  apply funext
  intro z
  have h3 : ∃ (b : B), Quotient.mk (CoeqSetoid f g) b = z := Quotient.exists_rep z
  apply Exists.elim h3
  intro b h4
  calc
    v z = v (prCoeq f g b)                 := congrArg v (h4.symm)
    _   = (v ∘ prCoeq f g) b               := rfl
    _   = h b                              := congrFun h2 b
    _   = ((u h1) ∘ (prCoeq f g)) b        := congrFun (TCoeqPrEq h1) b
    _   = (u h1) (prCoeq f g b)            := rfl
    _   = (u h1) z                         := congrArg (u h1) (h4)
```

In other words, any function from `B` that "coequalizes" `f` and `g` factors uniquely through the coequalizer. This property ensures that the coequalizer is the most general and canonical way to capture the quotient where the images the two functions are identified.

## 10.7 Exercises

### 10.7.1 Equivalences

```
-- The meet of two equivalence relations is an equivalence relation
instance TMeetEqv {A : Type} {R S : A → A → Prop} (hR : Equivalence R)
(hS : Equivalence S) : Equivalence (R ∧ S) := by sorry

-- If two setoids have equivalent underlying relations, the corresponding quotient types are equal
theorem TEqQuotype {A : Type} {R1 R2 : Setoid A}
(h : ∀ (a1 a2 : A), R1.r a1 a2 ↔ R2.r a1 a2) : Quotient R1 = Quotient R2 := by sorry
```

Prove the *The Universal Property of Quotient types*, which states that the Kernel of the projection function is precisely the original relation of the setoid.

```
-- Ker (pr R) = R.r
theorem TUPQuot {A : Type} {R : Setoid A} : Ker (pr R) = R.r := by sorry
```

### 10.7.2 Astriction

```
-- The astriction ast f is equal to pr ∘ f.
theorem TAst {A B : Type} {f : A → B} {S : Setoid B}: (@ast A B S f) =
(@pr B S) ∘ f := by sorry
```

### 10.7.3 Coastriction

Theorems `TUPCoast` and `TUPCoasttUn` establish the *universal property* of the coastriction of a function. The first result, `TUPCoast`, states that for any function `f : A → B` that respects a setoid relation `R` on `A` (i.e., for every pair `a1 a2 : A`, if `R.r a1 a2` then `f a1 = f a2`)), the function `f` can be expressed as the composition of its coastriction `coast f h` with the projection function `pr`, that is `f = coast ∘ pr`. The second result, `TUPCoastUn`, establishes the *uniqueness* of the coastriction. If there exists another function `g : Quotient R → B` such that `f = g ∘ pr`, then `g` must be exactly `coast f h`.

```
-- f = coast ∘ pr
theorem TUPCoast {A B : Type} {R : Setoid A} {f : A → B} (h : ∀ (a1 a2 : A),
R.r a1 a2 → (Ker f) a1 a2) : f =  (coast f h) ∘ (@pr A R) := by sorry

-- Unicity
```

```
theorem TUPCoastUn {A B : Type} {R : Setoid A} {f : A → B} (h : ∀ (a1 a2 : A),
R.r a1 a2 → (Ker f) a1 a2) (g : Quotient R → B) (h1 : f = g ∘ (@pr A R)) :
(coast f h) = g := by sorry
```

### 10.7.4 Isomorphisms

We next introduce the concept of *isomorphic types* and establish that, under `Classical.choice`, isomorphism defines an equivalence relation. We first define `Iso A B` as the subtype of all functions from `A` to `B` that are isomorphisms, meaning they admit a two-sided inverse. Two types `A` and `B` are then said to be *isomorphic*, written `A ≅ B`, if there exists at least one such isomorphism, formalized as the proposition `Nonempty (Iso A B)`. Prove that being isomorphic is an equivalence relation on `Type`.

```
-- The subtype of all isomorphisms from a type `A` to a type `B`
def Iso (A B : Type) := {f : A → B // isomorphism f}

-- Two types A and B are isomorphic if there is some isomorphism from `A` to `B`
def Isomorphic : Type → Type → Prop := by
  intro A B
  exact Nonempty (Iso A B)

-- Notation for Isomorphic types (`\cong`)
notation : 65 lhs:65 " ≅ " rhs:66 => Isomorphic lhs rhs

-- Being isomorphic is an equivalence relation
theorem TIsoEqv : Equivalence Isomorphic := by sorry
```

Assuming `Classical.choice`, every type `A` is isomorphic to the quotient of `A` by the diagonal relation, `A / Diag`.

```
theorem TDiag {A : Type} : A ≅ @QDiag A := by sorry
```

Under `Classical.choice`, any two `Nonempty` types `A` and `B` have isomorphic quotients `A / Total` and `B / Total`.

```
theorem TTotal {A B : Type} (hA : Nonempty A) (hB : Nonempty B) : @Qtotal A ≅ @Qtotal B := by sorry
```

Assuming `Classical.choice` prove that, for every function `f : A → B`, the quotient `A / Ker f` is isomorphic to `Im f`.

```
theorem TKerIm {A B : Type} (f : A → B) : QKer f ≅ Im f := by sorry
```

### 10.7.5 Coequalizers

```
-- The function prCoeq is an epimorphism
theorem TprCoeqEpi {A B : Type} {f g : A → B} : epimorphism (prCoeq f g) := by sorry

-- A monic prCoeq is an isomorphism
theorem TprCoeqMon {A B : Type} {f g : A → B} : monomorphism (prCoeq f g) → isomorphism (prCoeq f g) :=
    by sorry
```

# 11 Orders

In many areas of mathematics and computer science, we are interested in how elements compare. Can one element be considered *less* than another? Are two elements *incomparable*? Such questions motivate the study of *orders*, which capture various ways of comparing elements.

In this chapter, we begin by examining different kinds of order relations and how they combine to define *preorders* and *partial orders*. We'll look at concrete examples and highlight their differences. We then explore how order structures can be represented formally, particularly within a type-theoretic framework.

The chapter concludes with a series of exercises designed to deepen our understanding and give us hands-on practice with ordered structures.

## 11.1 Preorder

A **preorder** on a type `A` is a binary relation on `A` that is *reflexive* and *transitive*.

```
structure Preorder {A : Type} (R : A → A → Prop) : Prop where
  refl : ∀ (a : A), R a a
  trans : ∀ {a b c : A}, R a b → R b c → R a c
```

The keyword `structure` introduces a new structured proposition called `Preorder`, which is simply a collection of logical propositions. It has two fields, `refl` (reflexivity) and `trans` (transitivity). Now, statements of the form `Preoder R` are propositions and thus, can be proven.

## 11.2 Partial Order

A **partial order** is a preorder that is also *antisymmetric*.

```
structure PartialOrder {A : Type} (R : A → A → Prop) : Prop where
  toPreorder : Preorder R
  antisymm : ∀ {a b : A}, R a b → R b a → a = b
```

## 11.3 Partially Ordered Set

A **partially ordered set** (or **poset**) is a `structure` consisting of three components: a type `base`; a binary relation `R` on `base`; and a proof that this relation forms a partial order.

```
structure Poset where
  base : Type
  R    : base → base → Prop
  toPartialOrder : PartialOrder R
```

### 11.3.1 Special Elements

We say that `z` is a **least element** with respect to `R` if `R z a`, for every `a : A`. We say that `z` is a **greatest element** with respect to `R` if `R a z`, for every `a : A`.

```
-- Least
def Least {A : Type} (R : A → A → Prop) (z : A) : Prop := ∀ {a : A}, R z a
-- Greatest
def Greatest {A : Type} (R : A → A → Prop) (z : A) : Prop := ∀ {a : A}, R a z
```

We say that `z` is a **minimal element** with respect to `R` if, for every `a : A`, whenever `R a z` holds, it must follow that `a = z`. Similarly, we say that `z` is a **maximal element** with respect to `R` if, for every `a : A`, whenever `R z a` holds, it must follow that `a = z`.

```
-- Minimal
def Minimal {A : Type} (R : A → A → Prop) (z : A) : Prop := ∀ {a : A}, R a z → a = z
-- Maximal
def Maximal {A : Type} (R : A → A → Prop) (z : A) : Prop :=  ∀ {a : A}, R z a → a = z
```

### 11.3.2 Bounded Posets

A **bounded poset** is a poset that has both a **least element** and a **greatest element**.

```
structure BoundedPoset extends Poset where
  l : base
  least : Least R l
  g : base
  greatest : Greatest R g
```

### 11.3.3 Special Elements relative to a Subtype

We say that `z` is an **upper bound** of a subtype `P` if it is greater than or equal to every element of `Subtype P` with respect to the relation `R`. An upper bound `z` is called the **supremum** (or least upper bound) of `P` if, for any other upper bound `x`, the relation `R z x` holds—that is, `z` is less than or equal to every other upper bound. An element `z` is the **maximum** of `P` if it is both a supremum of `P` and an actual member of `Subtype P`.

```
-- UpperBound
def UpperBound {A : Type} (R : A → A → Prop) (P : A → Prop) (z : A) : Prop :=∀
 (a : A), P a → R a z
-- Supremum
structure Supremum {A : Type} (R : A → A → Prop) (P : A → Prop) (z : A) : Prop where
  -- Upper Bound
  UB : (UpperBound R P z)
  -- Least Upper Bound
  LUB : ∀ (x : A), (UpperBound R P x → R z x)
-- Maximum
structure Maximum {A : Type} (R : A → A → Prop) (P : A → Prop) (z : A) : Prop where
  -- Supremum
  toSupremum : (Supremum R P z)
  -- In Subtype P
  Sub : P z
```

Conversely, we say that `z` is a **lower bound** of a subtype `P` if it is smaller than or equal to every element of `Subtype P` with respect to the relation `R`. A lower bound `z` is called the **infimum** (or greatest lower bound) of `P` if, for any other lower bound `x`, the relation `R x z` holds—that is, `z` is greater than or equal to every other upper bound. An element `z` is the **minimum** of `P` if it is both an infimum of `P` and an actual member of `Subtype P`.

```
-- LowerBound
def LowerBound {A : Type} (R : A → A → Prop) (P : A → Prop) (z : A) : Prop := ∀ (a : A), P a → R z a
-- Infimum
structure Infimum {A : Type} (R : A → A → Prop) (P : A → Prop) (z : A) : Prop where
  -- Lower Bound
  LB : (LowerBound R P z)
  -- Greatest Lower Bound
  GLB : ∀ (x : A), (LowerBound R P x → R x z)
-- Minimum
structure Minimum {A : Type} (R : A → A → Prop) (P : A → Prop) (z : A) : Prop where
  -- Infimum
  toInfimum : (Infimum R P z)
  -- In Subtype P
  Sub : P z
```

## 11.4 Lattice

A **lattice** is an abstract mathematical structure that can be defined in two equivalent ways: either order-theoretically, as a partially ordered set satisfying certain conditions, or algebraically, as a structure equipped with operations that obey specific laws.

### 11.4.1 Lattice as a poset

A *lattice* is a partially ordered set in which every pair of elements has a unique supremum (also called the *join*) and a unique infimum (also called the *meet*). The following definition introduces a structure `Lattice` that builds on an existing `Poset` by adding operations and properties for `meet` and `join`. The corresponding fields `infimum` and `supremum` are proofs that `meet a b` and `join a b` do indeed satisfy the formal definitions of infimum and supremum with respect to the underlying order relation `R`.

```
@[ext] structure Lattice extends Poset where
  meet : base → base → base
  infimum : ∀ {a b : base}, Infimum R (fun (x : base) => (x = a) ∨ (x = b)) (meet a b)
  join : base → base → base
  supremum : ∀ {a b : base}, Supremum R (fun (x : base) => (x = a) ∨ (x = b)) (join a b)
```

The `@[ext]` attribute is a convenience provided by Lean's metaprogramming framework. It automatically generates an **extensionality lemma** for the structure, allowing users to prove equality between two lattice instances by showing that all their corresponding components are equal. In practice, this means that to show two lattices are equal, it suffices to prove that their base types and the relations on them are the same, and that the `meet` and `join` operations agree on all inputs. This can simplify proofs and reasoning about structures built on top of `Lattice`, as we will see below.

### 11.4.2 Lattice as an algebra

An alternative way to describe a lattice is as an *algebraic structure* consisting of a `base` type equipped with two binary operations, `meet` and `join`. These operations are required to be **commutative** and **associative**, and they must satisfy the **absorption laws**, as described below.

```
@[ext] structure LatticeAlg where
  base : Type
  meet : base → base → base
  join : base → base → base
  meetcomm : ∀ {a b : base}, meet a b = meet b a
  joincomm : ∀ {a b : base}, join a b = join b a
  meetass  : ∀ {a b c : base}, meet (meet a b) c = meet a (meet b c)
  joinass  : ∀ {a b c : base}, join (join a b) c = join a (join b c)
  abslaw1  : ∀ {a b : base}, join a (meet a b) = a
  abslaw2  : ∀ {a b : base}, meet a (join a b) = a
```

This algebraic perspective is equivalent to the order-theoretic definition and emphasizes the operational behavior of meets and joins rather than their characterization via suprema and infima.

There are usually 2 more laws regarding the **idempotency** of the `meet` and `join` operations that can be derived from the other axioms.

```
-- meet is idempotent
theorem meetidpt {C : LatticeAlg} : ∀ (a : C.base), C.meet a a = a := by
  intro a
  calc
    C.meet a a = C.meet a (C.join a (C.meet a a)) := congrArg (C.meet a) C.abslaw1.symm
    _          = a                                := C.abslaw2

-- join is idempotent
theorem joinidpt {C : LatticeAlg} : ∀ (a : C.base), C.join a a = a := by
  intro a
  calc
    C.join a a = C.join a (C.meet a (C.join a a)) := congrArg (C.join a) C.abslaw2.symm
    _          = a                                := C.abslaw1
```

The following result will be of interest later.

```
theorem meetjoin {C : LatticeAlg} : ∀ {a b : C.base}, (C.meet a b = a) ↔ (C.join a b = b) := by
  intro a b
  apply Iff.intro
  -- C.meet a b = a → C.join a b = b
  intro h
  rw [C.meetcomm] at h
  rw [C.joincomm, h.symm]
  exact C.abslaw1
  -- C.join a b = b → C.meet a b = a
  intro h
```

```
  rw [h.symm]
  exact C.abslaw2
```

### 11.4.3 From Lattice to LatticeAlg

Any Lattice structure naturally induces a corresponding LatticeAlg structure on its underlying base type with the meet and join operations from the lattice. The proof below demonstrates this construction, using the refine keyword to explicitly specify the values of all required LatticeAlg fields.

```
def LatticetoLatticeAlg : Lattice → LatticeAlg := by
  intro C
  refine {
    base := C.base,
    meet := C.meet,
    join := C.join,
    meetcomm := by
      intro a b
      apply C.toPoset.toPartialOrder.antisymm
      -- C.R (C.meet a b) (C.meet b a)
      apply C.infimum.GLB
      intro z h
      cases h
      -- b
      rename_i hz
      apply C.infimum.LB
      exact Or.inr hz
      -- a
      rename_i hz
      apply C.infimum.LB
      exact Or.inl hz
      -- C.R (C.meet b a) (C.meet a b)
      apply C.infimum.GLB
      intro z h
      cases h
      -- a
      rename_i hz
      apply C.infimum.LB
      exact Or.inr hz
      -- b
      rename_i hz
      apply C.infimum.LB
      exact Or.inl hz
    joincomm := by
      intro a b
      apply C.toPoset.toPartialOrder.antisymm
      -- C.R (C.join a b) (C.join b a)
      apply C.supremum.LUB
      intro z h
      cases h
      -- a
      rename_i hz
      apply C.supremum.UB
      exact Or.inr hz
      -- b
      rename_i hz
      apply C.supremum.UB
      exact Or.inl hz
      -- C.R (C.join b a) (C.join a b)
      apply C.supremum.LUB
      intro z h
      cases h
      -- b
      rename_i hz
      apply C.supremum.UB
      exact Or.inr hz
      -- a
      rename_i hz
      apply C.supremum.UB
      exact Or.inl hz
    meetass := by
      intro a b c
```

```
      apply C.toPoset.toPartialOrder.antisymm
      -- C.R (C.meet (C.meet a b) c) (C.meet a (C.meet b c))
      apply C.infimum.GLB
      intro z h
      cases h
      -- a
      rename_i hz
      have h1 : C.R (C.meet (C.meet a b) c) (C.meet a b) := by
        apply C.infimum.LB
        exact Or.inl rfl
      have h2 : C.R (C.meet a b) z := by
        apply C.infimum.LB
        exact Or.inl hz
      exact C.toPoset.toPartialOrder.toPreorder.trans h1 h2
      -- C.meet b c
      rename_i hz
      rw [hz]
      apply C.infimum.GLB
      intro d hd
      cases hd
      -- b
      rename_i hd
      rw [hd]
      have h1 : C.R (C.meet (C.meet a b) c) (C.meet a b) := by
        apply C.infimum.LB
        exact Or.inl rfl
      have h2 : C.R (C.meet a b) b := by
        apply C.infimum.LB
        exact Or.inr rfl
      exact C.toPoset.toPartialOrder.toPreorder.trans h1 h2
      -- c
      rename_i hd
      apply C.infimum.LB
      exact Or.inr hd
      -- C.R (C.meet a (C.meet b c)) (C.meet (C.meet a b) c)
      apply C.infimum.GLB
      intro z h
      cases h
      -- C.meet a b
      rename_i hz
      rw [hz]
      apply C.infimum.GLB
      intro d hd
      cases hd
      -- a
      rename_i hd
      apply C.infimum.LB
      exact Or.inl hd
      -- b
      rename_i hd
      rw [hd]
      have h1 : C.R (C.meet a (C.meet b c)) (C.meet b c) := by
        apply C.infimum.LB
        exact Or.inr rfl
      have h2 : C.R (C.meet b c) b := by
        apply C.infimum.LB
        exact Or.inl rfl
      exact C.toPoset.toPartialOrder.toPreorder.trans h1 h2
      -- c
      rename_i hz
      have h1 : C.R (C.meet a (C.meet b c)) (C.meet b c) := by
        apply C.infimum.LB
        exact Or.inr rfl
      have h2 : C.R (C.meet b c) z := by
        apply C.infimum.LB
        exact Or.inr hz
      exact C.toPoset.toPartialOrder.toPreorder.trans h1 h2
    joinass := by
      intro a b c
      apply C.toPoset.toPartialOrder.antisymm
      -- C.R (C.join (C.join a b) c) (C.join a (C.join b c))
      apply C.supremum.LUB
      intro z h
```

```
      cases h
      -- C.join a b
      rename_i hz
      rw [hz]
      apply C.supremum.LUB
      intro d hd
      cases hd
      -- a
      rename_i hd
      apply C.supremum.UB
      exact Or.inl hd
      -- b
      rename_i hd
      have h1 : C.R d (C.join b c) := by
        apply C.supremum.UB
        exact Or.inl hd
      have h2 : C.R (C.join b c) (C.join a (C.join b c)) := by
        apply C.supremum.UB
        exact Or.inr rfl
      exact C.toPoset.toPartialOrder.toPreorder.trans h1 h2
      -- c
      rename_i hz
      have h1 : C.R z (C.join b c) := by
        apply C.supremum.UB
        exact Or.inr hz
      have h2 : C.R (C.join b c) (C.join a (C.join b c)) := by
        apply C.supremum.UB
        exact Or.inr rfl
      exact C.toPoset.toPartialOrder.toPreorder.trans h1 h2
      -- C.R (C.join a (C.join b c)) (C.join (C.join a b) c)
      apply C.supremum.LUB
      intro z h
      cases h
      -- a
      rename_i hz
      have h1 : C.R z (C.join a b) := by
        apply C.supremum.UB
        exact Or.inl hz
      have h2 : C.R (C.join a b) (C.join (C.join a b) c) := by
        apply C.supremum.UB
        exact Or.inl rfl
      exact C.toPoset.toPartialOrder.toPreorder.trans h1 h2
      -- C.join b c
      rename_i hz
      rw [hz]
      apply C.supremum.LUB
      intro d hd
      cases hd
      -- b
      rename_i hd
      have h1 : C.R d (C.join a b) := by
        apply C.supremum.UB
        exact Or.inr hd
      have h2 : C.R (C.join a b) (C.join (C.join a b) c) := by
        apply C.supremum.UB
        exact Or.inl rfl
      exact C.toPoset.toPartialOrder.toPreorder.trans h1 h2
      -- c
      rename_i hd
      apply C.supremum.UB
      exact Or.inr hd
    abslaw1 := by
      intro a b
      apply C.toPoset.toPartialOrder.antisymm
      -- C.R (C.join a (C.meet a b)) a
      apply C.supremum.LUB
      intro d hd
      cases hd
      -- a
      rename_i hd
      rw [hd]
      apply C.toPoset.toPartialOrder.toPreorder.refl
      -- C.meet a b
```

```
      rename_i hd
      rw [hd]
      apply C.infimum.LB
      exact Or.inl rfl
      -- C.R a (C.join a (C.meet a b))
      apply C.supremum.UB
      exact Or.inl rfl
    abslaw2 := by
      intro a b
      apply C.toPoset.toPartialOrder.antisymm
      -- C.R (C.meet a (C.join a b)) a
      apply C.infimum.LB
      exact Or.inl rfl
      -- C.R a (C.meet a (C.join a b))
      apply C.infimum.GLB
      intro d hd
      cases hd
      -- a
      rename_i hd
      rw [hd]
      apply C.toPoset.toPartialOrder.toPreorder.refl
      -- C.join a b
      rename_i hd
      rw [hd]
      apply C.supremum.UB
      exact Or.inl rfl
  }
```

### 11.4.4 From `LatticeAlg` to `Lattice`

Conversely, every `LatticeAlg` structure gives rise to a corresponding `Lattice` structure on its underlying `base` type. The construction begins by defining a partial order `LAR` on the `base`, as shown below. We will say that `a ≤ b` if, and only if, `meet a b = a`.

```
def LAR {C : LatticeAlg} : C.base → C.base → Prop := by
  intro a b
  exact C.meet a b = a
```

The relation `LAR` is a `Preorder`.

```
theorem TLARPreorder {C : LatticeAlg} : Preorder (@LAR C) := by
  apply Preorder.mk
  -- refl
  intro a
  rw [LAR]
  exact meetidpt a
  -- trans
  intro a b c h1 h2
  rw [LAR] at *
  rw [h1.symm, C.meetass, h2]
```

The relation `LAR` is a `PartialOrder`.

```
theorem TLARPartialOrder {C : LatticeAlg} : PartialOrder (@LAR C) := by
  apply PartialOrder.mk
  -- toPreorder
  exact TLARPreorder
  -- antisymm
  intro a b h1 h2
  calc
    a = C.meet a b := h1.symm
    _ = C.meet b a := C.meetcomm
    _ = b          := h2
```

In summary, we are now ready to prove that every `LatticeAlg` structure induces a corresponding `Lattice` structure on its base type, preserving the same `meet` and `join` operations.

```
def LatticeAlgtoLattice : LatticeAlg → Lattice := by
  intro C
  refine {
    toPoset := {
```

```
    base := C.base,
    R := @LAR C,
    toPartialOrder := TLARPartialOrder
  }
  meet := C.meet,
  infimum := by
    intro a b
    apply Infimum.mk
    -- LowerBound
    intro z hz
    cases hz
    -- a
    rename_i hz
    rw [hz]
    have h1 : C.meet (C.meet a b) a = (C.meet a b) := by
      calc
        C.meet (C.meet a b) a = C.meet a (C.meet a b) := C.meetcomm.symm
        _                     = C.meet (C.meet a a) b := C.meetass.symm
        _                     = C.meet a b            := congrArg (fun x => C.meet x b) (meetidpt a)
    exact h1
    -- b
    rename_i hz
    rw [hz]
    have h1 : C.meet (C.meet a b) b = (C.meet a b) := by
      calc
        C.meet (C.meet a b) b = C.meet a (C.meet b b) := C.meetass
        _                     = C.meet a b            := congrArg (fun x => C.meet a x) (meetidpt b)
    exact h1
    -- Greatest LowerBound
    intro x h
    have ha : C.meet x a = x := by
      apply h
      exact Or.inl rfl
    have hb : C.meet x b = x := by
      apply h
      exact Or.inr rfl
    have h1 : C.meet x (C.meet a b) = x := by
      calc
        C.meet x (C.meet a b) = C.meet (C.meet x a) b := C.meetass.symm
        _                     = C.meet x b            := congrArg (fun z => C.meet z b) ha
        _                     = x                     := hb
    exact h1
  join := C.join,
  supremum := by
    intro a b
    apply Supremum.mk
    -- UpperBound
    intro z hz
    cases hz
    -- a
    rename_i hz
    rw [hz]
    exact C.abslaw2
    -- b
    rename_i hz
    rw [hz, C.joincomm]
    exact C.abslaw2
    -- Least UpperBound
    intro x h
    have ha : C.meet a x = a := by
      apply h
      exact Or.inl rfl
    have hb : C.meet b x = b := by
      apply h
      exact Or.inr rfl
    have hax : C.join a x = x := meetjoin.mp ha
    have hbx : C.join b x = x := meetjoin.mp hb
    have h1 : C.join (C.join a b) x = x := by
      calc
        C.join (C.join a b) x = C.join a (C.join b x) := C.joinass
        _                     = C.join a x            := congrArg (fun z => C.join a z) hbx
        _                     = x                     := hax
    exact meetjoin.mpr h1
```

```
}
```

### 11.4.5 Compositions

The `ext` attribute now allows us to prove that the two constructions defined above are mutually inverse.

**`LatticeAlgtoLattice ∘ LatticetoLatticeAlg = id`**

```
theorem TIdLattice : LatticeAlgtoLattice ∘ LatticetoLatticeAlg = id := by
  funext C
  apply Lattice.ext
  -- base
  exact rfl
  -- R
  have hR : (LatticeAlgtoLattice (LatticetoLatticeAlg C)).R = C.R := by
    funext a b
    apply propext
    apply Iff.intro
    -- ((LatticeAlgtoLattice ∘ LatticetoLatticeAlg) C).R a b → C.R a b
    intro h
    have hs : C.meet a b = a := h
    rw [hs.symm]
    apply C.infimum.LB
    exact Or.inr rfl
    -- C.R a b → (LatticeAlgtoLattice (LatticetoLatticeAlg C)).R a b
    intro h
    have hs : C.meet a b = a := by
      apply C.toPoset.toPartialOrder.antisymm
      -- C.R (C.meet a b) a
      apply C.infimum.LB
      exact Or.inl rfl
      -- C.R a (C.meet a b)
      apply C.infimum.GLB
      intro z hz
      cases hz
      -- a
      rename_i hz
      rw [hz]
      exact C.toPoset.toPartialOrder.toPreorder.refl a
      -- b
      rename_i hz
      rw [hz]
      exact h
    exact hs
  exact heq_of_eq hR
  -- meet
  exact HEq.refl C.meet
  -- join
  exact HEq.refl C.join
```

**`LatticetoLatticeAlg ∘ LatticeAlgtoLattice = id`**

```
theorem TIdLatticeAlg : LatticetoLatticeAlg ∘ LatticeAlgtoLattice = id := by
  funext C
  apply LatticeAlg.ext
  -- base
  exact rfl
  -- meet
  exact HEq.refl C.meet
  -- join
  exact HEq.refl C.join
```

### 11.4.6 Distributive Lattice

A **distributive lattice** is a lattice satisfying an extra law regarding the distributivity of `meet` over `join`.

```
@[ext] structure DistLatticeAlg extends LatticeAlg where
  dist : ∀ {a b c : base}, meet a (join b c) = join (meet a b) (meet a c)
```

## 11.5 Complete Lattice

A **complete lattice** is a partially ordered set in which every subtype has both an infimum, that we will call `meet`, and a supremum, that we will call `join`.

```
structure CompleteLattice extends Poset where
  meet : (base → Prop) → base
  infimum : ∀ {P : base → Prop}, Infimum R P (meet P)
  join : (base → Prop) → base
  supremum : ∀ {P : base → Prop}, Supremum R P (join P)
```

### 11.5.1 From CompleteLattice to Lattice

Clearly, every `CompleteLattice` is, in particular, a `Lattice`.

```
def CompleteLatticetoLattice : CompleteLattice → Lattice := by
  intro C
  refine {
    toPoset := C.toPoset,
    meet := (fun a b => C.meet (fun (x : C.base) => (x = a) ∨ (x = b))),
    infimum := fun {a b} => C.infimum
    join := (fun a b => C.join (fun (x : C.base) => (x = a) ∨ (x = b))),
    supremum := fun {a b} => C.supremum
  }
```

### 11.5.2 From CompleteLattice to BoundedPoset

Also, every `CompleteLattice` is, in particular, a `BoundedPoset`. To prove this fact, we need to prove that the supremum of the `PFalse` predicate is, precisely, the `Least` element (exercise) and the infimum of the `PFalse` predicate is, precisely, the `Greatest` element (exercise). Thus, every `CompleteLattice`, which has both infima and suprema for all subtypes, contains a `least` and a `greatest` element, i.e., is a `BoundedPoset`.

```
def CompleteLatticetoBoundedPoset : CompleteLattice → BoundedPoset := by
  intro C
  refine {
    toPoset := C.toPoset,
    l := C.join (PFalse),
    least := (TLeastSupPFalse C.R (C.join PFalse)).mpr (C.supremum)
    g := C.meet (PFalse),
    greatest := (TGreatestInfPFalse C.R (C.meet PFalse)).mpr (C.infimum)
  }
```

## 11.6 Exercises

### 11.6.1 Inverse Partial Order

```
-- If R is a preorder, then the inverse relation R^ is also a preorder
theorem TPreorderInv {A : Type} (R : A → A → Prop) : Preorder R → Preorder (inverse R) := by sorry

-- If R is a partial order, then the inverse relation R^ is also a partial order
theorem TPartialOrderInv {A : Type} (R : A → A → Prop) : PartialOrder R → PartialOrder (inverse R) := by
     sorry
```

### 11.6.2 Special Elements

```
-- If R is a partial order and z1 and z2 are least elements, then they are equal.
theorem LeastUnique {A : Type} (R : A → A → Prop) (z1 z2 : A) (h : PartialOrder R) (h1 : Least R z1) (h2 : Least R z2) : z1 = z2 := by sorry

-- If R is a partial order and z1 and z2 are greatest elements, then they are equal.
theorem GreatestUnique {A : Type} (R : A → A → Prop) (z1 z2 : A) (h : PartialOrder R) (h1 : Greatest R z1) (h2 : Greatest R z2) : z1 = z2  := by sorry

-- If R is a partial order and z is the least element, then it is a minimal element
def LeasttoMinimal {A : Type} (R : A → A → Prop) (z : A) (h : PartialOrder R) : Least R z → Minimal R z := by sorry

-- If R is a partial order and z is the greatest element, then it is a maximal element
def GreatesttoMaximal {A : Type} (R : A → A → Prop) (z : A) (h : PartialOrder R) : Greatest R z → Maximal R z := by sorry

-- A least element for R is a greatest element for R^
def LeasttoGreatestInv {A : Type} (R : A → A → Prop) (z : A) : Least R z → Greatest (inverse R) z := by sorry

-- A greatest element for R is a least element for R^
def GreatesttoLeastInv {A : Type} (R : A → A → Prop) (z : A) : Greatest R z → Least (inverse R) z := by sorry

-- A minimal element for R is a maximal element for R^
def MinimaltoMaximalInv {A : Type} (R : A → A → Prop) (z : A) : Minimal R z → Maximal (inverse R) z := by sorry

-- A maximal element for R is a minimal element for R^
def MaximaltoMinimalInv {A : Type} (R : A → A → Prop) (z : A) : Maximal R z → Minimal (inverse R) z := by sorry
```

### 11.6.3 Restriction

```
-- The Restriction of a relation to a Subtype
def Restriction {A : Type} (R : A → A → Prop) (P : A → Prop) : Subtype P → Subtype P → Prop := by
  intro a1 a2
  exact R a1.val a2.val

-- If R is a preorder then Restriction R P, for a predicate P, is a preorder
theorem TPRestriction {A : Type} (R : A → A → Prop) (P : A → Prop) : Preorder R → Preorder (Restriction R P) := by sorry

-- If R is a partial order then Restriction R P, for a predicate P, is a partial order
theorem TPORestriction {A : Type} (R : A → A → Prop) (P : A → Prop) :
PartialOrder R → PartialOrder (Restriction R P) := by sorry
```

### 11.6.4 Special Elements relative to a Subtype

```
-- The supremum of the False predicate is the least element
theorem TLeastSupPFalse {A : Type} (R : A → A → Prop) (z : A) : Least R z ↔ Supremum R PFalse z := by sorry

-- The infimum of the False predicate is the greatest element
theorem TGreatestInfPFalse {A : Type} (R : A → A → Prop) (z : A) : Greatest R z ↔ Infimum R PFalse z := by sorry

-- The infimum of the True predicate is the least element
theorem TLeastInfPTrue {A : Type} (R : A → A → Prop) (z : A) : Least R z ↔ Infimum R PTrue z := by sorry

-- The supremum of the True predicate is the greatest element
theorem TGreatestSupPTrue {A : Type} (R : A → A → Prop) (z : A) : Greatest R z ↔ Supremum R PTrue z := by sorry
```

## 11.6.5 (N, ≤)

```
-- The ≤ relation for N
def NLeq : N → N → Prop := by
  intro n m
  exact ∃ (k : N), n + k = m

-- Notation for ≤
notation : 65 lhs:65 " ≤ " rhs:66 => NLeq lhs rhs

-- ≤ is a preorder
theorem TPreorderNLeq : Preorder NLeq := by sorry

-- ≤ is a partial order
theorem TPartialOrderNLeq : PartialOrder NLeq := by sorry

-- (N, ≤) is a partially ordered type
def instPosetNLeq : Poset := by sorry

-- z is the least element
theorem TNLeqzL : ∀ {n : N}, z ≤ n := by sorry

-- No s n is below z
theorem TNLeqzR : ∀ {n : N}, ¬ (s n ≤ z) := by sorry

-- If n ≤ m then s n ≤ s m
theorem TNLeqSuccT : ∀ {n m : N}, (n ≤ m) → (s n ≤ s m) := by sorry

-- If n ≰ m then s n ≰ s m
theorem TNLeqSuccF : ∀ {n m : N}, (¬ (n ≤ m)) → (¬ (s n ≤ s m)) := by sorry

-- ≤ is decidable
def instDecidableNLeq : ∀ {n m : N}, Decidable (n ≤ m) := by sorry

-- min n m is a lower bound of n
theorem TMinNLeqL : ∀ {n m : N}, (mini n m) ≤ n := by sorry

-- min n m is a lower bound of m
theorem TMinNLeqR : ∀ {n m : N}, (mini n m) ≤ m := by sorry

-- min n m is the infimum for n and m
theorem TInfNLeq : ∀ {n m : N}, Infimum NLeq (fun (x : N) => (x = n) ∨ (x = m)) (mini n m) := by sorry

-- max n m is an upper bound of n
theorem TMaxNLeqL : ∀ {n m : N}, n ≤ (maxi n m) := by sorry

-- max n m is an upper bound of m
theorem TMaxNLeqR : ∀ {n m : N}, m ≤ (maxi n m) := by sorry

-- max n m is the supremum of n and m
theorem TSupNLeq : ∀ {n m : N}, Supremum NLeq (fun (x : N) => (x = n) ∨ (x = m)) (maxi n m) := by sorry

-- (N, ≤) is a lattice
def instLatticeNLeq : Lattice := by sorry

-- min n (max m p) = max (min n m) (min n p)
theorem TDistNLeq : ∀ {n m p : N}, mini n (maxi m p) = maxi (mini n m) (mini n p) := by sorry

-- (N≤,) is a distributive lattice
def instDistLatticeAlgNLeq : DistLatticeAlg := by sorry
```

## 11.6.6 (N, |)

```
-- The | (divisor) relation for N
def NDiv : N → N → Prop := by
  intro n m
  exact ∃ (k : N), n * k = m

-- Notation for divisor (\mid)
notation : 65 lhs:65 " | " rhs:66 => NDiv lhs rhs

```

```
-- | is a preorder
theorem TPreorderNDiv : Preorder NDiv := by sorry

-- | is a partial order
theorem TPartialOrderNDiv : PartialOrder NDiv := by sorry

-- (N, |) is a partially ordered type
def instPosetNDiv : Poset := by sorry

-- one is the least element for |
theorem TNDivOne : Least NDiv one := by sorry

-- z is the `greatest` element for |``
theorem TNDivZ : Greatest NDiv z := by sorry

-- (N, |) is a bounded partially ordered type
def instBoundedPosetNDiv : BoundedPoset := by sorry

-- z does not divide any successor
theorem TNDivzL : ∀ {n : N}, ¬ (z | s n) := by sorry

-- (N, |) is a lattice
def instLatticeNDiv : Lattice := by sorry
```

### 11.6.7 (Prop, →)

```
-- The → relation for Prop
def PropLeq : Prop → Prop → Prop := by
  intro P Q
  exact P → Q

-- → is a preorder
theorem TPreorderPropLeq : Preorder PropLeq := by sorry

-- → is a partial order
theorem TPartialOrderPropLeq : PartialOrder PropLeq := by sorry

-- (Prop, →) is a partially ordered type
def instPosetPropLeq : Poset := by sorry

-- False is the least element for →
theorem TPropLeqFalse : Least PropLeq False := by sorry

-- True is the greatest element for →
theorem TPropLeqTrue : Greatest PropLeq True := by sorry

-- (Prop, →) is a bounded partially ordered type
def instBoundedPropLeq : BoundedPoset := by sorry

-- (Prop, ∧, ∨) is a lattice (as an algebra)
def instLatticeAlgProp : LatticeAlg := by sorry

-- (Prop, →) is a complete lattice
def instCompleteLatticeProp : CompleteLattice := by sorry
```

# 12 Empty and Unit types

In this chapter, we explore two of the simplest—but most fundamental—types in Lean: the `Empty` type and the `Unit` type. Though these types may seem trivial at first glance, they play a crucial role in the logic and structure of formal proofs.

## 12.1 Empty

The **Empty type**, written as `Empty`, is a type with *no* inhabitants.

If we `#print Empty`, Lean returns:

```
inductive Empty : Type
number of parameters: 0
constructors:
```

This means that if we have a value of type `Empty`, we can derive a value of any other type from it. In Lean, this is done using the `Empty.elim` function, which expresses the logical principle that from falsehood, anything follows.

```
def emptyToAny {A : Type} : Empty → A := by
  intro x
  exact Empty.elim x
```

An interesting property of the `emptyToAny` function is that it is *unique up to definitional equality*; any two functions with type `Empty → A` are definitionally the same.

```
theorem emptyToAnyUnique {A : Type} {f g : Empty → A} : f = g := by
  funext x
  exact Empty.elim x
```

This implies that the `emptyToAny` function with codomain `Empty` is, in particular, the identity function on `Empty`.

```
theorem emptyToAnyId : @emptyToAny Empty = id := by
  funext x
  exact Empty.elim x
```

## 12.2 Unit

The **Unit type**, written as `Unit`, is a type with *exactly one* inhabitant: `Unit.unit`, usually just written `()`.

If we `#print Unit`, Lean returns:

```
@[reducible] def Unit : Type :=
PUnit
```

There is always a function from any type to the `Unit` type, since we can simply ignore the input and return `()`.

```
def anyToUnit {A : Type} : A → Unit := by
  intro _
  exact ()
```

The `anyToUnit` function is *unique up to definitional equality*; any two functions with type `A → Unit` are definitionally the same.

```
theorem anyToUnitUnique {A : Type} {f g : A → Unit} : f = g := by
  funext x
  exact rfl
```

This implies that when the domain is `Unit`, the `anyToUnit` function is, in particular, the identity function on `Unit`.

```
theorem anyToUnitId : @anyToUnit Unit = id := by
  funext x
  exact rfl
```

## 12.3 Exercises

### 12.3.1 Empty

```
-- All the elements of Empty are equal
theorem emptyUnique : ∀ (x y : Empty), x = y := by sorry

-- The emptyToAny function is injective
theorem emptyToAnyInj {A : Type} : injective (@emptyToAny A) := by sorry

-- Under Classical.choice, if the emptyToAny function is surjective, the codomain cannot be Nonempty
theorem emptyToAnySurj {A : Type} : surjective (@emptyToAny A) ↔ ¬ (Nonempty A) := by sorry
```

### 12.3.2 Unit

```
-- All the elements of Unit are equal
theorem unitUnique : ∀ (x y : Unit), x = y := by sorry

-- The anyToUnit function is injective if, and only if, the domain has only one element
theorem anyToUnitInj {A : Type} : injective (@anyToUnit A) ↔ ∀( (a1 a2 : A), a1 = a2) := by sorry

-- Under Classical.choice, the anyToUnit function is surjective if and only if the domain is Nonempty
theorem anyToUnitSurj {A : Type} : surjective (@anyToUnit A) ↔ Nonempty A := by sorry
```

# 13 Product and Sum types

This chapter explores **product and sum types**, two foundational constructs in type theory.

We begin with the **product type**, which models the combination of two types into a single type whose elements are pairs. The section introduces the construction of product types, how to access their components, and their role in structuring data. The **universal property of the product** is then presented, characterizing the product as the type that uniquely supports projections and pairing. The concept is extended to the **product of a family of types**, generalizing the binary product to arbitrary indexed collections.

Next, we examine the **sum type**, which represents a value that belongs to one of several types. It captures alternatives or choices between types and is central to defining tagged unions. The **universal property of the sum** provides its defining characteristic: a type that uniquely supports case analysis using injections. This is generalized to the **sum of a family of types**, where each summand is indexed, enabling more expressive and flexible type constructions.

The chapter concludes with a set of **exercises** aimed at reinforcing understanding through practical applications and formal reasoning about product and sum types.

## 13.1 Product type

A **product type** combines two types into a single type whose values consist of pairs drawn from each component. In Lean, the product type of `A` and `B` is written as `Prod A B` or, alternatively as, `A × B`, using the `×` symbol (typed as `\times`).

If we `#print Prod`, Lean returns:

```
structure Prod.{u, v} : Type u → Type v → Type (max u v)
number of parameters: 2
constructor:
Prod.mk : { α : Type u } → { β : Type v } → α → β → α × β
fields:
fst : α
snd : β
```

Lean defines the product type internally using the `Prod` structure. This definition shows that `Prod` takes two types—one from universe `u` and one from universe `v`—and returns a type in the larger of the two universes, `max u v`. The constructor `Prod.mk` builds a pair from values of types `α` and `β`, and the resulting pair belongs to the product type `α × β`. The structure has two fields: `fst`, which retrieves the first component of the pair, and `snd`, which retrieves the second component.

This definition implies that to construct a value of type `A × B`, we must provide **both** a value of type `A` and a value of type `B`. The constructor `Prod.mk` enforces this requirement: given `a : A` and `b : B`, the expression `Prod.mk a b` (or simply `⟨a, b⟩` using Lean's pair notation) yields a value of type `A × B`. This reflects the nature of product types as containers of exactly one value from each of their component types. In other words, a product type does not represent a choice between `A` and `B`, but rather a combination of the two.

```
def toPair {A B : Type} : A → B → A × B := by
  intro a b
  exact Prod.mk a b -- alternatively ⟨a,\ ⟩b
```

The `Prod` type provides two **projections**, `π1` and `π1`, or, alternatively, `fst` and `snd`, which allow us to extract the individual components of a product. Given a value `p : A × B`, the expression `p.fst` retrieves the first element (of type `A`), and `p.snd` retrieves the second element (of type `B`).

```
-- projection on the first component
def π1 {A B : Type} : (A × B) → A := by
  intro p
  exact p.fst

-- projection on the second component
```

```
def π2 {A B : Type} : (A × B) → B := by
  intro p
  exact p.snd
```

Two values of type `A × B` are **equal** if, and only if, their respective components are equal. That is, given `p1 p2 : A × B`, we have `p1 = p2` precisely when `p1.fst = p2.fst` and `p1.snd = p2.snd`. Lean provides the theorem `Prod.ext` to formalize and prove such equalities.

```
theorem prodEq {A B : Type} (p1 p2 : A × B) : ( p1 = p2 ) ↔ ( π1 p1 = π1 p2 ) ∧ ( π2 p1 = π2 p2 ) := by
  apply Iff.intro
  -- p1 = p2 → π1 p1 = π1 p2 ∧ π2 p1 = π2 p2
  intro h
  apply And.intro
  exact congrArg π1 h
  exact congrArg π2 h
  -- π1 p1 = π1 p2 ∧ π2 p1 = π2 p2 → p1 = p2
  intro ⟨ h1, h2 ⟩
  apply Prod.ext
  exact h1
  exact h2
```

### 13.1.1 Universal property of the product

The **universal property of the product type** characterizes it as the *best* type that supports pairing of data. Specifically, given types `A`, `B`, and `C`, and functions `f : C → A` and `g : C → B`, there exists a unique function `h : C → A × B` such that the projections of `h` recover `f` and `g`; that is, $\pi_1 \circ h = f$ and $\pi_2 \circ h = g$. In Lean, this function `h` is constructed by sending each `c : C` to the pair `(f c, g c)`.

```
def toProd {A B C : Type} (f : C → A) (g : C → B) : (C → A × B) := by
  intro c
  exact Prod.mk (f c) (g c)
```

This function has the key property that composing it with the product projections recovers the original functions.

```
-- Composition with π1
theorem toProdp1 {A B C : Type} (f : C → A) (g : C → B) : π1 ∘ (toProd f g) = f := by
  funext c
  exact rfl

-- Composition with π2
theorem toProdp2 {A B C : Type} (f : C → A) (g : C → B) : π2 ∘ (toProd f g) = g := by
  funext c
  exact rfl
```

The universal property of the product type not only guarantees the **existence** of the function `toProd f g : C → A × B` satisfying the projection identities, but also ensures its **uniqueness**. That is, if we have a function `h : C → A × B` such that $\pi_1 \circ h = f$ and $\pi_2 \circ h = g$, then `h` must be equal to `toProd f g`. This uniqueness clause completes the universal property: it tells us that `toProd f g` is the *only* function from `C` to `A × B` whose projections are `f` and `g`. This powerful principle often allows us to characterize and prove properties about functions into product types by reasoning solely about their projections.

```
theorem toProdUnique {A B C : Type} {f : C → A} {g : C → B} {h : C → A × B} :
( π1 ∘ h = f ) → ( π2 ∘ h = g ) → ( h = toProd f g ) := by
  intro h1 h2
  funext c
  apply Prod.ext
  exact congrFun h1 c
  exact congrFun h2 c
```

## 13.2 Generalized product type

Given an index type `I` and a family of types `𝔸 : I → Type`, the product type consists of a collection of values, each corresponding to a type in the family `𝔸 i` for every index `i : I`. In Lean, this is expressed as the dependent function type `∀ (i : I), 𝔸 i`, which can be thought of as the type of functions that assign a value of type `𝔸 i` to each index `i : I`.

```
variable ( I : Type )
variable ( 𝔸 : I → Type )
#check ∀ (i : I), 𝔸 i
```

To produce a value of type `∀ (i : I), 𝔸 i`, we must provide a value of type `𝔸 i` for each `i : I`.

```
def toPairg { I : Type } { 𝔸 : I → Type } : ( (i : I) → 𝔸 i ) → ∀ (i : I), 𝔸 i := by
  intro 𝕗 i
  exact 𝕗 i
```

The type `∀ (i : I), 𝔸 i` has a natural projection that allows us to extract the value of type `𝔸 i` for a specific index `i : I`. This projection is a function that, given an element of type `∀ (i : I), 𝔸 i`, returns the corresponding value of type `𝔸 i` for a particular index `i`.

```
def π { I : Type } { 𝔸 : I → Type } ( i : I ) : ( ∀ (i : I), 𝔸 i ) → 𝔸 i := by
  intro 𝕒
  exact 𝕒 i
```

Two values of type `∀ (i : I), 𝔸 i` are equal if and only if their corresponding `i`-th components are equal for every index `i : I`.

```
theorem prodEqg { I : Type } { 𝔸 : I → Type } ( 𝕒1 𝕒2 : ∀ (i : I), 𝔸 i ) : ( 𝕒1 = 𝕒2 ) ↔ ∀ (i : I), π i 𝕒1 = π i 𝕒2 := by
  apply Iff.intro
  -- 𝕒1 = 𝕒2 → ∀ (i : I), π i 𝕒1 = π i 𝕒2
  intro h i
  exact congrArg ( π i ) h
  -- ( ∀ (i : I), π i 𝕒1 = π i 𝕒2 ) → 𝕒1 = 𝕒2
  intro h
  funext i
  exact h i
```

### 13.2.1 Universal property of the generalized product

The universal property of the product type `∀ (i : I), 𝔸 i` is a key characteristic that allows us to construct a function from a given family of functions. Specifically, if we have a family of functions `𝕗 i : C → 𝔸 i` for each `i : I`, the universal property guarantees the existence of a unique function `h : C → ∀ (i : I), 𝔸 i` such that for every `i : I`, `π i ∘ h = 𝕗 i`.

```
def toProdg { I C : Type } { 𝔸 : I → Type } ( 𝕗 : (i : I) → C → 𝔸 i ) : C → ( ∀ (i : I), 𝔸 i ) := by
  intro c i
  exact ( 𝕗 i ) c
```

Applying the projection `π i` to the result of `toProdg 𝕗` yields the corresponding function `𝕗 i`.

```
theorem toProdpg { I C : Type } { 𝔸 : I → Type } ( 𝕗 : (i : I) → C → 𝔸 i ) ( i : I ) : ( π i ) ∘ ( toProdg 𝕗 ) = 𝕗 i := by
  funext c
  exact rfl
```

The universal property of the product type also asserts a uniqueness condition: if we have a function `h : C → ∀ (i : I), 𝔸 i` such that, for every `i : I`, the composition of `h` with the projection `π i` satisfies the equality `π i ∘ h = 𝕗 i`, then the function `h` must be equal to the function constructed by `toProdg 𝕗`.

```
theorem toProdgUnique { I C : Type } { 𝔸 : I → Type } { 𝕗 : (i : I) → C → 𝔸 i } { h : C → ∀ (i : I), 𝔸 i } : ( ∀ (i : I), ( π i ) ∘ h = ( 𝕗 i ) ) → ( h = toProdg 𝕗 ) := by
  intro hp
  funext c
  funext i
  exact congrFun (hp i) c
```

## 13.3 Sum type

A **sum type** combines two types into a single type whose values are drawn from some component. In Lean, the sum type of `A` and `B` is written as `Sum A B` or, alternatively as, `A ⊕ B`, using the `⊕` symbol (typed as `\oplus`).

If we `#print Sum`, Lean returns:

```
inductive Sum.{u, v} : Type u → Type v → Type (max u v)
number of parameters: 2
constructors:
Sum.inl : { α : Type u } → { β : Type v } → α → α ⊕ β
Sum.inr : { α : Type u } → { β : Type v } → β → α ⊕ β
```

Lean defines the sum type internally using the inductive `Sum` type. This definition shows that `Sum` takes two types—one from universe `u` and one from universe `v`—and returns a type in the larger of the two universes, `max u v`. It includes two constructors: `Sum.inl`, which wraps a value of type `α`, and `Sum.inr`, which wraps a value of type `β`. The type `Sum` ensures that values can be of one type or the other.

The `Sum` type provides two **injections**, `ι1` and `ι2` or, alternatively, `Sum.inl` and `Sum.inr`, which allows us to insert the individual components on a sum. Given a value `a : A`, the expression `Sum.inl a` retrieves an element of type `A ⊕ B`. Given a value `b : B`, the expression `Sum.inr b` retrieves an element of type `A ⊕ B`.

```
-- injection on the first component
def ι1 {A B : Type} : A → A ⊕ B := by
  intro a
  exact Sum.inl a

-- injection on the second component
def ι2 {A B : Type} : B → A ⊕ B := by
  intro b
  exact Sum.inr b
```

Two values of type `A ⊕ B` are considered equal if and only if they are both injections on the same element. This means that if we have two elements `p1, p2 : A ⊕ B`, we say `p1 = p2` if, for both elements, either both are wrapped using the `Sum.inl` constructor (i.e., both come from type `A`), or both are wrapped using the `Sum.inr` constructor (i.e., both come from type `B`), and, in each case, the underlying values are equal.

```
theorem sumEq {A B : Type} (p1 p2 : A ⊕ B) : (p1 = p2) ↔ ( ∃ (a : A), ( ι1 a = p1 ) ∧ ( ι1 a = p2 ) ) ∨
    ∃( (b : B), ( ( ι2 b = p1 ) ∧ ( ι2 b = p2 ) ) := by
  apply Iff.intro
  -- p1 = p2 → ( ∃ a, ι1 a = p1 ∧ ι1 a = p2 ) ∨ ( ∃ b, ι2 b = p1 ∧ ι2 b = p2 )
  intro h
  cases p1 with
  | inl a => cases p2 with
    | inl b =>
      injection h with h1
      apply Or.inl
      apply Exists.intro a
      apply And.intro rfl
      rw [h1]
      exact rfl
    | inr b =>
      exact Sum.noConfusion h
  | inr a => cases p2 with
    | inl b =>
      exact Sum.noConfusion h
    | inr b =>
      injection h with h1
      apply Or.inr
      apply Exists.intro a
      apply And.intro rfl
      rw [h1]
      exact rfl
  -- ( ∃ a, ι1 a = p1 ∧ ι1 a = p2 ) ∨ ( ∃ b, ι2 b = p1 ∧ ι2 b = p2 ) → p1 = p2
  intro h
  cases h with
  | inl h =>
    apply Exists.elim h
    intro a ⟨ h1, h2 ⟩
    exact h1.symm.trans h2
  | inr h =>
    apply Exists.elim h
    intro b ⟨ h1, h2 ⟩
    exact h1.symm.trans h2
```

### 13.3.1 Universal property of the sum

The **universal property of the sum type** characterizes it as the *best* type that supports pairing of data. Specifically, given types `A`, `B`, and `C`, and functions `f : A → C` and `g : B → C`, there exists a unique function `h : A ⊕ B → C` such that `h` , when composed with the injections, recover `f` and `g`; that is, `h ∘ ι1 = f` and `h ∘ ι2 = g`. In Lean, this function `h` is constructed by sending `o : A ⊕ B` to `f o` or to `g o` depending on its nature.

```
def fromSum {A B C : Type} (f : A → C) (g : B → C) : (A ⊕ B) → C := by
  intro o
  cases o with
  | inl a => exact f a
  | inr b => exact g b
```

This function has the key property that composing it with the sum injections recovers the original functions.

```
-- Composition with ι1
theorem fromSumi1 {A B C : Type} (f : A → C) (g : B → C) : (fromSum f g) ∘ ι1 = f := by
  funext a
  exact rfl

-- Composition with ι2
theorem fromSumi2 {A B C : Type} (f : A → C) (g : B → C) : (fromSum f g) ∘ ι2 = g := by
  funext b
  exact rfl
```

The universal property of the sum type not only guarantees the **existence** of the function `fromSum f g : A ⊕ B → C` satisfying the injection identities, but also ensures its **uniqueness**. That is, if we have a function `h : A ⊕ B → C` such that `h ∘ ι1 = f` and `h ∘ ι2 = g`, then `h` must be equal to `fromSum f g`. This uniqueness clause completes the universal property: it tells us that `fromSum f g` is the *only* function from `A ⊕ B` to `C` whose injections are `f` and `g`. This powerful principle often allows us to characterize and prove properties about functions from sum types by reasoning solely about their injections.

```
theorem fromSumUnique {A B C : Type} {f : A → C} {g : B → C} {h : (A ⊕ B) → C} : (h ∘ ι1 = f) → (h ∘ ι2
    = g) → (h = fromSum f g) := by
  intro h1 h2
  funext o
  cases o with
  | inl a => exact congrFun h1 a
  | inr b => exact congrFun h2 b
```

## 13.4 Generalized sum type

Given an index type `I` and a family of types `𝔸 : I → Type`, the sum type consists of a collection of values, corresponding to some type in the family `𝔸 i` for some index `i : I`. In Lean, this is expressed as the type `Sigma 𝔸` or, alternatively, `(Σ (i : I), 𝔸 i)`, which can be thought of as the type of functions that assign a value of type `𝔸 i` to some index `i : I`.

```
variable ( I : Type )
variable ( 𝔸 : I → Type )
#check ( Σ (i : I), 𝔸 i )
```

The type `(Σ (i : I), 𝔸 i)` has a natural injection that allows us to insert the value of type `𝔸 i` for an specific index `i : I`. This injection is a function that, given an index `i : I` and an element of type `𝔸 i`, returns an element of type `(Σ (i : I), 𝔸 i)`.

```
def ι { I : Type } { 𝔸 : I → Type } (i : I) : 𝔸 i → ( Σ (i : I), 𝔸 i ) := by
  intro a
  exact ⟨ i, a ⟩
```

Two values of type `(Σ (i : I), 𝔸 i)` are equal if and only if they are injected from the same index `i : I` on the same element. For this we will use `Sigma.ext` keyword.

```
theorem sumEqg { I : Type } { 𝔸 : I → Type } ( 𝕒1 𝕒2 : ( Σ (i : I), 𝔸 i ) ) : ( 𝕒1 = 𝕒2 ) ↔ ∃ (i : I),
    ∃ (a : 𝔸 i), ( 𝕒1 = ι i a ) ∧ ( 𝕒2 = ι i a ) := by
```

```
  apply Iff.intro
  -- 𝕒1 = 𝕒2 → ∃ i a, 𝕒1 = ι i a ∧ 𝕒2 = ι i a
  intro h
  cases 𝕒1 with
  | mk i a => cases 𝕒2 with
    | mk j b =>
      injection h with h1 h2
      apply Exists.intro i
      apply Exists.intro a
      apply And.intro rfl
      exact Sigma.ext h1.symm h2.symm
  -- ∃ i a, 𝕒1 = ι i a ∧ 𝕒2 = ι i a → 𝕒1 = 𝕒2
  intro ⟨ i, ⟨ a, ⟨ h1, h2 ⟩ ⟩ ⟩
  exact  h1.trans h2.symm
```

### 13.4.1 Universal property of the generalized sum

The universal property of the sum type `(Σ (i : I), 𝔸 i)` is a key characteristic that allows us to construct a function from a given family of functions. Specifically, if we have a family of functions `𝕗 i :𝔸 i → C` for each `i : I`, the universal property guarantees the existence of a unique function `h : (Σ (i : I), 𝔸 i) → C` such that for every `i : I`, `h ∘ (ι i) = 𝕗 i`.

```
def fromSumg { I C : Type } { 𝔸 : I → Type } ( 𝕗 : (i : I) → 𝔸 i → C ) : ( Σ (i : I), 𝔸 i ) → C := by
  intro ⟨ i, a ⟩
  exact 𝕗 i a
```

Applying the injection `ι i` and then `fromSumg 𝕗` yields the corresponding function `𝕗 i`.

```
theorem fromSumig { I C : Type } { 𝔸 : I → Type } ( 𝕗 : (i : I) → 𝔸 i → C ) (i : I) : ( fromSumg 𝕗 ) ∘ (
    ι i ) = 𝕗 i := by
  funext c
  exact rfl
```

The universal property of the sum type also asserts a uniqueness condition: if we have a function `h : (Σ (i : I), 𝔸 i) → C` such that, for every `i : I`, the composition of `h` with the injection `ι i` satisfies the equality `h ∘ (ι i) = 𝕗 i`, then the function `h` must be equal to the function constructed by `fromSumg 𝕗`.

```
theorem fromSumgUnique { I C : Type } { 𝔸 : I → Type } { 𝕗 : (i : I) → 𝔸 i → C } { h : ( Σ (i : I), 𝔸 i
    ) → C } : ( ∀ (i : I), h ∘ ( ι i ) = ( 𝕗 i ) ) → ( h = fromSumg 𝕗)  := by
  intro hs
  funext ⟨ i, a ⟩
  exact congrFun (hs i) a
```

## 13.5 Exercises

### 13.5.1 Product

```
-- The product is commutative
theorem prodComm {A B : Type} : (A × B) ≅ (B × A) := by sorry

-- The product is associative
theorem prodAssoc {A B C : Type} : ((A × B) × C) ≅ (A × (B × C)) := by sorry

-- Empty is a left zero
theorem TEmptyProdL {A : Type} : (Empty × A) ≅ Empty := by sorry

-- Empty is a right zero
theorem TEmptyProdR {A : Type} : (A × Empty) ≅ Empty := by sorry

-- Unit is a right unit
theorem TUnitProdR {A : Type} : (A × Unit) ≅ A := by sorry

-- Unit is a left unit
theorem TUnitProdL {A : Type} : (Unit × A) ≅ A := by sorry
```

### 13.5.2 Sum

```
-- The sum is commutative
theorem sumComm {A B : Type} : (A ⊕ B) ≅ (B ⊕ A) := by sorry

-- The sum is associative
theorem sumAssoc {A B C : Type} : ((A ⊕ B) ⊕ C) ≅ (A ⊕ (B ⊕ C)) := by sorry

-- Empty is a left unit
theorem TEmptySumL {A : Type} : (Empty ⊕ A) ≅ A := by sorry

-- Empty is a right unit
theorem TEmptySumR {A : Type} : (A ⊕ Empty) ≅ A := by sorry

-- Product distributes over sum on the right
theorem TProdSumDistR {A B C : Type} : (A × (B ⊕ C)) ≅ ((A × B) ⊕ (A × C)) := by sorry

-- Product distributes over sum on the left
theorem TProdSumDistL {A B C : Type} : ((A ⊕ B) × C) ≅ ((A × C) ⊕ (B × C)) := by sorry
```

# 14 Lists and Monoids

In this chapter, we explore the foundational concept of **monoids** and their deep connection to **lists**, one of the most fundamental data structures in both mathematics and computer science. We begin by examining lists as sequences of elements drawn from a type `α`, highlighting their structure and operations such as concatenation and the empty list. Building on this, we introduce **monoids**—algebraic structures consisting of a list equipped with an associative binary operation and an identity element.

We will see that addition and multiplication over the natural numbers naturally form monoids. We then introduce the **free monoid** over a type `α` and examine its defining properties. A central focus of the chapter is the **universal property** of the free monoid, which characterizes it as the most general monoid generated by a type of elements.

We conclude the theoretical discussion by applying the universal property to define the **length of a list** as a monoid homomorphism into the natural numbers with addition. This example showcases the practical utility of the abstract theory. Finally, the chapter ends with a set of **exercises** designed to reinforce the concepts presented in this chapter.

## 14.1 Lists

In functional programming and formal systems like Lean, a **list** is a fundamental data structure that represents a sequence of elements of a given type.

If we `#print List`, Lean returns:

```
inductive List.{u} : Type u → Type u
number of parameters: 1
constructors:
List.nil : { α : Type u } → List α
List.cons : { α : Type u } → α → List α → List α
```

The `List` type is defined as an **inductive type**. `List` is a **parametric type** that takes one type parameter—say, `α`—and produces the type `List α`, representing lists of elements of type `α`. This definition includes two constructors. The first, `List.nil`, represents the **empty list**, also written `[]`, meaning it can construct an empty list for any type `α`. The second constructor, `List.cons`, builds a nonempty list by taking an element of type `α`, say `x`, and a list of elements of type `α`, say `xs`, returning a new list of type `List α`, `List.cons x xs`, also written `x :: xs`. This new list will have `x` as its **head** and the list `xs` as its **tail**. This construction makes lists easy to process recursively, as each list is either empty or built by adding an element to the front of another list.

For example, in `List N`, the expression `[]` represents the empty list, `z :: []` is a list containing a single element—namely `[z]`, and `z :: s z :: []` constructs a list with two elements, written as `[z, s z]`.

Using the `List.cons` constructor we can define the **concatenation** operation, `List.append`, which takes two lists `l1` and `l2` of type `List α` and returns a new list `List.append l1 l2`, also written `l1 ++ l2`, which appends the two lists together. For example, for the lists `[z, s z]` and `[z]` in `List N`, `[z, s z] ++ [z]` returns the list `[z, s z, z]`.

## 14.2 Monoids

The following Lean code defines the algebraic structure of a **monoid** and **monoid homomorphisms** as `structure` types in Lean.

```
-- A monoid
@[ext] structure Monoid.{u} where
  base : Type u
  mul : base → base → base
  one : base
  assoc : ∀ {a b c : base}, mul a (mul b c) = mul (mul a b) c
```

```
  idl : ∀ {a : base}, mul one a = a
  idr : ∀ {a : base}, mul a one = a

-- A monoid homomorphism
@[ext] structure MonoidHom (M N : Monoid) where
  map : M.base → N.base
  map_mul : ∀ {a b : M.base}, map (M.mul a b) = N.mul (map a) (map b)
  map_one : map M.one = N.one
```

The first structure, `Monoid`, represents a monoid as a type `base` equipped with a binary operation `mul` (interpreted as multiplication), an identity element `one`, and three axioms. The **associativity axiom** (`assoc`) asserts that multiplication is associative: for all elements `a`, `b`, and `c`, we have `mul a (mul b c) = mul (mul a b) c`. The **left identity** (`idl`) and **right identity** (`idr`) laws state that the element `one` behaves as left and right identity for multiplication: `mul one a = a` and `mul a one = a`, respectively. The attribute `@[ext]` enables Lean to automatically generate extensionality lemmas for these structures, making it easier to prove equalities between instances.

The second structure, `MonoidHom`, formalizes **monoid homomorphisms** between two monoids `M` and `N`. A homomorphism consists of a function `map` between the underlying sets of `M` and `N`, which preserves the monoid operations: it satisfies `map (M.mul a b) = N.mul (map a) (map b)` for all `a`, `b`, and also maps the identity element of `M` to that of `N`, i.e., `map M.one = N.one`. Together, these definitions provide a foundation for reasoning formally about monoids and their structure-preserving maps within Lean's type theory framework.

### 14.2.1 Examples of monoids

We present two examples of monoid structures defined over the natural numbers. In the first example, the binary operation is addition, with `0` serving as the identity element. In the second example, the operation is multiplication, and the identity element is `1`. Both structures satisfy the monoid axioms.

```
-- (N, +, 0) is a monoid
def instMonoidNAdd : Monoid where
  base  := N
  mul   := Addition
  one   := z
  assoc := TAddAss.symm
  idl   := TAdd0L
  idr   := TAdd0R

-- (N, *, 1) is a monoid
def instMonoidNMul : Monoid where
  base  := N
  mul   := Multiplication
  one   := one
  assoc := TMultAss.symm
  idl   := TMult1L
  idr   := TMult1R
```

### 14.2.2 The free monoid over a type α

For any type `α`, the free monoid over `α` is given by the type `List α`, equipped with list concatenation (`++`) as the binary operation and the empty list `[]` as the identity element. This structure forms a monoid because concatenation is associative and the empty list acts as a neutral element for concatenation on both sides.

```
-- (List α, ++, []) is a monoid for any type α
def FreeMonoid { α : Type u } : Monoid where
  base := List α
  mul := List.append
  one := []
  assoc := by
    intro a b c
    induction a with
      | nil => simp [List.append]
      | cons x xs ih => simp [List.append, ih]
  idl := by
    intro a
    induction a with
```

```
      | nil => simp [List.append]
      | cons x xs ih => simp [List.append, ih]
  idr := by
    intro a
    induction a with
      | nil => simp [List.append]
      | cons x xs ih => simp [List.append, ih]
```

### 14.2.3 The universal property of the free monoid

The following Lean code defines the canonical **insertion of generators** function `η` from a type `α` into `List α`.

```
-- Insertion of generators
def η { α : Type u } : α → (@FreeMonoid α).base := by
  intro a
  exact List.cons a []
```

The function `η` takes an element `a : α` and returns the singleton list `[a]`, implemented here as `List.cons a []`. This reflects the standard way of embedding generators into a free monoid: each element of `α` is mapped to a list containing just that element.

The **universal property** of the `FreeMonoid α` states that for any monoid `M` and any function `f : α → M.base`, there exists a **unique monoid homomorphism** `Lift f : FreeMonoid α → M` such that `Lift f ∘ η = f`. This means that `Lift f` extends `f` in a way that respects the monoid structure, making `FreeMonoid α` the most general monoid generated freely by the elements of `α`.

The definition of `Lift f` is defined recursively on lists, as follows:

```
def Lift { α : Type u } {M : Monoid} (f : α → M.base) : (@FreeMonoid α).base → M.base := by
  intro xs
  cases xs with
    | nil => exact M.one
    | cons x xs => exact M.mul (f x) (Lift f xs)
```

The base case corresponds to the empty list: `Lift f [] = M.one`, ensuring the identity element of the monoid is preserved. For non-empty lists, `Lift f` applies the homomorphism recursively to the tail of the list and then combines it with the image of the head element using the monoid multiplication `M.mul`. Specifically, for a list `x::xs'`, we have `Lift f (x::xs) = M.mul (f x) (Lift f xs)`.

This construction guarantees that `Lift f` is a monoid homomorphism, mapping the empty list to the identity element and preserving the monoid operation, as we can prove below.

```
-- The function Lift f is a monoid homomorphism from the free monoid to the monoid M
def LiftMonoidHom { α : Type u } {M : Monoid} (f : α → M.base) : MonoidHom (@FreeMonoid α) M where
  map := Lift f
  map_mul := by
    intro a b
    induction a with
      | nil => calc
        Lift f (FreeMonoid.mul [] b) = Lift f b                    := rfl
        _                            = M.mul (M.one) (Lift f b)     := M.idl.symm
        _                            = M.mul (Lift f []) (Lift f b) := congrArg (fun x => M.mul x (Lift f b)) rfl
      | cons x xs ih => calc
        Lift f (FreeMonoid.mul (x::xs) b) = Lift f (x :: (FreeMonoid.mul xs b))         := rfl
        _                                 = M.mul (f x) (Lift f (FreeMonoid.mul xs b))  := rfl
        _                                 = M.mul (f x) (M.mul (Lift f xs) (Lift f b))  := congrArg (fun y => M.mul (f x) y) ih
        _                                 = M.mul (M.mul (f x) (Lift f xs)) (Lift f b)  := M.assoc
        _                                 = M.mul (Lift f (x::xs)) (Lift f b)           := congrArg (fun y => M.mul y (Lift f b)) rfl
  map_one := rfl
```

Furthermore, `Lift f` extends `f` in the sense that for each element `a : α`, it satisfies `Lift f (η a) = f a`, where `η` is the insertion map that sends `a` to the singleton list `[a]`. This is proven in the theorem below.

```
theorem LiftEta { α : Type u } {M : Monoid} (f : α → M.base) : Lift f ∘ η = f := by
  funext a
  calc
```

```
  Lift f η( a) = Lift f (a::[])          := rfl
  _            = M.mul (f a) (Lift f [])  := rfl
  _            = M.mul (f a) M.one        := congrArg (fun x => M.mul (f a) x) rfl
  _            = f a                      := M.idr
```

Finally, we can prove that the function `Lift f` is the unique monoid homomorphism from the free monoid `FreeMonoid α` to any monoid `M` that satisfies the property `Lift f ∘ η = f`. This is proven in the theorem below.

```
theorem LiftUnique { α : Type u } {M : Monoid} (f : α → M.base) (g : MonoidHom (@FreeMonoid α) M) : g.map ∘ η = f → g = LiftMonoidHom f := by
  intro h
  apply MonoidHom.ext
  funext a
  induction a with
    | nil => calc
      g.map []  = M.one := g.map_one
      _         = (LiftMonoidHom f).map [] := (LiftMonoidHom f).map_one
    | cons x xs ih => calc
      g.map (x::xs) = g.map (FreeMonoid.mul ( η x ) xs) := rfl
      _ = M.mul (g.map ( η x )) (g.map xs) := g.map_mul
      _ = M.mul (f x) (g.map xs)  := congrArg (fun y => M.mul y (g.map xs)) (congrFun h x)
      _ = M.mul ((LiftMonoidHom f).map ( η x )) (g.map xs) := congrArg (fun y => M.mul y (g.map xs)) ( congrFun (LiftEta f).symm x)
      _ = M.mul ((LiftMonoidHom f).map ( η x) ) ((LiftMonoidHom f).map xs) := congrArg (fun y => M.mul ((LiftMonoidHom f).map ( η x )) y) ih
      _ = (LiftMonoidHom f).map (FreeMonoid.mul ( η x ) xs) := (LiftMonoidHom f).map_mul.symm
      _ = (LiftMonoidHom f).map (x::xs) := rfl
```

### 14.2.4 The length of a list

As an application of the universal property of the free monoid, we define a function `Length` that computes the length of a list. First, we define `Len : α → N`, a function that maps each element of type `α` to the natural number `1`, representing the fact that each element in a list contributes exactly one to the length.

```
def Len : α → N := by
  intro _
  exact one
```

Using the universal property of the free monoid, we extend `Len` to a monoid homomorphism `Length : (List α,++,[]) → (N,+,0)` by applying the `Lift` function.

```
def Length { α : Type u } : (@FreeMonoid α).base → instMonoidNAdd.base := Lift Len
```

This guarantees that `Length` satisfies the required monoid homomorphism properties: it maps the empty list to `0` (the identity in `(N,+,0)`), and for any two lists, the length of their concatenation is the sum of their individual lengths. Thus, `Length` is a monoid homomorphism that respects the structure of the free monoid and computes the number of elements in a list. This example highlights how the universal property of the free monoid enables the definition of homomorphisms that extend functions from the generators to any target monoid.

## 14.3 Exercises

```
-- The definition of Monoid Isomorphism
@[ext] structure MonoidIso (M N : Monoid) extends (MonoidHom M N) where
  iso : isomorphism map

-- Prove that the monoids (N,+,0) and (List Unit,++,[]) are isomorphic
def NFreeMonoidIso : MonoidIso (@FreeMonoid Unit) instMonoidNAdd where
  sorry
```

# Bibliography


[1] Lean 4 documentation. https://lean-lang.org/lean4/doc/, 2025. Accessed: 2025-08-27; comprehensive manual covering installation, language manual, reference manual, FAQs, development guide, and more.

[2] The lean language reference. https://lean-lang.org/doc/reference/latest/, 2025. Accessed: 2025-08-27; covers Lean version 4.23.0-rc2.

[3] Lean prover community zulip chat. https://leanprover.zulipchat.com/, 2025. Accessed: 2025-08-27; primary community discussion hub for Lean users, browsable without registering.

[4] Learning lean 4. https://leanprover-community.github.io/learn.html, 2025. Accessed: 2025-08-27; introduction to learning resources, tutorials, books, metaprogramming guides, and more.

[5] Mathematics in lean. https://leanprover-community.github.io/mathematics_in_lean/index.html, 2025. Accessed: 2025-08-27.

[6] J. Avigad, L. de Moura, S. Kong, and S. Ullrich. Theorem proving in lean 4. https://lean-lang.org/theorem_proving_in_lean4/, 2025. Accessed: 2025-08-27; based on version v4.21.0.

[7] D. T. Christiansen. Functional programming in lean. https://lean-lang.org/functional_programming_in_lean/, 2025. Accessed: 2025-08-27; code samples tested with Lean 4.21.0; Copyright Microsoft Corporation 2023 and Lean FRO, LLC 20232025.

[8] D. Clemente Laboreo. Introduction to natural deduction. https://www.danielclemente.com/logica/dn.en.pdf, 2004. August 2004; reviewed May 2005; accessed 2025-08-27.

[9] J. Climent Vidal. Teoría de conjuntos. https://www.uv.es/jkliment/Documentos/SetTheory.pc.pdf, 2010. Date: 25 de junio de 2010; accessed: 2025-08-27; comprehensive lecture notes on Zermelo-Fraenkel set theory.

[10] L. de Moura and S. Ullrich. The Lean 4 theorem prover and programming language. In A. Platzer and G. Sutcliffe, editors, *Automated Deduction CADE 28*, volume 12699 of *Lecture Notes in Artificial Intelligence*, pages 625–635. Springer, 2021.

[11] H. Macbeth. The mechanics of proof. https://hrmacbeth.github.io/math2001/, 2025. Accessed: 2025-08-27; course Math 2001, Fordham University; Lean code available at GitHub.

[12] P. Smith. Introducing category theory. https://www.logicmatters.net/resources/pdfs/SmithCat.pdf, 2025. Version 2.9, Ï 2025; second edition; PDF available for educational use; print-on-demand from June 2025; accessed 2025-08-27.

[13] R. Zach. Boxes and diamonds: An open introduction to modal logic. https://bd.openlogicproject.org/, 2025. Accessed: 2025-08-27; based on the Open Logic Project; licensed under CC BY 4.0.