Theorem Proving in Lean 4

by Jeremy Avigad, Leonardo de Moura, Soonho Kong and Sebastian Ullrich, with contributions from the Lean Community

This version of the text assumes you're using Lean 4. See the Quickstart section of the Lean 4 Manual to install Lean. The first version of this book was written for Lean 2, and the Lean 3 version is available here.

Introduction

Computers and Theorem Proving

Formal verification involves the use of logical and computational methods to establish claims that are expressed in precise mathematical terms. These can include ordinary mathematical theorems, as well as claims that pieces of hardware or software, network protocols, and mechanical and hybrid systems meet their specifications. In practice, there is not a sharp distinction between verifying a piece of mathematics and verifying the correctness of a system: formal verification requires describing hardware and software systems in mathematical terms, at which point establishing claims as to their correctness becomes a form of theorem proving. Conversely, the proof of a mathematical theorem may require a lengthy computation, in which case verifying the truth of the theorem requires verifying that the computation does what it is supposed to do.

The gold standard for supporting a mathematical claim is to provide a proof, and twentieth-century developments in logic show most if not all conventional proof methods can be reduced to a small set of axioms and rules in any of a number of foundational systems. With this reduction, there are two ways that a computer can help establish a claim: it can help find a proof in the first place, and it can help verify that a purported proof is correct.

Automated theorem proving focuses on the "finding" aspect. Resolution theorem provers, tableau theorem provers, fast satisfiability solvers, and so on provide means of establishing the validity of formulas in propositional and first-order logic. Other systems provide search procedures and decision procedures for specific languages and domains, such as linear or nonlinear expressions over the integers or the real numbers. Architectures like SMT ("satisfiability modulo theories") combine domain-general search methods with domain-specific procedures. Computer algebra systems and specialized mathematical software packages provide means of carrying out mathematical computations, establishing mathematical bounds, or finding mathematical objects. A calculation can be viewed as a proof as well, and these systems, too, help establish mathematical claims.

Automated reasoning systems strive for power and efficiency, often at the expense of guaranteed soundness. Such systems can have bugs, and it can be difficult to ensure that the results they deliver are correct. In contrast, *interactive theorem proving* focuses on the "verification" aspect of theorem proving, requiring that every claim is supported by a proof in a suitable axiomatic foundation. This sets a very high standard: every rule of inference and every step of a calculation has to be justified by appealing to prior definitions and theorems, all the way down to basic axioms and rules. In fact, most such systems provide fully elaborated "proof objects" that can be communicated to other systems and checked independently. Constructing such proofs typically requires much more input and interaction from users, but it allows you to obtain deeper and more complex proofs.

The *Lean Theorem Prover* aims to bridge the gap between interactive and automated theorem proving, by situating automated tools and methods in a framework that supports user interaction and the construction of fully specified axiomatic proofs. The goal is to support both mathematical reasoning and reasoning about complex systems, and to verify claims in both domains.

Lean's underlying logic has a computational interpretation, and Lean can be viewed equally well as a programming language. More to the point, it can be viewed as a system for writing programs with a precise semantics, as well as reasoning about the functions that the programs compute. Lean also has mechanisms to serve as its own *metaprogramming language*, which means that you can implement automation and extend the functionality of Lean using Lean itself. These aspects of Lean are described in the free online book, Functional Programming in Lean, though computational aspects of the system will make an appearance here.

About Lean

The *Lean* project was launched by Leonardo de Moura at Microsoft Research Redmond in 2013. It is an ongoing, long-term effort, and much of the potential for automation will be realized only gradually over time. Lean is released under the Apache 2.0 license, a permissive open source license that permits others to use and extend the code and mathematical libraries freely.

To install Lean in your computer consider using the Quickstart instructions. The Lean source code, and instructions for building Lean, are available at https://github.com/leanprover/lean4/.

This tutorial describes the current version of Lean, known as Lean 4.

About this Book

This book is designed to teach you to develop and verify proofs in Lean. Much of the background information you will need in order to do this is not specific to Lean at all. To start with, you will learn the logical system that Lean is based on, a version of *dependent type theory* that is powerful enough to prove almost any conventional mathematical theorem, and expressive enough to do it in a natural way. More specifically, Lean is based on a version of a system known as the Calculus of Constructions with inductive types. Lean can not only define mathematical objects and express mathematical assertions in dependent type theory, but it also can be used as a language for writing proofs.

Because fully detailed axiomatic proofs are so complicated, the challenge of theorem proving is to have the computer fill in as many of the details as possible. You will learn various methods to support this in dependent type theory. For example, term rewriting, and Lean's automated methods for simplifying terms and expressions automatically. Similarly, methods of *elaboration* and *type inference*, which can be used to support flexible forms of algebraic reasoning.

Finally, you will learn about features that are specific to Lean, including the language you use to communicate with the system, and the mechanisms Lean offers for managing complex theories and data.

Throughout the text you will find examples of Lean code like the one below:

```
theorem and_commutative (p q : Prop) : p ∧ q → q ∧ p :=
  fun hpq : p ∧ q =>
  have hp : p := And.left hpq
  have hq : q := And.right hpq
  show q ∧ p from And.intro hq hp
```

If you are reading the book inside of VS Code, you will see a button that reads "try it!" Pressing the button copies the example to your editor with enough surrounding context to make the code compile correctly. You can type things into the editor and modify the examples, and Lean will check the results and provide feedback continuously as you type. We recommend running the examples and experimenting with the code on your own as you work through the chapters that follow. You can open this book on VS Code by using the command "Lean 4: Open Documentation View".

Acknowledgments

This tutorial is an open access project maintained on Github. Many people have contributed to the effort, providing corrections, suggestions, examples, and text. We are grateful to Ulrik Buchholz, Kevin Buzzard, Mario Carneiro, Nathan Carter, Eduardo Cavazos, Amine Chaieb, Joe Corneli, William DeMeo, Marcus Klaas de Vries, Ben Dyer, Gabriel Ebner, Anthony Hart, Simon Hudon, Sean Leather, Assia Mahboubi, Gihan Marasingha, Patrick Massot,

Christopher John Mazey, Sebastian Ullrich, Floris van Doorn, Daniel Velleman, Théo Zimmerman, Paul Chisholm, Chris Lovett, and Siddhartha Gadgil for their contributions. Please see lean prover and lean community for an up to date list of our amazing contributors.

Dependent Type Theory

Dependent type theory is a powerful and expressive language, allowing you to express complex mathematical assertions, write complex hardware and software specifications, and reason about both of these in a natural and uniform way. Lean is based on a version of dependent type theory known as the *Calculus of Constructions*, with a countable hierarchy of non-cumulative universes and inductive types. By the end of this chapter, you will understand much of what this means.

Simple Type Theory

"Type theory" gets its name from the fact that every expression has an associated *type*. For example, in a given context, `x + 0` may denote a natural number and `f` may denote a function on the natural numbers. For those who like precise definitions, a Lean natural number is an arbitrary-precision unsigned integer.

Here are some examples of how you can declare objects in Lean and check their types.

```
/- Define some constants. -/

def m : Nat := 1       -- m is a natural number
def n : Nat := 0
def b1 : Bool := true  -- b1 is a Boolean
def b2 : Bool := false

/- Check their types. -/

#check m            -- output: Nat
#check n
#check n + 0        -- Nat
#check m * (n + 0)  -- Nat
#check b1           -- Bool
#check b1 && b2     -- "&&" is the Boolean and
#check b1 || b2     -- Boolean or
#check true         -- Boolean "true"

/- Evaluate -/

#eval 5 * 4         -- 20
#eval m + 2         -- 3
#eval b1 && b2      -- false
```

Any text between `/-` and `-/` constitutes a comment block that is ignored by Lean. Similarly, two dashes `--` indicate that the rest of the line contains a comment that is also ignored. Comment blocks can be nested, making it possible to "comment out" chunks of code, just as in many programming languages.

The `def` keyword declares new constant symbols into the working environment. In the example above, `def m : Nat := 1` defines a new constant `m` of type `Nat` whose value is `1`. The `#check` command asks Lean to report their types; in Lean, auxiliary commands that query the system for information typically begin with the hash (#) symbol. The `#eval` command asks Lean to evaluate the given expression. You should try declaring some constants and type checking some expressions on your own. Declaring new objects in this manner is a good way to experiment with the system.

What makes simple type theory powerful is that you can build new types out of others. For example, if `a` and `b` are types, `a -> b` denotes the type of functions from `a` to `b`, and `a × b` denotes the type of pairs consisting of an element of `a` paired with an element of `b`, also known as the *Cartesian product*. Note that `×` is a Unicode symbol. The judicious use of Unicode improves legibility, and all modern editors have great support for it. In the Lean standard library, you often see Greek letters to denote types, and the Unicode symbol `→` as a more compact version of `->`.

```
#check Nat → Nat      -- type the arrow as "\to" or "\r"
#check Nat -> Nat     -- alternative ASCII notation

#check Nat × Nat      -- type the product as "\times"
#check Prod Nat Nat   -- alternative notation

#check Nat → Nat → Nat
#check Nat → (Nat → Nat)  --  same type as above

#check Nat × Nat → Nat
#check (Nat → Nat) → Nat -- a "functional"

#check Nat.succ     -- Nat → Nat
#check (0, 1)       -- Nat × Nat
#check Nat.add      -- Nat → Nat → Nat

#check Nat.succ 2   -- Nat
#check Nat.add 3    -- Nat → Nat
#check Nat.add 5 2  -- Nat
#check (5, 9).1     -- Nat
#check (5, 9).2     -- Nat

#eval Nat.succ 2   -- 3
#eval Nat.add 5 2  -- 7
#eval (5, 9).1     -- 5
#eval (5, 9).2     -- 9
```

Once again, you should try some examples on your own.

Let's take a look at some basic syntax. You can enter the unicode arrow `→` by typing `\to` or `\r` or `\->`. You can also use the ASCII alternative `->`, so the expressions `Nat -> Nat` and `Nat → Nat` mean the same thing. Both expressions denote the type of functions that take a natural number as input and return a natural number as output. The unicode symbol `×` for the Cartesian product is entered as `\times`. You will generally use lower-case Greek letters like `α`, `β`, and `γ` to range over types. You can enter these particular ones with `\a`, `\b`, and `\g`.

There are a few more things to notice here. First, the application of a function `f` to a value `x` is denoted `f x` (e.g., `Nat.succ 2`). Second, when writing type expressions, arrows associate to the *right*; for example, the type of `Nat.add` is `Nat → Nat → Nat` which is equivalent to `Nat → (Nat → Nat)`. Thus you can view `Nat.add` as a function that takes a natural number and returns another function that takes a natural number and returns a natural number. In type theory, this is generally more convenient than writing `Nat.add` as a function that takes a pair of natural numbers as input and returns a natural number as output. For example, it allows you to "partially apply" the function `Nat.add`. The example above shows that `Nat.add 3` has type `Nat → Nat`, that is, `Nat.add 3` returns a function that "waits" for a second argument, `n`, which is then equivalent to writing `Nat.add 3 n`.

You have seen that if you have `m : Nat` and `n : Nat`, then `(m, n)` denotes the ordered pair of `m` and `n` which is of type `Nat × Nat`. This gives you a way of creating pairs of natural numbers. Conversely, if you have `p : Nat × Nat`, then you can write `p.1 : Nat` and `p.2 : Nat`. This gives you a way of extracting its two components.

Types as objects

One way in which Lean's dependent type theory extends simple type theory is that types themselves --- entities like `Nat` and `Bool` --- are first-class citizens, which is to say that they themselves are objects. For that to be the case, each of them also has to have a type.

```
#check Nat                -- Type
#check Bool               -- Type
#check Nat → Bool         -- Type
#check Nat × Bool         -- Type
#check Nat → Nat          -- ...
#check Nat × Nat → Nat
#check Nat → Nat → Nat
#check Nat → (Nat → Nat)
#check Nat → Nat → Bool
#check (Nat → Nat) → Nat
```

You can see that each one of the expressions above is an object of type `Type`. You can also declare new constants for types:

```
def α : Type := Nat
def β : Type := Bool
def F : Type → Type := List
def G : Type → Type → Type := Prod

#check α         -- Type
#check F α       -- Type
#check F Nat     -- Type
#check G α       -- Type → Type
#check G α β     -- Type
#check G α Nat   -- Type
```

As the example above suggests, you have already seen an example of a function of type `Type → Type → Type`, namely, the Cartesian product `Prod`:

```
def α : Type := Nat
def β : Type := Bool

#check Prod α β       -- Type
#check α × β          -- Type

#check Prod Nat Nat   -- Type
#check Nat × Nat      -- Type
```

Here is another example: given any type `α`, the type `List α` denotes the type of lists of elements of type `α`.

```
def α : Type := Nat

#check List α    -- Type
#check List Nat  -- Type
```

Given that every expression in Lean has a type, it is natural to ask: what type does `Type` itself have?

```
#check Type      -- Type 1
```

You have actually come up against one of the most subtle aspects of Lean's typing system. Lean's underlying foundation has an infinite hierarchy of types:

```
#check Type     -- Type 1
#check Type 1   -- Type 2
#check Type 2   -- Type 3
#check Type 3   -- Type 4
#check Type 4   -- Type 5
```

Think of `Type 0` as a universe of "small" or "ordinary" types. `Type 1` is then a larger universe of types, which contains `Type 0` as an element, and `Type 2` is an even larger universe of types, which contains `Type 1` as an element. The list is infinite: there is a `Type n` for every natural number `n`. `Type` is an abbreviation for `Type 0`:

```
#check Type
#check Type 0
```

The following table may help concretize the relationships being discussed. Movement along the x-axis represents a change in the universe, while movement along the y-axis represents a change in what is sometimes referred to as "degree".

sort	Prop (Sort 0)	Type (Sort 1)	Type 1 (Sort 2)	Type 2 (Sort 3)	...
type	True	Bool	Nat -> Type	Type -> Type 1	...
term	trivial	true	fun n => Fin n	fun (_ : Type) => Type	...

Some operations, however, need to be *polymorphic* over type universes. For example, `List α` should make sense for any type `α`, no matter which type universe `α` lives in. This explains the type signature of the function `List`:

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

Here `u` is a variable ranging over type levels. The output of the `#check` command means that whenever `α` has type `Type n`, `List α` also has type `Type n`. The function `Prod` is similarly polymorphic:

```
#check Prod    -- Prod.{u, v} (α : Type u) (β : Type v) : Type (max u v)
```

To define polymorphic constants, Lean allows you to declare universe variables explicitly using the `universe` command:

```
universe u

def F (α : Type u) : Type u := Prod α α

#check F    -- Type u → Type u
```

You can avoid the `universe` command by providing the universe parameters when defining `F`:

```
def F.{u} (α : Type u) : Type u := Prod α α

#check F    -- Type u → Type u
```

Function Abstraction and Evaluation

Lean provides a `fun` (or `λ`) keyword to create a function from an expression as follows:

```
#check fun (x : Nat) => x + 5   -- Nat → Nat
#check λ (x : Nat) => x + 5     -- λ and fun mean the same thing
#check fun x => x + 5     -- Nat inferred
#check λ x => x + 5       -- Nat inferred
```

You can evaluate a lambda function by passing the required parameters:

```
#eval (λ x : Nat => x + 5) 10    -- 15
```

Creating a function from another expression is a process known as *lambda abstraction*. Suppose you have the variable `x : α` and you can construct an expression `t : β`, then the expression `fun (x : α) => t`, or, equivalently, `λ (x : α) => t`, is an object of type `α → β`. Think of this as the function from `α` to `β` which maps any value `x` to the value `t`.

Here are some more examples

```
#check fun x : Nat => fun y : Bool => if not y then x + 1 else x + 2
#check fun (x : Nat) (y : Bool) => if not y then x + 1 else x + 2
#check fun x y => if not y then x + 1 else x + 2   -- Nat → Bool → Nat
```

Lean interprets the final three examples as the same expression; in the last expression, Lean infers the type of `x` and `y` from the expression `if not y then x + 1 else x + 2`.

Some mathematically common examples of operations of functions can be described in terms of lambda abstraction:

```
def f (n : Nat) : String := toString n
def g (s : String) : Bool := s.length > 0

#check fun x : Nat => x         -- Nat → Nat
#check fun x : Nat => true      -- Nat → Bool
#check fun x : Nat => g (f x)   -- Nat → Bool
#check fun x => g (f x)         -- Nat → Bool
```

Think about what these expressions mean. The expression `fun x : Nat => x` denotes the identity function on `Nat`, the expression `fun x : Nat => true` denotes the constant function that always returns `true`, and `fun x : Nat => g (f x)` denotes the composition of `f` and `g`. You can, in general, leave off the type annotation and let Lean infer it for you. So, for example, you can write `fun x => g (f x)` instead of `fun x : Nat => g (f x)`.

You can pass functions as parameters and by giving them names `f` and `g` you can then use those functions in the implementation:

```
#check fun (g : String → Bool) (f : Nat → String) (x : Nat) => g (f x)
-- (String → Bool) → (Nat → String) → Nat → Bool
```

You can also pass types as parameters:

```
#check fun (α β γ : Type) (g : β → γ) (f : α → β) (x : α) => g (f x)
```

The last expression, for example, denotes the function that takes three types, `α`, `β`, and `γ`, and two functions, `g : β → γ` and `f : α → β`, and returns the composition of `g` and `f`. (Making sense of the type of this function requires an understanding of *dependent products*, which will be explained below.)

The general form of a lambda expression is `fun x : α => t`, where the variable `x` is a "bound variable": it is really a placeholder, whose "scope" does not extend beyond the expression `t`. For example, the variable `b` in the expression `fun (b : β) (x : α) => b` has nothing to do with the constant `b` declared earlier. In fact, the expression denotes the same function as `fun (u : β) (z : α) => u`.

Formally, expressions that are the same up to a renaming of bound variables are called *alpha equivalent*, and are considered "the same." Lean recognizes this equivalence.

Notice that applying a term `t : α → β` to a term `s : α` yields an expression `t s : β`. Returning to the previous example and renaming bound variables for clarity, notice the types of the following expressions:

```
#check (fun x : Nat => x) 1      -- Nat
#check (fun x : Nat => true) 1   -- Bool

def f (n : Nat) : String := toString n
def g (s : String) : Bool := s.length > 0

#check
  (fun (α β γ : Type) (u : β → γ) (v : α → β) (x : α) => u (v x)) Nat String
Bool g f 0
  -- Bool
```

As expected, the expression `(fun x : Nat =>  x) 1` has type `Nat`. In fact, more should be true: applying the expression `(fun x : Nat => x)` to `1` should "return" the value `1`. And, indeed, it does:

```
#eval (fun x : Nat => x) 1      -- 1
#eval (fun x : Nat => true) 1   -- true
```

You will see later how these terms are evaluated. For now, notice that this is an important feature of dependent type theory: every term has a computational behavior, and supports a notion of *normalization*. In principle, two terms that reduce to the same value are called *definitionally equal*. They are considered "the same" by Lean's type checker, and Lean does its best to recognize and support these identifications.

Lean is a complete programming language. It has a compiler that generates a binary executable and an interactive interpreter. You can use the command `#eval` to execute expressions, and it is the preferred way of testing your functions.

Definitions

Recall that the `def` keyword provides one important way of declaring new named objects.

```
def double (x : Nat) : Nat :=
  x + x
```

This might look more familiar to you if you know how functions work in other programming languages. The name `double` is defined as a function that takes an input parameter `x` of type `Nat`, where the result of the call is `x + x`, so it is returning type `Nat`. You can then invoke this function using:

```
#eval double 3    -- 6
```

In this case you can think of `def` as a kind of named `lambda`. The following yields the same result:

```
def double : Nat → Nat :=
  fun x => x + x

#eval double 3    -- 6
```

You can omit the type declarations when Lean has enough information to infer it. Type inference is an important part of Lean:

```
def double :=
  fun (x : Nat) => x + x
```

The general form of a definition is `def foo : α := bar` where `α` is the type returned from the expression `bar`. Lean can usually infer the type `α`, but it is often a good idea to write it explicitly. This clarifies your intention, and Lean will flag an error if the right-hand side of the definition does not have a matching type.

The right hand side `bar` can be any expression, not just a lambda. So `def` can also be used to simply name a value like this:

```
def pi := 3.141592654
```

`def` can take multiple input parameters. Let's create one that adds two natural numbers:

```
def add (x y : Nat) :=
  x + y

#eval add 3 2                -- 5
```

The parameter list can be separated like this:

```
def add (x : Nat) (y : Nat) :=
  x + y

#eval add (double 3) (7 + 9)  -- 22
```

Notice here we called the `double` function to create the first parameter to `add`.

You can use other more interesting expressions inside a `def`:

```
def greater (x y : Nat) :=
  if x > y then x
  else y
```

You can probably guess what this one will do.

You can also define a function that takes another function as input. The following calls a given function twice passing the output of the first invocation to the second:

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

#eval doTwice double 2   -- 8
```

Now to get a bit more abstract, you can also specify arguments that are like type parameters:

```
def compose (α β γ : Type) (g : β → γ) (f : α → β) (x : α) : γ :=
  g (f x)
```

This means `compose` is a function that takes any two functions as input arguments, so long as those functions each take only one input. The type algebra `β → γ` and `α → β` means it is a requirement that the type of the output of the second function must match the type of the input to the first function - which makes sense, otherwise the two functions would not be composable.

`compose` also takes a 3rd argument of type `α` which it uses to invoke the second function (locally named `f`) and it passes the result of that function (which is type `β`) as input to the first function (locally named `g`). The first function returns a type `γ` so that is also the return type of the `compose` function.

`compose` is also very general in that it works over any type `α β γ`. This means `compose` can compose just about any 2 functions so long as they each take one parameter, and so long as the type of output of the second matches the input of the first. For example:

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

#eval compose Nat Nat Nat double square 3  -- 18
```

Local Definitions

Lean also allows you to introduce "local" definitions using the `let` keyword. The expression `let a := t1; t2` is definitionally equal to the result of replacing every occurrence of `a` in `t2` by `t1`.

```
#check let y := 2 + 2; y * y   -- Nat
#eval  let y := 2 + 2; y * y   -- 16

def twice_double (x : Nat) : Nat :=
  let y := x + x; y * y

#eval twice_double 2   -- 16
```

Here, `twice_double x` is definitionally equal to the term `(x + x) * (x + x)`.

You can combine multiple assignments by chaining `let` statements:

```
#check let y := 2 + 2; let z := y + y; z * z   -- Nat
#eval  let y := 2 + 2; let z := y + y; z * z   -- 64
```

The `;` can be omitted when a line break is used.

```
def t (x : Nat) : Nat :=
  let y := x + x
  y * y
```

Notice that the meaning of the expression `let a := t1; t2` is very similar to the meaning of `(fun a => t2) t1`, but the two are not the same. In the first expression, you should think of every instance of `a` in `t2` as a syntactic abbreviation for `t1`. In the second expression, `a` is a variable, and the expression `fun a => t2` has to make sense independently of the value of `a`. The `let` construct is a stronger means of abbreviation, and there are expressions of the form `let a := t1; t2` that cannot be expressed as `(fun a => t2) t1`. As an exercise, try to understand why the definition of `foo` below type checks, but the definition of `bar` does not.

```
def foo := let a := Nat; fun x : a => x + 2
/-
  def bar := (fun a => fun x : a => x + 2) Nat
-/
```

Variables and Sections

Consider the following three function definitions:

```
def compose (α β γ : Type) (g : β → γ) (f : α → β) (x : α) : γ :=
  g (f x)

def doTwice (α : Type) (h : α → α) (x : α) : α :=
  h (h x)

def doThrice (α : Type) (h : α → α) (x : α) : α :=
  h (h (h x))
```

Lean provides you with the `variable` command to make such declarations look more compact:

```
variable (α β γ : Type)

def compose (g : β → γ) (f : α → β) (x : α) : γ :=
  g (f x)

def doTwice (h : α → α) (x : α) : α :=
  h (h x)

def doThrice (h : α → α) (x : α) : α :=
  h (h (h x))
```

You can declare variables of any type, not just `Type` itself:

```
variable (α β γ : Type)
variable (g : β → γ) (f : α → β) (h : α → α)
variable (x : α)

def compose := g (f x)
def doTwice := h (h x)
def doThrice := h (h (h x))

#print compose
#print doTwice
#print doThrice
```

Printing them out shows that all three groups of definitions have exactly the same effect.

The `variable` command instructs Lean to insert the declared variables as bound variables in definitions that refer to them by name. Lean is smart enough to figure out which variables are used explicitly or implicitly in a definition. You can therefore proceed as though `α`, `β`, `γ`, `g`, `f`, `h`, and `x` are fixed objects when you write your definitions, and let Lean abstract the definitions for you automatically.

When declared in this way, a variable stays in scope until the end of the file you are working on. Sometimes, however, it is useful to limit the scope of a variable. For that purpose, Lean provides the notion of a `section`:

```
section useful
  variable (α β γ : Type)
  variable (g : β → γ) (f : α → β) (h : α → α)
  variable (x : α)

  def compose := g (f x)
  def doTwice := h (h x)
  def doThrice := h (h (h x))
end useful
```

When the section is closed, the variables go out of scope, and cannot be referenced any more.

You do not have to indent the lines within a section. Nor do you have to name a section, which is to say, you can use an anonymous `section` / `end` pair. If you do name a section, however, you have to close it using the same name. Sections can also be nested, which allows you to declare new variables incrementally.

Namespaces

Lean provides you with the ability to group definitions into nested, hierarchical *namespaces*:

```
namespace Foo
  def a : Nat := 5
  def f (x : Nat) : Nat := x + 7

  def fa : Nat := f a
  def ffa : Nat := f (f a)

  #check a
  #check f
  #check fa
  #check ffa
  #check Foo.fa
end Foo

-- #check a  -- error
-- #check f  -- error
#check Foo.a
#check Foo.f
#check Foo.fa
#check Foo.ffa

open Foo

#check a
#check f
#check fa
#check Foo.fa
```

When you declare that you are working in the namespace `Foo`, every identifier you declare has a full name with prefix "`Foo.`". Within the namespace, you can refer to identifiers by

their shorter names, but once you end the namespace, you have to use the longer names. Unlike `section`, namespaces require a name. There is only one anonymous namespace at the root level.

The `open` command brings the shorter names into the current context. Often, when you import a module, you will want to open one or more of the namespaces it contains, to have access to the short identifiers. But sometimes you will want to leave this information protected by a fully qualified name, for example, when they conflict with identifiers in another namespace you want to use. Thus namespaces give you a way to manage names in your working environment.

For example, Lean groups definitions and theorems involving lists into a namespace `List`.

```
#check List.nil
#check List.cons
#check List.map
```

The command `open List` allows you to use the shorter names:

```
open List

#check nil
#check cons
#check map
```

Like sections, namespaces can be nested:

```
namespace Foo
  def a : Nat := 5
  def f (x : Nat) : Nat := x + 7

  def fa : Nat := f a

  namespace Bar
    def ffa : Nat := f (f a)

    #check fa
    #check ffa
  end Bar

  #check fa
  #check Bar.ffa
end Foo

#check Foo.fa
#check Foo.Bar.ffa

open Foo

#check fa
#check Bar.ffa
```

Namespaces that have been closed can later be reopened, even in another file:

```
namespace Foo
  def a : Nat := 5
  def f (x : Nat) : Nat := x + 7

  def fa : Nat := f a
end Foo

#check Foo.a
#check Foo.f

namespace Foo
  def ffa : Nat := f (f a)
end Foo
```

Like sections, nested namespaces have to be closed in the order they are opened. Namespaces and sections serve different purposes: namespaces organize data and sections declare variables for insertion in definitions. Sections are also useful for delimiting the scope of commands such as `set_option` and `open`.

In many respects, however, a `namespace ... end` block behaves the same as a `section ... end` block. In particular, if you use the `variable` command within a namespace, its scope is limited to the namespace. Similarly, if you use an `open` command within a namespace, its effects disappear when the namespace is closed.

What makes dependent type theory dependent?

The short explanation is that types can depend on parameters. You have already seen a nice example of this: the type `List α` depends on the argument `α`, and this dependence is what distinguishes `List Nat` and `List Bool`. For another example, consider the type `Vector α n`, the type of vectors of elements of `α` of length `n`. This type depends on *two* parameters: the type of the elements in the vector (`α : Type`) and the length of the vector `n : Nat`.

Suppose you wish to write a function `cons` which inserts a new element at the head of a list. What type should `cons` have? Such a function is *polymorphic*: you expect the `cons` function for `Nat`, `Bool`, or an arbitrary type `α` to behave the same way. So it makes sense to take the type to be the first argument to `cons`, so that for any type, `α`, `cons α` is the insertion function for lists of type `α`. In other words, for every `α`, `cons α` is the function that takes an element `a : α` and a list `as : List α`, and returns a new list, so you have `cons α a as : List α`.

It is clear that `cons α` should have type `α → List α → List α`. But what type should `cons` have? A first guess might be `Type → α → List α → List α`, but, on reflection, this does not make sense: the `α` in this expression does not refer to anything, whereas it should refer to the argument of type `Type`. In other words, *assuming* `α : Type` is the first argument to the

function, the type of the next two elements are `α` and `List α`. These types vary depending on the first argument, `α`.

```
def cons (α : Type) (a : α) (as : List α) : List α :=
  List.cons a as

#check cons Nat        -- Nat → List Nat → List Nat
#check cons Bool       -- Bool → List Bool → List Bool
#check cons            -- (α : Type) → α → List α → List α
```

This is an instance of a *dependent function type*, or *dependent arrow type*. Given `α : Type` and `β : α → Type`, think of `β` as a family of types over `α`, that is, a type `β a` for each `a : α`. In that case, the type `(a : α) → β a` denotes the type of functions `f` with the property that, for each `a : α`, `f a` is an element of `β a`. In other words, the type of the value returned by `f` depends on its input.

Notice that `(a : α) → β` makes sense for any expression `β : Type`. When the value of `β` depends on `a` (as does, for example, the expression `β a` in the previous paragraph), `(a : α) → β` denotes a dependent function type. When `β` doesn't depend on `a`, `(a : α) → β` is no different from the type `α → β`. Indeed, in dependent type theory (and in Lean), `α → β` is just notation for `(a : α) → β` when `β` does not depend on `a`.

Returning to the example of lists, you can use the command `#check` to inspect the type of the following `List` functions. The `@` symbol and the difference between the round and curly braces will be explained momentarily.

```
#check @List.cons    -- {α : Type u_1} → α → List α → List α
#check @List.nil     -- {α : Type u_1} → List α
#check @List.length  -- {α : Type u_1} → List α → Nat
#check @List.append  -- {α : Type u_1} → List α → List α → List α
```

Just as dependent function types `(a : α) → β a` generalize the notion of a function type `α → β` by allowing `β` to depend on `α`, dependent Cartesian product types `(a : α) × β a` generalize the Cartesian product `α × β` in the same way. Dependent products are also called *sigma* types, and you can also write them as `Σ a : α, β a`. You can use `⟨a, b⟩` or `Sigma.mk a b` to create a dependent pair. The `⟨` and `⟩` characters may be typed with `\langle` and `\rangle` or `\<` and `\>`, respectively.

```
universe u v

def f (α : Type u) (β : α → Type v) (a : α) (b : β a) : (a : α) × β a :=
  ⟨a, b⟩

def g (α : Type u) (β : α → Type v) (a : α) (b : β a) : Σ a : α, β a :=
  Sigma.mk a b

def h1 (x : Nat) : Nat :=
  (f Type (fun α => α) Nat x).2

#eval h1 5 -- 5

def h2 (x : Nat) : Nat :=
  (g Type (fun α => α) Nat x).2

#eval h2 5 -- 5
```

The functions `f` and `g` above denote the same function.

Implicit Arguments

Suppose we have an implementation of lists as:

```
#check Lst          -- Lst.{u} (α : Type u) : Type u
#check Lst.cons     -- Lst.cons.{u} (α : Type u) (a : α) (as : Lst α) : Lst α
#check Lst.nil      -- Lst.nil.{u} (α : Type u) : Lst α
#check Lst.append   -- Lst.append.{u} (α : Type u) (as bs : Lst α) : Lst α
```

Then, you can construct lists of `Nat` as follows:

```
#check Lst.cons Nat 0 (Lst.nil Nat)

def as : Lst Nat := Lst.nil Nat
def bs : Lst Nat := Lst.cons Nat 5 (Lst.nil Nat)

#check Lst.append Nat as bs
```

Because the constructors are polymorphic over types, we have to insert the type `Nat` as an argument repeatedly. But this information is redundant: one can infer the argument `α` in `Lst.cons Nat 5 (Lst.nil Nat)` from the fact that the second argument, `5`, has type `Nat`. One can similarly infer the argument in `Lst.nil Nat`, not from anything else in that expression, but from the fact that it is sent as an argument to the function `Lst.cons`, which expects an element of type `Lst α` in that position.

This is a central feature of dependent type theory: terms carry a lot of information, and often some of that information can be inferred from the context. In Lean, one uses an underscore, `_`, to specify that the system should fill in the information automatically. This is known as an "implicit argument."

```
#check Lst.cons _ 0 (Lst.nil _)

def as : Lst Nat := Lst.nil _
def bs : Lst Nat := Lst.cons _ 5 (Lst.nil _)

#check Lst.append _ as bs
```

It is still tedious, however, to type all these underscores. When a function takes an argument that can generally be inferred from context, Lean allows you to specify that this argument should, by default, be left implicit. This is done by putting the arguments in curly braces, as follows:

```
universe u
def Lst (α : Type u) : Type u := List α

def Lst.cons {α : Type u} (a : α) (as : Lst α) : Lst α := List.cons a as
def Lst.nil {α : Type u} : Lst α := List.nil
def Lst.append {α : Type u} (as bs : Lst α) : Lst α := List.append as bs

#check Lst.cons 0 Lst.nil

def as : Lst Nat := Lst.nil
def bs : Lst Nat := Lst.cons 5 Lst.nil

#check Lst.append as bs
```

All that has changed are the braces around `α : Type u` in the declaration of the variables. We can also use this device in function definitions:

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

#check ident          -- ?m → ?m
#check ident 1        -- Nat
#check ident "hello" -- String
#check @ident         -- {α : Type u_1} → α → α
```

This makes the first argument to `ident` implicit. Notationally, this hides the specification of the type, making it look as though `ident` simply takes an argument of any type. In fact, the function `id` is defined in the standard library in exactly this way. We have chosen a nontraditional name here only to avoid a clash of names.

Variables can also be specified as implicit when they are declared with the `variable` command:

```
universe u

section
  variable {α : Type u}
  variable (x : α)
  def ident := x
end

#check ident
#check ident 4
#check ident "hello"
```

This definition of `ident` here has the same effect as the one above.

Lean has very complex mechanisms for instantiating implicit arguments, and we will see that they can be used to infer function types, predicates, and even proofs. The process of instantiating these "holes," or "placeholders," in a term is often known as *elaboration*. The presence of implicit arguments means that at times there may be insufficient information to fix the meaning of an expression precisely. An expression like `id` or `List.nil` is said to be *polymorphic*, because it can take on different meanings in different contexts.

One can always specify the type `T` of an expression `e` by writing `(e : T)`. This instructs Lean's elaborator to use the value `T` as the type of `e` when trying to resolve implicit arguments. In the second pair of examples below, this mechanism is used to specify the desired types of the expressions `id` and `List.nil`:

```
#check List.nil               -- List ?m
#check id                     -- ?m → ?m

#check (List.nil : List Nat)  -- List Nat
#check (id : Nat → Nat)       -- Nat → Nat
```

Numerals are overloaded in Lean, but when the type of a numeral cannot be inferred, Lean assumes, by default, that it is a natural number. So the expressions in the first two `#check` commands below are elaborated in the same way, whereas the third `#check` command interprets `2` as an integer.

```
#check 2            -- Nat
#check (2 : Nat)    -- Nat
#check (2 : Int)    -- Int
```

Sometimes, however, we may find ourselves in a situation where we have declared an argument to a function to be implicit, but now want to provide the argument explicitly. If `foo` is such a function, the notation `@foo` denotes the same function with all the arguments made explicit.

```
#check @id         -- {α : Sort u_1} → α → α
#check @id Nat     -- Nat → Nat
#check @id Bool    -- Bool → Bool

#check @id Nat 1     -- Nat
#check @id Bool true -- Bool
```

Notice that now the first `#check` command gives the type of the identifier, `id`, without inserting any placeholders. Moreover, the output indicates that the first argument is implicit.

Propositions and Proofs

By now, you have seen some ways of defining objects and functions in Lean. In this chapter, we will begin to explain how to write mathematical assertions and proofs in the language of dependent type theory as well.

Propositions as Types

One strategy for proving assertions about objects defined in the language of dependent type theory is to layer an assertion language and a proof language on top of the definition language. But there is no reason to multiply languages in this way: dependent type theory is flexible and expressive, and there is no reason we cannot represent assertions and proofs in the same general framework.

For example, we could introduce a new type, `Prop`, to represent propositions, and introduce constructors to build new propositions from others.

```
#check And     -- Prop → Prop → Prop
#check Or      -- Prop → Prop → Prop
#check Not     -- Prop → Prop
#check Implies -- Prop → Prop → Prop

variable (p q r : Prop)
#check And p q                      -- Prop
#check Or (And p q) r               -- Prop
#check Implies (And p q) (And q p)  -- Prop
```

We could then introduce, for each element `p : Prop`, another type `Proof p`, for the type of proofs of `p`. An "axiom" would be a constant of such a type.

```
#check Proof   -- Proof : Prop → Type

axiom and_comm (p q : Prop) : Proof (Implies (And p q) (And q p))

variable (p q : Prop)
#check and_comm p q     -- Proof (Implies (And p q) (And q p))
```

In addition to axioms, however, we would also need rules to build new proofs from old ones. For example, in many proof systems for propositional logic, we have the rule of *modus ponens*:

> From a proof of `Implies p q` and a proof of `p`, we obtain a proof of `q`.

We could represent this as follows:

```
axiom modus_ponens : (p q : Prop) → Proof (Implies p q) → Proof p → Proof q
```

Systems of natural deduction for propositional logic also typically rely on the following rule:

> Suppose that, assuming `p` as a hypothesis, we have a proof of `q`. Then we can "cancel" the hypothesis and obtain a proof of `Implies p q`.

We could render this as follows:

```
axiom implies_intro : (p q : Prop) → (Proof p → Proof q) → Proof (Implies p q)
```

This approach would provide us with a reasonable way of building assertions and proofs. Determining that an expression `t` is a correct proof of assertion `p` would then simply be a matter of checking that `t` has type `Proof p`.

Some simplifications are possible, however. To start with, we can avoid writing the term `Proof` repeatedly by conflating `Proof p` with `p` itself. In other words, whenever we have `p : Prop`, we can interpret `p` as a type, namely, the type of its proofs. We can then read `t : p` as the assertion that `t` is a proof of `p`.

Moreover, once we make this identification, the rules for implication show that we can pass back and forth between `Implies p q` and `p → q`. In other words, implication between propositions `p` and `q` corresponds to having a function that takes any element of `p` to an element of `q`. As a result, the introduction of the connective `Implies` is entirely redundant: we can use the usual function space constructor `p → q` from dependent type theory as our notion of implication.

This is the approach followed in the Calculus of Constructions, and hence in Lean as well. The fact that the rules for implication in a proof system for natural deduction correspond exactly to the rules governing abstraction and application for functions is an instance of the *Curry-Howard isomorphism*, sometimes known as the *propositions-as-types* paradigm. In fact, the type `Prop` is syntactic sugar for `Sort 0`, the very bottom of the type hierarchy described in the last chapter. Moreover, `Type u` is also just syntactic sugar for `Sort (u+1)`. `Prop` has some special features, but like the other type universes, it is closed under the arrow constructor: if we have `p q : Prop`, then `p → q : Prop`.

There are at least two ways of thinking about propositions as types. To some who take a constructive view of logic and mathematics, this is a faithful rendering of what it means to be a proposition: a proposition `p` represents a sort of data type, namely, a specification of the type of data that constitutes a proof. A proof of `p` is then simply an object `t : p` of the right type.

Those not inclined to this ideology can view it, rather, as a simple coding trick. To each proposition `p` we associate a type that is empty if `p` is false and has a single element, say `*`, if `p` is true. In the latter case, let us say that (the type associated with) `p` is *inhabited*. It just so happens that the rules for function application and abstraction can conveniently help us keep track of which elements of `Prop` are inhabited. So constructing an element `t : p` tells us that `p` is indeed true. You can think of the inhabitant of `p` as being the "fact that `p` is true." A proof of `p → q` uses "the fact that `p` is true" to obtain "the fact that `q` is true."

Indeed, if `p : Prop` is any proposition, Lean's kernel treats any two elements `t1 t2 : p` as being definitionally equal, much the same way as it treats `(fun x => t) s` and `t[s/x]` as definitionally equal. This is known as *proof irrelevance,* and is consistent with the interpretation in the last paragraph. It means that even though we can treat proofs `t : p` as ordinary objects in the language of dependent type theory, they carry no information beyond the fact that `p` is true.

The two ways we have suggested thinking about the propositions-as-types paradigm differ in a fundamental way. From the constructive point of view, proofs are abstract mathematical objects that are *denoted* by suitable expressions in dependent type theory. In contrast, if we think in terms of the coding trick described above, then the expressions themselves do not denote anything interesting. Rather, it is the fact that we can write them down and check that they are well-typed that ensures that the proposition in question is true. In other words, the expressions *themselves* are the proofs.

In the exposition below, we will slip back and forth between these two ways of talking, at times saying that an expression "constructs" or "produces" or "returns" a proof of a proposition, and at other times simply saying that it "is" such a proof. This is similar to the way that computer scientists occasionally blur the distinction between syntax and semantics by saying, at times, that a program "computes" a certain function, and at other times speaking as though the program "is" the function in question.

In any case, all that really matters is the bottom line. To formally express a mathematical assertion in the language of dependent type theory, we need to exhibit a term `p : Prop`. To *prove* that assertion, we need to exhibit a term `t : p`. Lean's task, as a proof assistant, is to help us to construct such a term, `t`, and to verify that it is well-formed and has the correct type.

Working with Propositions as Types

In the propositions-as-types paradigm, theorems involving only `→` can be proved using lambda abstraction and application. In Lean, the `theorem` command introduces a new theorem:

```
variable {p : Prop}
variable {q : Prop}

theorem t1 : p → q → p := fun hp : p => fun hq : q => hp
```

Compare this proof to the expression `fun x : α => fun y : β => x` of type `α → β → α`, where `α` and `β` are data types. This describes the function that takes arguments `x` and `y` of type `α` and `β`, respectively, and returns `x`. The proof of `t1` has the same form, the only difference being that `p` and `q` are elements of `Prop` rather than `Type`. Intuitively, our proof of `p → q → p` assumes `p` and `q` are true, and uses the first hypothesis (trivially) to establish that the conclusion, `p`, is true.

Note that the `theorem` command is really a version of the `def` command: under the propositions and types correspondence, proving the theorem `p → q → p` is really the same as defining an element of the associated type. To the kernel type checker, there is no difference between the two.

There are a few pragmatic differences between definitions and theorems, however. In normal circumstances, it is never necessary to unfold the "definition" of a theorem; by proof irrelevance, any two proofs of that theorem are definitionally equal. Once the proof of a theorem is complete, typically we only need to know that the proof exists; it doesn't matter what the proof is. In light of that fact, Lean tags proofs as *irreducible*, which serves as a hint to the parser (more precisely, the *elaborator*) that there is generally no need to unfold them when processing a file. In fact, Lean is generally able to process and check proofs in parallel, since assessing the correctness of one proof does not require knowing the details of another.

As with definitions, the `#print` command will show you the proof of a theorem:

```
theorem t1 : p → q → p := fun hp : p => fun hq : q => hp

#print t1
```

Notice that the lambda abstractions `hp : p` and `hq : q` can be viewed as temporary assumptions in the proof of `t1`. Lean also allows us to specify the type of the final term `hp`, explicitly, with a `show` statement:

```
theorem t1 : p → q → p :=
  fun hp : p =>
  fun hq : q =>
  show p from hp
```

Adding such extra information can improve the clarity of a proof and help detect errors when writing a proof. The `show` command does nothing more than annotate the type, and, internally, all the presentations of `t1` that we have seen produce the same term.

As with ordinary definitions, we can move the lambda-abstracted variables to the left of the colon:

```
theorem t1 (hp : p) (hq : q) : p := hp

#print t1    -- p → q → p
```

We can use the theorem `t1` just as a function application:

```
theorem t1 (hp : p) (hq : q) : p := hp

axiom hp : p

theorem t2 : q → p := t1 hp
```

The `axiom` declaration postulates the existence of an element of the given type and may compromise logical consistency. For example, we can use it to postulate that the empty type `False` has an element:

```
axiom unsound : False
-- Everything follows from false
theorem ex : 1 = 0 :=
  False.elim unsound
```

Declaring an "axiom" `hp : p` is tantamount to declaring that `p` is true, as witnessed by `hp`. Applying the theorem `t1 : p → q → p` to the fact `hp : p` that `p` is true yields the theorem `t1 hp : q → p`.

Recall that we can also write theorem `t1` as follows:

```
theorem t1 {p q : Prop} (hp : p) (hq : q) : p := hp

#print t1
```

The type of `t1` is now `∀ {p q : Prop}, p → q → p`. We can read this as the assertion "for every pair of propositions `p q`, we have `p → q → p`." For example, we can move all parameters to the right of the colon:

```
theorem t1 : ∀ {p q : Prop}, p → q → p :=
  fun {p q : Prop} (hp : p) (hq : q) => hp
```

If `p` and `q` have been declared as variables, Lean will generalize them for us automatically:

```
variable {p q : Prop}

theorem t1 : p → q → p := fun (hp : p) (hq : q) => hp
```

In fact, by the propositions-as-types correspondence, we can declare the assumption `hp` that `p` holds, as another variable:

```
variable {p q : Prop}
variable (hp : p)

theorem t1 : q → p := fun (hq : q) => hp
```

Lean detects that the proof uses `hp` and automatically adds `hp : p` as a premise. In all cases, the command `#print t1` still yields `∀ p q : Prop, p → q → p`. Remember that this type can just as well be written `∀ (p q : Prop) (hp : p) (hq : q), p`, since the arrow denotes nothing more than an arrow type in which the target does not depend on the bound variable.

When we generalize `t1` in such a way, we can then apply it to different pairs of propositions, to obtain different instances of the general theorem.

```
theorem t1 (p q : Prop) (hp : p) (hq : q) : p := hp

variable (p q r s : Prop)

#check t1 p q                -- p → q → p
#check t1 r s                -- r → s → r
#check t1 (r → s) (s → r)    -- (r → s) → (s → r) → r → s

variable (h : r → s)
#check t1 (r → s) (s → r) h  -- (s → r) → r → s
```

Once again, using the propositions-as-types correspondence, the variable `h` of type `r → s` can be viewed as the hypothesis, or premise, that `r → s` holds.

As another example, let us consider the composition function discussed in the last chapter, now with propositions instead of types.

```
variable (p q r s : Prop)

theorem t2 (h₁ : q → r) (h₂ : p → q) : p → r :=
  fun h₃ : p =>
  show r from h₁ (h₂ h₃)
```

As a theorem of propositional logic, what does `t2` say?

Note that it is often useful to use numeric unicode subscripts, entered as `\0`, `\1`, `\2`, ..., for hypotheses, as we did in this example.

Propositional Logic

Lean defines all the standard logical connectives and notation. The propositional connectives come with the following notation:

Ascii	Unicode	Editor shortcut	Definition
True			True
False			False
Not	¬	`\not`, `\neg`	Not
/\	∧	`\and`	And
\/	∨	`\or`	Or
->	→	`\to`, `\r`, `\imp`	
<->	↔	`\iff`, `\lr`	Iff

They all take values in `Prop`.

```
variable (p q : Prop)

#check p → q → p ∧ q
#check ¬p → p ↔ False
#check p ∨ q → q ∨ p
```

The order of operations is as follows: unary negation `¬` binds most strongly, then `∧`, then `∨`, then `→`, and finally `↔`. For example, `a ∧ b → c ∨ d ∧ e` means `(a ∧ b) → (c ∨ (d ∧ e))`. Remember that `→` associates to the right (nothing changes now that the arguments are elements of `Prop`, instead of some other `Type`), as do the other binary connectives. So if we have `p q r : Prop`, the expression `p → q → r` reads "if `p`, then if `q`, then `r`." This is just the "curried" form of `p ∧ q → r`.

In the last chapter we observed that lambda abstraction can be viewed as an "introduction rule" for `→`. In the current setting, it shows how to "introduce" or establish an implication. Application can be viewed as an "elimination rule," showing how to "eliminate" or use an implication in a proof. The other propositional connectives are defined in Lean's library in the file `Prelude.core` (see importing files for more information on the library hierarchy), and each connective comes with its canonical introduction and elimination rules.

Conjunction

The expression `And.intro h1 h2` builds a proof of `p ∧ q` using proofs `h1 : p` and `h2 : q`. It is common to describe `And.intro` as the *and-introduction* rule. In the next example we use `And.intro` to create a proof of `p → q → p ∧ q`.

```
variable (p q : Prop)

example (hp : p) (hq : q) : p ∧ q := And.intro hp hq

#check fun (hp : p) (hq : q) => And.intro hp hq
```

The `example` command states a theorem without naming it or storing it in the permanent context. Essentially, it just checks that the given term has the indicated type. It is convenient for illustration, and we will use it often.

The expression `And.left h` creates a proof of `p` from a proof `h : p ∧ q`. Similarly, `And.right h` is a proof of `q`. They are commonly known as the left and right *and-elimination* rules.

```
variable (p q : Prop)

example (h : p ∧ q) : p := And.left h
example (h : p ∧ q) : q := And.right h
```

We can now prove `p ∧ q → q ∧ p` with the following proof term.

```
variable (p q : Prop)

example (h : p ∧ q) : q ∧ p :=
  And.intro (And.right h) (And.left h)
```

Notice that and-introduction and and-elimination are similar to the pairing and projection operations for the Cartesian product. The difference is that given `hp : p` and `hq : q`, `And.intro hp hq` has type `p ∧ q : Prop`, while `Prod hp hq` has type `p × q : Type`. The similarity between `∧` and `×` is another instance of the Curry-Howard isomorphism, but in contrast to implication and the function space constructor, `∧` and `×` are treated separately in Lean. With the analogy, however, the proof we have just constructed is similar to a function that swaps the elements of a pair.

We will see in Chapter Structures and Records that certain types in Lean are *structures*, which is to say, the type is defined with a single canonical *constructor* which builds an element of the type from a sequence of suitable arguments. For every `p q : Prop`, `p ∧ q` is an example: the canonical way to construct an element is to apply `And.intro` to suitable arguments `hp : p` and `hq : q`. Lean allows us to use *anonymous constructor* notation `⟨arg1, arg2, ...⟩` in situations like these, when the relevant type is an inductive type and can be inferred from the context. In particular, we can often write `⟨hp, hq⟩` instead of `And.intro hp hq`:

```
variable (p q : Prop)
variable (hp : p) (hq : q)

#check (⟨hp, hq⟩ : p ∧ q)
```

These angle brackets are obtained by typing `\<` and `\>`, respectively.

Lean provides another useful syntactic gadget. Given an expression `e` of an inductive type `Foo` (possibly applied to some arguments), the notation `e.bar` is shorthand for `Foo.bar e`. This provides a convenient way of accessing functions without opening a namespace. For example, the following two expressions mean the same thing:

```
variable (xs : List Nat)

#check List.length xs
#check xs.length
```

As a result, given `h : p ∧ q`, we can write `h.left` for `And.left h` and `h.right` for `And.right h`. We can therefore rewrite the sample proof above conveniently as follows:

```
variable (p q : Prop)

example (h : p ∧ q) : q ∧ p :=
  ⟨h.right, h.left⟩
```

There is a fine line between brevity and obfuscation, and omitting information in this way can sometimes make a proof harder to read. But for straightforward constructions like the one above, when the type of `h` and the goal of the construction are salient, the notation is clean and effective.

It is common to iterate constructions like "And." Lean also allows you to flatten nested constructors that associate to the right, so that these two proofs are equivalent:

```
variable (p q : Prop)

example (h : p ∧ q) : q ∧ p ∧ q :=
  ⟨h.right, ⟨h.left, h.right⟩⟩

example (h : p ∧ q) : q ∧ p ∧ q :=
  ⟨h.right, h.left, h.right⟩
```

This is often useful as well.

Disjunction

The expression `Or.intro_left q hp` creates a proof of `p ∨ q` from a proof `hp : p`. Similarly, `Or.intro_right p hq` creates a proof for `p ∨ q` using a proof `hq : q`. These are the left and right *or-introduction* rules.

```
variable (p q : Prop)
example (hp : p) : p ∨ q := Or.intro_left q hp
example (hq : q) : p ∨ q := Or.intro_right p hq
```

The *or-elimination* rule is slightly more complicated. The idea is that we can prove `r` from `p ∨ q`, by showing that `r` follows from `p` and that `r` follows from `q`. In other words, it is a proof by cases. In the expression `Or.elim hpq hpr hqr`, `Or.elim` takes three arguments, `hpq : p ∨ q`, `hpr : p → r` and `hqr : q → r`, and produces a proof of `r`. In the following example, we use `Or.elim` to prove `p ∨ q → q ∨ p`.

```
variable (p q r : Prop)

example (h : p ∨ q) : q ∨ p :=
  Or.elim h
    (fun hp : p =>
      show q ∨ p from Or.intro_right q hp)
    (fun hq : q =>
      show q ∨ p from Or.intro_left p hq)
```

In most cases, the first argument of `Or.intro_right` and `Or.intro_left` can be inferred automatically by Lean. Lean therefore provides `Or.inr` and `Or.inl` which can be viewed as shorthand for `Or.intro_right _` and `Or.intro_left _`. Thus the proof term above could be written more concisely:

```
variable (p q r : Prop)

example (h : p ∨ q) : q ∨ p :=
  Or.elim h (fun hp => Or.inr hp) (fun hq => Or.inl hq)
```

Notice that there is enough information in the full expression for Lean to infer the types of `hp` and `hq` as well. But using the type annotations in the longer version makes the proof more readable, and can help catch and debug errors.

Because `Or` has two constructors, we cannot use anonymous constructor notation. But we can still write `h.elim` instead of `Or.elim h`:

```
variable (p q r : Prop)

example (h : p ∨ q) : q ∨ p :=
  h.elim (fun hp => Or.inr hp) (fun hq => Or.inl hq)
```

Once again, you should exercise judgment as to whether such abbreviations enhance or diminish readability.

Negation and Falsity

Negation, `¬p`, is actually defined to be `p → False`, so we obtain `¬p` by deriving a contradiction from `p`. Similarly, the expression `hnp hp` produces a proof of `False` from `hp : p` and `hnp : ¬p`. The next example uses both these rules to produce a proof of `(p → q) → ¬q → ¬p`. (The symbol `¬` is produced by typing `\not` or `\neg`.)

```
variable (p q : Prop)

example (hpq : p → q) (hnq : ¬q) : ¬p :=
  fun hp : p =>
  show False from hnq (hpq hp)
```

The connective `False` has a single elimination rule, `False.elim`, which expresses the fact that anything follows from a contradiction. This rule is sometimes called *ex falso* (short for *ex falso sequitur quodlibet*), or the *principle of explosion*.

```
variable (p q : Prop)

example (hp : p) (hnp : ¬p) : q := False.elim (hnp hp)
```

The arbitrary fact, `q`, that follows from falsity is an implicit argument in `False.elim` and is inferred automatically. This pattern, deriving an arbitrary fact from contradictory hypotheses, is quite common, and is represented by `absurd`.

```
variable (p q : Prop)

example (hp : p) (hnp : ¬p) : q := absurd hp hnp
```

Here, for example, is a proof of `¬p → q → (q → p) → r`:

```
variable (p q r : Prop)

example (hnp : ¬p) (hq : q) (hqp : q → p) : r :=
  absurd (hqp hq) hnp
```

Incidentally, just as `False` has only an elimination rule, `True` has only an introduction rule, `True.intro : true`. In other words, `True` is simply true, and has a canonical proof, `True.intro`.

Logical Equivalence

The expression `Iff.intro h1 h2` produces a proof of `p ↔ q` from `h1 : p → q` and `h2 : q → p`. The expression `Iff.mp h` produces a proof of `p → q` from `h : p ↔ q`. Similarly, `Iff.mpr h` produces a proof of `q → p` from `h : p ↔ q`. Here is a proof of `p ∧ q ↔ q ∧ p`:

```
variable (p q : Prop)

theorem and_swap : p ∧ q ↔ q ∧ p :=
  Iff.intro
    (fun h : p ∧ q =>
     show q ∧ p from And.intro (And.right h) (And.left h))
    (fun h : q ∧ p =>
     show p ∧ q from And.intro (And.right h) (And.left h))

#check and_swap p q    -- p ∧ q ↔ q ∧ p

variable (h : p ∧ q)
example : q ∧ p := Iff.mp (and_swap p q) h
```

We can use the anonymous constructor notation to construct a proof of `p ↔ q` from proofs of the forward and backward directions, and we can also use `.` notation with `mp` and `mpr`. The previous examples can therefore be written concisely as follows:

```
variable (p q : Prop)

theorem and_swap : p ∧ q ↔ q ∧ p :=
  ⟨ fun h => ⟨h.right, h.left⟩, fun h => ⟨h.right, h.left⟩ ⟩

example (h : p ∧ q) : q ∧ p := (and_swap p q).mp h
```

Introducing Auxiliary Subgoals

This is a good place to introduce another device Lean offers to help structure long proofs, namely, the `have` construct, which introduces an auxiliary subgoal in a proof. Here is a small example, adapted from the last section:

```
variable (p q : Prop)

example (h : p ∧ q) : q ∧ p :=
  have hp : p := h.left
  have hq : q := h.right
  show q ∧ p from And.intro hq hp
```

Internally, the expression `have h : p := s; t` produces the term `(fun (h : p) => t) s`. In other words, `s` is a proof of `p`, `t` is a proof of the desired conclusion assuming `h : p`, and the two are combined by a lambda abstraction and application. This simple device is extremely useful when it comes to structuring long proofs, since we can use intermediate `have`'s as stepping stones leading to the final goal.

Lean also supports a structured way of reasoning backwards from a goal, which models the "suffices to show" construction in ordinary mathematics. The next example simply permutes the last two lines in the previous proof.

```
variable (p q : Prop)

example (h : p ∧ q) : q ∧ p :=
  have hp : p := h.left
  suffices hq : q from And.intro hq hp
  show q from And.right h
```

Writing `suffices hq : q` leaves us with two goals. First, we have to show that it indeed suffices to show `q`, by proving the original goal of `q ∧ p` with the additional hypothesis `hq : q`. Finally, we have to show `q`.

Classical Logic

The introduction and elimination rules we have seen so far are all constructive, which is to say, they reflect a computational understanding of the logical connectives based on the propositions-as-types correspondence. Ordinary classical logic adds to this the law of the excluded middle, `p ∨ ¬p`. To use this principle, you have to open the classical namespace.

```
open Classical

variable (p : Prop)
#check em p
```

Intuitively, the constructive "Or" is very strong: asserting `p ∨ q` amounts to knowing which is the case. If `RH` represents the Riemann hypothesis, a classical mathematician is willing to assert `RH ∨ ¬RH`, even though we cannot yet assert either disjunct.

One consequence of the law of the excluded middle is the principle of double-negation elimination:

```
open Classical

theorem dne {p : Prop} (h : ¬¬p) : p :=
  Or.elim (em p)
    (fun hp : p => hp)
    (fun hnp : ¬p => absurd hnp h)
```

Double-negation elimination allows one to prove any proposition, `p`, by assuming `¬p` and deriving `false`, because that amounts to proving `¬¬p`. In other words, double-negation elimination allows one to carry out a proof by contradiction, something which is not generally possible in constructive logic. As an exercise, you might try proving the converse, that is, showing that `em` can be proved from `dne`.

The classical axioms also give you access to additional patterns of proof that can be justified by appeal to `em`. For example, one can carry out a proof by cases:

```
open Classical
variable (p : Prop)

example (h : ¬¬p) : p :=
  byCases
    (fun h1 : p => h1)
    (fun h1 : ¬p => absurd h1 h)
```

Or you can carry out a proof by contradiction:

```
open Classical
variable (p : Prop)

example (h : ¬¬p) : p :=
  byContradiction
    (fun h1 : ¬p =>
     show False from h h1)
```

If you are not used to thinking constructively, it may take some time for you to get a sense of where classical reasoning is used. It is needed in the following example because, from a constructive standpoint, knowing that `p` and `q` are not both true does not necessarily tell you which one is false:

```
example (h : ¬(p ∧ q)) : ¬p ∨ ¬q :=
  Or.elim (em p)
    (fun hp : p =>
      Or.inr
        (show ¬q from
          fun hq : q =>
          h ⟨hp, hq⟩))
    (fun hp : ¬p =>
      Or.inl hp)
```

We will see later that there *are* situations in constructive logic where principles like excluded middle and double-negation elimination are permissible, and Lean supports the use of classical reasoning in such contexts without relying on excluded middle.

The full list of axioms that are used in Lean to support classical reasoning are discussed in Axioms and Computation.

Examples of Propositional Validities

Lean's standard library contains proofs of many valid statements of propositional logic, all of which you are free to use in proofs of your own. The following list includes a number of common identities.

Commutativity:

1. `p ∧ q ↔ q ∧ p`

2. `p ∨ q ↔ q ∨ p`

Associativity:

3. `(p ∧ q) ∧ r ↔ p ∧ (q ∧ r)`
4. `(p ∨ q) ∨ r ↔ p ∨ (q ∨ r)`

Distributivity:

5. `p ∧ (q ∨ r) ↔ (p ∧ q) ∨ (p ∧ r)`
6. `p ∨ (q ∧ r) ↔ (p ∨ q) ∧ (p ∨ r)`

Other properties:

7. `(p → (q → r)) ↔ (p ∧ q → r)`
8. `((p ∨ q) → r) ↔ (p → r) ∧ (q → r)`
9. `¬(p ∨ q) ↔ ¬p ∧ ¬q`
10. `¬p ∨ ¬q → ¬(p ∧ q)`
11. `¬(p ∧ ¬p)`
12. `p ∧ ¬q → ¬(p → q)`
13. `¬p → (p → q)`
14. `(¬p ∨ q) → (p → q)`
15. `p ∨ False ↔ p`
16. `p ∧ False ↔ False`
17. `¬(p ↔ ¬p)`
18. `(p → q) → (¬q → ¬p)`

These require classical reasoning:

19. `(p → r ∨ s) → ((p → r) ∨ (p → s))`
20. `¬(p ∧ q) → ¬p ∨ ¬q`
21. `¬(p → q) → p ∧ ¬q`
22. `(p → q) → (¬p ∨ q)`
23. `(¬q → ¬p) → (p → q)`
24. `p ∨ ¬p`
25. `(((p → q) → p) → p)`

The `sorry` identifier magically produces a proof of anything, or provides an object of any data type at all. Of course, it is unsound as a proof method -- for example, you can use it to prove `False` -- and Lean produces severe warnings when files use or import theorems which depend on it. But it is very useful for building long proofs incrementally. Start writing the proof from the top down, using `sorry` to fill in subproofs. Make sure Lean accepts the term with all the `sorry`'s; if not, there are errors that you need to correct. Then go back and replace each `sorry` with an actual proof, until no more remain.

Here is another useful trick. Instead of using `sorry`, you can use an underscore `_` as a placeholder. Recall this tells Lean that the argument is implicit, and should be filled in

automatically. If Lean tries to do so and fails, it returns with an error message "don't know how to synthesize placeholder," followed by the type of the term it is expecting, and all the objects and hypotheses available in the context. In other words, for each unresolved placeholder, Lean reports the subgoal that needs to be filled at that point. You can then construct a proof by incrementally filling in these placeholders.

For reference, here are two sample proofs of validities taken from the list above.

```
open Classical

-- distributivity
example (p q r : Prop) : p ∧ (q ∨ r) ↔ (p ∧ q) ∨ (p ∧ r) :=
  Iff.intro
    (fun h : p ∧ (q ∨ r) =>
      have hp : p := h.left
      Or.elim (h.right)
        (fun hq : q =>
          show (p ∧ q) ∨ (p ∧ r) from Or.inl ⟨hp, hq⟩)
        (fun hr : r =>
          show (p ∧ q) ∨ (p ∧ r) from Or.inr ⟨hp, hr⟩))
    (fun h : (p ∧ q) ∨ (p ∧ r) =>
      Or.elim h
        (fun hpq : p ∧ q =>
          have hp : p := hpq.left
          have hq : q := hpq.right
          show p ∧ (q ∨ r) from ⟨hp, Or.inl hq⟩)
        (fun hpr : p ∧ r =>
          have hp : p := hpr.left
          have hr : r := hpr.right
          show p ∧ (q ∨ r) from ⟨hp, Or.inr hr⟩))

-- an example that requires classical reasoning
example (p q : Prop) : ¬(p ∧ ¬q) → (p → q) :=
  fun h : ¬(p ∧ ¬q) =>
  fun hp : p =>
  show q from
    Or.elim (em q)
      (fun hq : q => hq)
      (fun hnq : ¬q => absurd (And.intro hp hnq) h)
```

Exercises

Prove the following identities, replacing the "sorry" placeholders with actual proofs.

```
variable (p q r : Prop)

-- commutativity of ∧ and ∨
example : p ∧ q ↔ q ∧ p := sorry
example : p ∨ q ↔ q ∨ p := sorry

-- associativity of ∧ and ∨
example : (p ∧ q) ∧ r ↔ p ∧ (q ∧ r) := sorry
example : (p ∨ q) ∨ r ↔ p ∨ (q ∨ r) := sorry

-- distributivity
example : p ∧ (q ∨ r) ↔ (p ∧ q) ∨ (p ∧ r) := sorry
example : p ∨ (q ∧ r) ↔ (p ∨ q) ∧ (p ∨ r) := sorry

-- other properties
example : (p → (q → r)) ↔ (p ∧ q → r) := sorry
example : ((p ∨ q) → r) ↔ (p → r) ∧ (q → r) := sorry
example : ¬(p ∨ q) ↔ ¬p ∧ ¬q := sorry
example : ¬p ∨ ¬q → ¬(p ∧ q) := sorry
example : ¬(p ∧ ¬p) := sorry
example : p ∧ ¬q → ¬(p → q) := sorry
example : ¬p → (p → q) := sorry
example : (¬p ∨ q) → (p → q) := sorry
example : p ∨ False ↔ p := sorry
example : p ∧ False ↔ False := sorry
example : (p → q) → (¬q → ¬p) := sorry
```

Prove the following identities, replacing the "sorry" placeholders with actual proofs. These require classical reasoning.

```
open Classical

variable (p q r : Prop)

example : (p → q ∨ r) → ((p → q) ∨ (p → r)) := sorry
example : ¬(p ∧ q) → ¬p ∨ ¬q := sorry
example : ¬(p → q) → p ∧ ¬q := sorry
example : (p → q) → (¬p ∨ q) := sorry
example : (¬q → ¬p) → (p → q) := sorry
example : p ∨ ¬p := sorry
example : (((p → q) → p) → p) := sorry
```

Prove `¬(p ↔ ¬p)` without using classical logic.

Quantifiers and Equality

The last chapter introduced you to methods that construct proofs of statements involving the propositional connectives. In this chapter, we extend the repertoire of logical constructions to include the universal and existential quantifiers, and the equality relation.

The Universal Quantifier

Notice that if `α` is any type, we can represent a unary predicate `p` on `α` as an object of type `α → Prop`. In that case, given `x : α`, `p x` denotes the assertion that `p` holds of `x`. Similarly, an object `r : α → α → Prop` denotes a binary relation on `α`: given `x y : α`, `r x y` denotes the assertion that `x` is related to `y`.

The universal quantifier, `∀ x : α, p x` is supposed to denote the assertion that "for every `x : α`, `p x`" holds. As with the propositional connectives, in systems of natural deduction, "forall" is governed by an introduction and elimination rule. Informally, the introduction rule states:

> Given a proof of `p x`, in a context where `x : α` is arbitrary, we obtain a proof `∀ x : α, p x`.

The elimination rule states:

> Given a proof `∀ x : α, p x` and any term `t : α`, we obtain a proof of `p t`.

As was the case for implication, the propositions-as-types interpretation now comes into play. Remember the introduction and elimination rules for dependent arrow types:

> Given a term `t` of type `β x`, in a context where `x : α` is arbitrary, we have `(fun x : α => t) : (x : α) → β x`.

The elimination rule states:

> Given a term `s : (x : α) → β x` and any term `t : α`, we have `s t : β t`.

In the case where `p x` has type `Prop`, if we replace `(x : α) → β x` with `∀ x : α, p x`, we can read these as the correct rules for building proofs involving the universal quantifier.

The Calculus of Constructions therefore identifies dependent arrow types with forall-expressions in this way. If `p` is any expression, `∀ x : α, p` is nothing more than alternative notation for `(x : α) → p`, with the idea that the former is more natural than the latter in cases where `p` is a proposition. Typically, the expression `p` will depend on `x : α`. Recall that, in the case of ordinary function spaces, we could interpret `α → β` as the special case of `(x : α) → β` in which `β` does not depend on `x`. Similarly, we can think of an implication `p`

`→ q` between propositions as the special case of `∀ x : p, q` in which the expression `q` does not depend on `x`.

Here is an example of how the propositions-as-types correspondence gets put into practice.

```
example (α : Type) (p q : α → Prop) : (∀ x : α, p x ∧ q x) → ∀ y : α, p y :=
  fun h : ∀ x : α, p x ∧ q x =>
  fun y : α =>
  show p y from (h y).left
```

As a notational convention, we give the universal quantifier the widest scope possible, so parentheses are needed to limit the quantifier over `x` to the hypothesis in the example above. The canonical way to prove `∀ y : α, p y` is to take an arbitrary `y`, and prove `p y`. This is the introduction rule. Now, given that `h` has type `∀ x : α, p x ∧ q x`, the expression `h y` has type `p y ∧ q y`. This is the elimination rule. Taking the left conjunct gives the desired conclusion, `p y`.

Remember that expressions which differ up to renaming of bound variables are considered to be equivalent. So, for example, we could have used the same variable, `x`, in both the hypothesis and conclusion, and instantiated it by a different variable, `z`, in the proof:

```
example (α : Type) (p q : α → Prop) : (∀ x : α, p x ∧ q x) → ∀ x : α, p x :=
  fun h : ∀ x : α, p x ∧ q x =>
  fun z : α =>
  show p z from And.left (h z)
```

As another example, here is how we can express the fact that a relation, `r`, is transitive:

```
variable (α : Type) (r : α → α → Prop)
variable (trans_r : ∀ x y z, r x y → r y z → r x z)

variable (a b c : α)
variable (hab : r a b) (hbc : r b c)

#check trans_r    -- ∀ (x y z : α), r x y → r y z → r x z
#check trans_r a b c -- r a b → r b c → r a c
#check trans_r a b c hab -- r b c → r a c
#check trans_r a b c hab hbc -- r a c
```

Think about what is going on here. When we instantiate `trans_r` at the values `a b c`, we end up with a proof of `r a b → r b c → r a c`. Applying this to the "hypothesis" `hab : r a b`, we get a proof of the implication `r b c → r a c`. Finally, applying it to the hypothesis `hbc` yields a proof of the conclusion `r a c`.

In situations like this, it can be tedious to supply the arguments `a b c`, when they can be inferred from `hab hbc`. For that reason, it is common to make these arguments implicit:

```
variable (α : Type) (r : α → α → Prop)
variable (trans_r : ∀ {x y z}, r x y → r y z → r x z)

variable (a b c : α)
variable (hab : r a b) (hbc : r b c)

#check trans_r
#check trans_r hab
#check trans_r hab hbc
```

The advantage is that we can simply write `trans_r hab hbc` as a proof of `r a c`. A disadvantage is that Lean does not have enough information to infer the types of the arguments in the expressions `trans_r` and `trans_r hab`. The output of the first `#check` command is `r ?m.1 ?m.2 → r ?m.2 ?m.3 → r ?m.1 ?m.3`, indicating that the implicit arguments are unspecified in this case.

Here is an example of how we can carry out elementary reasoning with an equivalence relation:

```
variable (α : Type) (r : α → α → Prop)

variable (refl_r : ∀ x, r x x)
variable (symm_r : ∀ {x y}, r x y → r y x)
variable (trans_r : ∀ {x y z}, r x y → r y z → r x z)

example (a b c d : α) (hab : r a b) (hcb : r c b) (hcd : r c d) : r a d :=
  trans_r (trans_r hab (symm_r hcb)) hcd
```

To get used to using universal quantifiers, you should try some of the exercises at the end of this section.

It is the typing rule for dependent arrow types, and the universal quantifier in particular, that distinguishes `Prop` from other types. Suppose we have `α : Sort i` and `β : Sort j`, where the expression `β` may depend on a variable `x : α`. Then `(x : α) → β` is an element of `Sort (imax i j)`, where `imax i j` is the maximum of `i` and `j` if `j` is not 0, and 0 otherwise.

The idea is as follows. If `j` is not `0`, then `(x : α) → β` is an element of `Sort (max i j)`. In other words, the type of dependent functions from `α` to `β` "lives" in the universe whose index is the maximum of `i` and `j`. Suppose, however, that `β` is of `Sort 0`, that is, an element of `Prop`. In that case, `(x : α) → β` is an element of `Sort 0` as well, no matter which type universe `α` lives in. In other words, if `β` is a proposition depending on `α`, then `∀ x : α, β` is again a proposition. This reflects the interpretation of `Prop` as the type of propositions rather than data, and it is what makes `Prop` *impredicative*.

The term "predicative" stems from foundational developments around the turn of the twentieth century, when logicians such as Poincaré and Russell blamed set-theoretic paradoxes on the "vicious circles" that arise when we define a property by quantifying over a collection that includes the very property being defined. Notice that if `α` is any type, we can

form the type `α → Prop` of all predicates on `α` (the "power type of `α`"). The impredicativity of `Prop` means that we can form propositions that quantify over `α → Prop`. In particular, we can define predicates on `α` by quantifying over all predicates on `α`, which is exactly the type of circularity that was once considered problematic.

Equality

Let us now turn to one of the most fundamental relations defined in Lean's library, namely, the equality relation. In Chapter Inductive Types, we will explain *how* equality is defined from the primitives of Lean's logical framework. In the meanwhile, here we explain how to use it.

Of course, a fundamental property of equality is that it is an equivalence relation:

```
#check Eq.refl    -- Eq.refl.{u_1} {α : Sort u_1} (a : α) : a = a
#check Eq.symm    -- Eq.symm.{u} {α : Sort u} {a b : α} (h : a = b) : b = a
#check Eq.trans   -- Eq.trans.{u} {α : Sort u} {a b c : α} (h₁ : a = b) (h₂ : b
= c) : a = c
```

We can make the output easier to read by telling Lean not to insert the implicit arguments (which are displayed here as metavariables).

```
universe u

#check @Eq.refl.{u}   -- @Eq.refl : ∀ {α : Sort u} (a : α), a = a
#check @Eq.symm.{u}   -- @Eq.symm : ∀ {α : Sort u} {a b : α}, a = b → b = a
#check @Eq.trans.{u}  -- @Eq.trans : ∀ {α : Sort u} {a b c : α}, a = b → b = c →
a = c
```

The inscription `.{u}` tells Lean to instantiate the constants at the universe `u`.

Thus, for example, we can specialize the example from the previous section to the equality relation:

```
variable (α : Type) (a b c d : α)
variable (hab : a = b) (hcb : c = b) (hcd : c = d)

example : a = d :=
  Eq.trans (Eq.trans hab (Eq.symm hcb)) hcd
```

We can also use the projection notation:

```
example : a = d := (hab.trans hcb.symm).trans hcd
```

Reflexivity is more powerful than it looks. Recall that terms in the Calculus of Constructions have a computational interpretation, and that the logical framework treats terms with a common reduct as the same. As a result, some nontrivial identities can be proved by reflexivity:

```
variable (α β : Type)

example (f : α → β) (a : α) : (fun x => f x) a = f a := Eq.refl _
example (a : α) (b : β) : (a, b).1 = a := Eq.refl _
example : 2 + 3 = 5 := Eq.refl _
```

This feature of the framework is so important that the library defines a notation `rfl` for `Eq.refl _`:

```
example (f : α → β) (a : α) : (fun x => f x) a = f a := rfl
example (a : α) (b : β) : (a, b).1 = a := rfl
example : 2 + 3 = 5 := rfl
```

Equality is much more than an equivalence relation, however. It has the important property that every assertion respects the equivalence, in the sense that we can substitute equal expressions without changing the truth value. That is, given `h1 : a = b` and `h2 : p a`, we can construct a proof for `p b` using substitution: `Eq.subst h1 h2`.

```
example (α : Type) (a b : α) (p : α → Prop)
        (h1 : a = b) (h2 : p a) : p b :=
  Eq.subst h1 h2

example (α : Type) (a b : α) (p : α → Prop)
    (h1 : a = b) (h2 : p a) : p b :=
  h1 ▸ h2
```

The triangle in the second presentation is a macro built on top of `Eq.subst` and `Eq.symm`, and you can enter it by typing `\t`.

The rule `Eq.subst` is used to define the following auxiliary rules, which carry out more explicit substitutions. They are designed to deal with applicative terms, that is, terms of form `s t`. Specifically, `congrArg` can be used to replace the argument, `congrFun` can be used to replace the term that is being applied, and `congr` can be used to replace both at once.

```
variable (α : Type)
variable (a b : α)
variable (f g : α → Nat)
variable (h₁ : a = b)
variable (h₂ : f = g)

example : f a = f b := congrArg f h₁
example : f a = g a := congrFun h₂ a
example : f a = g b := congr h₂ h₁
```

Lean's library contains a large number of common identities, such as these:

```
variable (a b c : Nat)

example : a + 0 = a := Nat.add_zero a
example : 0 + a = a := Nat.zero_add a
example : a * 1 = a := Nat.mul_one a
example : 1 * a = a := Nat.one_mul a
example : a + b = b + a := Nat.add_comm a b
example : a + b + c = a + (b + c) := Nat.add_assoc a b c
example : a * b = b * a := Nat.mul_comm a b
example : a * b * c = a * (b * c) := Nat.mul_assoc a b c
example : a * (b + c) = a * b + a * c := Nat.mul_add a b c
example : a * (b + c) = a * b + a * c := Nat.left_distrib a b c
example : (a + b) * c = a * c + b * c := Nat.add_mul a b c
example : (a + b) * c = a * c + b * c := Nat.right_distrib a b c
```

Note that `Nat.mul_add` and `Nat.add_mul` are alternative names for `Nat.left_distrib` and `Nat.right_distrib`, respectively. The properties above are stated for the natural numbers (type `Nat`).

Here is an example of a calculation in the natural numbers that uses substitution combined with associativity and distributivity.

```
example (x y : Nat) : (x + y) * (x + y) = x * x + y * x + x * y + y * y :=
  have h1 : (x + y) * (x + y) = (x + y) * x + (x + y) * y :=
    Nat.mul_add (x + y) x y
  have h2 : (x + y) * (x + y) = x * x + y * x + (x * y + y * y) :=
    (Nat.add_mul x y x) ▸ (Nat.add_mul x y y) ▸ h1
  h2.trans (Nat.add_assoc (x * x + y * x) (x * y) (y * y)).symm
```

Notice that the second implicit parameter to `Eq.subst`, which provides the context in which the substitution is to occur, has type `α → Prop`. Inferring this predicate therefore requires an instance of *higher-order unification*. In full generality, the problem of determining whether a higher-order unifier exists is undecidable, and Lean can at best provide imperfect and approximate solutions to the problem. As a result, `Eq.subst` doesn't always do what you want it to. The macro `h ▸ e` uses more effective heuristics for computing this implicit parameter, and often succeeds in situations where applying `Eq.subst` fails.

Because equational reasoning is so common and important, Lean provides a number of mechanisms to carry it out more effectively. The next section offers syntax that allow you to write calculational proofs in a more natural and perspicuous way. But, more importantly, equational reasoning is supported by a term rewriter, a simplifier, and other kinds of automation. The term rewriter and simplifier are described briefly in the next section, and then in greater detail in the next chapter.

Calculational Proofs

A calculational proof is just a chain of intermediate results that are meant to be composed by basic principles such as the transitivity of equality. In Lean, a calculational proof starts with the keyword `calc`, and has the following syntax:

```
calc
  <expr>_0  'op_1'  <expr>_1  ':='  <proof>_1
  '_'       'op_2'  <expr>_2  ':='  <proof>_2
  ...
  '_'       'op_n'  <expr>_n  ':='  <proof>_n
```

Note that the `calc` relations all have the same indentation. Each `<proof>_i` is a proof for `<expr>_{i-1} op_i <expr>_i`.

We can also use `_` in the first relation (right after `<expr>_0`) which is useful to align the sequence of relation/proof pairs:

```
calc <expr>_0
    '_' 'op_1' <expr>_1 ':=' <proof>_1
    '_' 'op_2' <expr>_2 ':=' <proof>_2
    ...
    '_' 'op_n' <expr>_n ':=' <proof>_n
```

Here is an example:

```
variable (a b c d e : Nat)
variable (h1 : a = b)
variable (h2 : b = c + 1)
variable (h3 : c = d)
variable (h4 : e = 1 + d)

theorem T : a = e :=
  calc
    a = b      := h1
    _ = c + 1  := h2
    _ = d + 1  := congrArg Nat.succ h3
    _ = 1 + d  := Nat.add_comm d 1
    _ = e      := Eq.symm h4
```

This style of writing proofs is most effective when it is used in conjunction with the `simp` and `rewrite` tactics, which are discussed in greater detail in the next chapter. For example, using the abbreviation `rw` for rewrite, the proof above could be written as follows:

```
theorem T : a = e :=
  calc
    a = b      := by rw [h1]
    _ = c + 1  := by rw [h2]
    _ = d + 1  := by rw [h3]
    _ = 1 + d  := by rw [Nat.add_comm]
    _ = e      := by rw [h4]
```

Essentially, the `rw` tactic uses a given equality (which can be a hypothesis, a theorem name, or a complex term) to "rewrite" the goal. If doing so reduces the goal to an identity `t = t`, the tactic applies reflexivity to prove it.

Rewrites can be applied sequentially, so that the proof above can be shortened to this:

```
theorem T : a = e :=
  calc
    a = d + 1  := by rw [h1, h2, h3]
    _ = 1 + d  := by rw [Nat.add_comm]
    _ = e      := by rw [h4]
```

Or even this:

```
theorem T : a = e :=
  by rw [h1, h2, h3, Nat.add_comm, h4]
```

The `simp` tactic, instead, rewrites the goal by applying the given identities repeatedly, in any order, anywhere they are applicable in a term. It also uses other rules that have been previously declared to the system, and applies commutativity wisely to avoid looping. As a result, we can also prove the theorem as follows:

```
theorem T : a = e :=
  by simp [h1, h2, h3, Nat.add_comm, h4]
```

We will discuss variations of `rw` and `simp` in the next chapter.

The `calc` command can be configured for any relation that supports some form of transitivity. It can even combine different relations.

```
example (a b c d : Nat) (h1 : a = b) (h2 : b ≤ c) (h3 : c + 1 < d) : a < d :=
  calc
    a = b     := h1
    _ < b + 1 := Nat.lt_succ_self b
    _ ≤ c + 1 := Nat.succ_le_succ h2
    _ < d     := h3
```

You can "teach" `calc` new transitivity theorems by adding new instances of the `Trans` type class. Type classes are introduced later, but the following small example demonstrates how to extend the `calc` notation using new `Trans` instances.

```
def divides (x y : Nat) : Prop :=
  ∃ k, k*x = y

def divides_trans (h₁ : divides x y) (h₂ : divides y z) : divides x z :=
  let ⟨k₁, d₁⟩ := h₁
  let ⟨k₂, d₂⟩ := h₂
  ⟨k₁ * k₂, by rw [Nat.mul_comm k₁ k₂, Nat.mul_assoc, d₁, d₂]⟩

def divides_mul (x : Nat) (k : Nat) : divides x (k*x) :=
  ⟨k, rfl⟩

instance : Trans divides divides divides where
  trans := divides_trans

example (h₁ : divides x y) (h₂ : y = z) : divides x (2*z) :=
  calc
    divides x y     := h₁
    _ = z           := h₂
    divides _ (2*z) := divides_mul ..

infix:50 " | " => divides

example (h₁ : divides x y) (h₂ : y = z) : divides x (2*z) :=
  calc
    x | y   := h₁
    _ = z   := h₂
    _ | 2*z := divides_mul ..
```

The example above also makes it clear that you can use `calc` even if you do not have an infix notation for your relation. Finally we remark that the vertical bar `|` in the example above is the unicode one. We use unicode to make sure we do not overload the ASCII `|` used in the `match .. with` expression.

With `calc`, we can write the proof in the last section in a more natural and perspicuous way.

```
example (x y : Nat) : (x + y) * (x + y) = x * x + y * x + x * y + y * y :=
  calc
    (x + y) * (x + y) = (x + y) * x + (x + y) * y  := by rw [Nat.mul_add]
    _ = x * x + y * x + (x + y) * y                := by rw [Nat.add_mul]
    _ = x * x + y * x + (x * y + y * y)            := by rw [Nat.add_mul]
    _ = x * x + y * x + x * y + y * y              := by rw [←Nat.add_assoc]
```

The alternative `calc` notation is worth considering here. When the first expression is taking this much space, using `_` in the first relation naturally aligns all relations:

```
example (x y : Nat) : (x + y) * (x + y) = x * x + y * x + x * y + y * y :=
  calc (x + y) * (x + y)
    _ = (x + y) * x + (x + y) * y       := by rw [Nat.mul_add]
    _ = x * x + y * x + (x + y) * y     := by rw [Nat.add_mul]
    _ = x * x + y * x + (x * y + y * y) := by rw [Nat.add_mul]
    _ = x * x + y * x + x * y + y * y   := by rw [←Nat.add_assoc]
```

Here the left arrow before `Nat.add_assoc` tells rewrite to use the identity in the opposite direction. (You can enter it with `\l` or use the ascii equivalent, `<-`.) If brevity is what we are after, both `rw` and `simp` can do the job on their own:

```
example (x y : Nat) : (x + y) * (x + y) = x * x + y * x + x * y + y * y :=
  by rw [Nat.mul_add, Nat.add_mul, Nat.add_mul, ←Nat.add_assoc]

example (x y : Nat) : (x + y) * (x + y) = x * x + y * x + x * y + y * y :=
  by simp [Nat.mul_add, Nat.add_mul, Nat.add_assoc]
```

The Existential Quantifier

Finally, consider the existential quantifier, which can be written as either `exists x : α, p x` or `∃ x : α, p x`. Both versions are actually notationally convenient abbreviations for a more long-winded expression, `Exists (fun x : α => p x)`, defined in Lean's library.

As you should by now expect, the library includes both an introduction rule and an elimination rule. The introduction rule is straightforward: to prove `∃ x : α, p x`, it suffices to provide a suitable term `t` and a proof of `p t`. Here are some examples:

```
example : ∃ x : Nat, x > 0 :=
  have h : 1 > 0 := Nat.zero_lt_succ 0
  Exists.intro 1 h

example (x : Nat) (h : x > 0) : ∃ y, y < x :=
  Exists.intro 0 h

example (x y z : Nat) (hxy : x < y) (hyz : y < z) : ∃ w, x < w ∧ w < z :=
  Exists.intro y (And.intro hxy hyz)

#check @Exists.intro -- ∀ {α : Sort u_1} {p : α → Prop} (w : α), p w → Exists p
```

We can use the anonymous constructor notation `⟨t, h⟩` for `Exists.intro t h`, when the type is clear from the context.

```
example : ∃ x : Nat, x > 0 :=
  have h : 1 > 0 := Nat.zero_lt_succ 0
  ⟨1, h⟩

example (x : Nat) (h : x > 0) : ∃ y, y < x :=
  ⟨0, h⟩

example (x y z : Nat) (hxy : x < y) (hyz : y < z) : ∃ w, x < w ∧ w < z :=
  ⟨y, hxy, hyz⟩
```

Note that `Exists.intro` has implicit arguments: Lean has to infer the predicate `p : α → Prop` in the conclusion `∃ x, p x`. This is not a trivial affair. For example, if we have `hg : g 0 0 = 0` and write `Exists.intro 0 hg`, there are many possible values for the predicate `p`,

corresponding to the theorems `∃ x, g x x = x`, `∃ x, g x x = 0`, `∃ x, g x 0 = x`, etc. Lean uses the context to infer which one is appropriate. This is illustrated in the following example, in which we set the option `pp.explicit` to true to ask Lean's pretty-printer to show the implicit arguments.

```
variable (g : Nat → Nat → Nat)
variable (hg : g 0 0 = 0)

theorem gex1 : ∃ x, g x x = x := ⟨0, hg⟩
theorem gex2 : ∃ x, g x 0 = x := ⟨0, hg⟩
theorem gex3 : ∃ x, g 0 0 = x := ⟨0, hg⟩
theorem gex4 : ∃ x, g x x = 0 := ⟨0, hg⟩

set_option pp.explicit true  -- display implicit arguments
#print gex1
#print gex2
#print gex3
#print gex4
```

We can view `Exists.intro` as an information-hiding operation, since it hides the witness to the body of the assertion. The existential elimination rule, `Exists.elim`, performs the opposite operation. It allows us to prove a proposition `q` from `∃ x : α, p x`, by showing that `q` follows from `p w` for an arbitrary value `w`. Roughly speaking, since we know there is an `x` satisfying `p x`, we can give it a name, say, `w`. If `q` does not mention `w`, then showing that `q` follows from `p w` is tantamount to showing that `q` follows from the existence of any such `x`. Here is an example:

```
variable (α : Type) (p q : α → Prop)

example (h : ∃ x, p x ∧ q x) : ∃ x, q x ∧ p x :=
  Exists.elim h
    (fun w =>
     fun hw : p w ∧ q w =>
     show ∃ x, q x ∧ p x from ⟨w, hw.right, hw.left⟩)
```

It may be helpful to compare the exists-elimination rule to the or-elimination rule: the assertion `∃ x : α, p x` can be thought of as a big disjunction of the propositions `p a`, as `a` ranges over all the elements of `α`. Note that the anonymous constructor notation `⟨w, hw.right, hw.left⟩` abbreviates a nested constructor application; we could equally well have written `⟨w, ⟨hw.right, hw.left⟩⟩`.

Notice that an existential proposition is very similar to a sigma type, as described in dependent types section. The difference is that given `a : α` and `h : p a`, the term `Exists.intro a h` has type `(∃ x : α, p x) : Prop` and `Sigma.mk a h` has type `(Σ x : α, p x) : Type`. The similarity between `∃` and `Σ` is another instance of the Curry-Howard isomorphism.

Lean provides a more convenient way to eliminate from an existential quantifier with the `match` expression:

```
variable (α : Type) (p q : α → Prop)

example (h : ∃ x, p x ∧ q x) : ∃ x, q x ∧ p x :=
  match h with
  | ⟨w, hw⟩ => ⟨w, hw.right, hw.left⟩
```

The `match` expression is part of Lean's function definition system, which provides convenient and expressive ways of defining complex functions. Once again, it is the Curry-Howard isomorphism that allows us to co-opt this mechanism for writing proofs as well. The `match` statement "destructs" the existential assertion into the components `w` and `hw`, which can then be used in the body of the statement to prove the proposition. We can annotate the types used in the match for greater clarity:

```
example (h : ∃ x, p x ∧ q x) : ∃ x, q x ∧ p x :=
  match h with
  | ⟨(w : α), (hw : p w ∧ q w)⟩ => ⟨w, hw.right, hw.left⟩
```

We can even use the match statement to decompose the conjunction at the same time:

```
example (h : ∃ x, p x ∧ q x) : ∃ x, q x ∧ p x :=
  match h with
  | ⟨w, hpw, hqw⟩ => ⟨w, hqw, hpw⟩
```

Lean also provides a pattern-matching `let` expression:

```
example (h : ∃ x, p x ∧ q x) : ∃ x, q x ∧ p x :=
  let ⟨w, hpw, hqw⟩ := h
  ⟨w, hqw, hpw⟩
```

This is essentially just alternative notation for the `match` construct above. Lean will even allow us to use an implicit `match` in the `fun` expression:

```
example : (∃ x, p x ∧ q x) → ∃ x, q x ∧ p x :=
  fun ⟨w, hpw, hqw⟩ => ⟨w, hqw, hpw⟩
```

We will see in Chapter Induction and Recursion that all these variations are instances of a more general pattern-matching construct.

In the following example, we define `is_even a` as `∃ b, a = 2 * b`, and then we show that the sum of two even numbers is an even number.

```
def is_even (a : Nat) := ∃ b, a = 2 * b

theorem even_plus_even (h1 : is_even a) (h2 : is_even b) : is_even (a + b) :=
  Exists.elim h1 (fun w1 (hw1 : a = 2 * w1) =>
  Exists.elim h2 (fun w2 (hw2 : b = 2 * w2) =>
    Exists.intro (w1 + w2)
      (calc a + b
        _ = 2 * w1 + 2 * w2 := by rw [hw1, hw2]
        _ = 2 * (w1 + w2)   := by rw [Nat.mul_add]))))
```

Using the various gadgets described in this chapter --- the match statement, anonymous constructors, and the `rewrite` tactic, we can write this proof concisely as follows:

```
theorem even_plus_even (h1 : is_even a) (h2 : is_even b) : is_even (a + b) :=
  match h1, h2 with
  | ⟨w1, hw1⟩, ⟨w2, hw2⟩ => ⟨w1 + w2, by rw [hw1, hw2, Nat.mul_add]⟩
```

Just as the constructive "or" is stronger than the classical "or," so, too, is the constructive "exists" stronger than the classical "exists". For example, the following implication requires classical reasoning because, from a constructive standpoint, knowing that it is not the case that every `x` satisfies `¬ p` is not the same as having a particular `x` that satisfies `p`.

```
open Classical
variable (p : α → Prop)

example (h : ¬ ∀ x, ¬ p x) : ∃ x, p x :=
  byContradiction
    (fun h1 : ¬ ∃ x, p x =>
      have h2 : ∀ x, ¬ p x :=
        fun x =>
        fun h3 : p x =>
        have h4 : ∃ x, p x := ⟨x, h3⟩
        show False from h1 h4
      show False from h h2)
```

What follows are some common identities involving the existential quantifier. In the exercises below, we encourage you to prove as many as you can. We also leave it to you to determine which are nonconstructive, and hence require some form of classical reasoning.

```
open Classical

variable (α : Type) (p q : α → Prop)
variable (r : Prop)

example : (∃ x : α, r) → r := sorry
example (a : α) : r → (∃ x : α, r) := sorry
example : (∃ x, p x ∧ r) ↔ (∃ x, p x) ∧ r := sorry
example : (∃ x, p x ∨ q x) ↔ (∃ x, p x) ∨ (∃ x, q x) := sorry

example : (∀ x, p x) ↔ ¬ (∃ x, ¬ p x) := sorry
example : (∃ x, p x) ↔ ¬ (∀ x, ¬ p x) := sorry
example : (¬ ∃ x, p x) ↔ (∀ x, ¬ p x) := sorry
example : (¬ ∀ x, p x) ↔ (∃ x, ¬ p x) := sorry

example : (∀ x, p x → r) ↔ (∃ x, p x) → r := sorry
example (a : α) : (∃ x, p x → r) ↔ (∀ x, p x) → r := sorry
example (a : α) : (∃ x, r → p x) ↔ (r → ∃ x, p x) := sorry
```

Notice that the second example and the last two examples require the assumption that there is at least one element `a` of type `α`.

Here are solutions to two of the more difficult ones:

```
open Classical

variable (α : Type) (p q : α → Prop)
variable (a : α)
variable (r : Prop)

example : (∃ x, p x ∨ q x) ↔ (∃ x, p x) ∨ (∃ x, q x) :=
  Iff.intro
    (fun ⟨a, (h1 : p a ∨ q a)⟩ =>
      Or.elim h1
        (fun hpa : p a => Or.inl ⟨a, hpa⟩)
        (fun hqa : q a => Or.inr ⟨a, hqa⟩))
    (fun h : (∃ x, p x) ∨ (∃ x, q x) =>
      Or.elim h
        (fun ⟨a, hpa⟩ => ⟨a, (Or.inl hpa)⟩)
        (fun ⟨a, hqa⟩ => ⟨a, (Or.inr hqa)⟩))

example : (∃ x, p x → r) ↔ (∀ x, p x) → r :=
  Iff.intro
    (fun ⟨b, (hb : p b → r)⟩ =>
     fun h2 : ∀ x, p x =>
     show r from hb (h2 b))
    (fun h1 : (∀ x, p x) → r =>
     show ∃ x, p x → r from
       byCases
         (fun hap : ∀ x, p x => ⟨a, λ h' => h1 hap⟩)
         (fun hnap : ¬ ∀ x, p x =>
          byContradiction
            (fun hnex : ¬ ∃ x, p x → r =>
              have hap : ∀ x, p x :=
                fun x =>
                byContradiction
                  (fun hnp : ¬ p x =>
                    have hex : ∃ x, p x → r := ⟨x, (fun hp => absurd hp hnp)⟩
                    show False from hnex hex)
              show False from hnap hap)))
```

More on the Proof Language

We have seen that keywords like `fun`, `have`, and `show` make it possible to write formal proof terms that mirror the structure of informal mathematical proofs. In this section, we discuss some additional features of the proof language that are often convenient.

To start with, we can use anonymous "have" expressions to introduce an auxiliary goal without having to label it. We can refer to the last expression introduced in this way using the keyword `this`:

```
variable (f : Nat → Nat)
variable (h : ∀ x : Nat, f x ≤ f (x + 1))

example : f 0 ≤ f 3 :=
  have : f 0 ≤ f 1 := h 0
  have : f 0 ≤ f 2 := Nat.le_trans this (h 1)
  show f 0 ≤ f 3 from Nat.le_trans this (h 2)
```

Often proofs move from one fact to the next, so this can be effective in eliminating the clutter of lots of labels.

When the goal can be inferred, we can also ask Lean instead to fill in the proof by writing `by assumption`:

```
example : f 0 ≤ f 3 :=
  have : f 0 ≤ f 1 := h 0
  have : f 0 ≤ f 2 := Nat.le_trans (by assumption) (h 1)
  show f 0 ≤ f 3 from Nat.le_trans (by assumption) (h 2)
```

This tells Lean to use the `assumption` tactic, which, in turn, proves the goal by finding a suitable hypothesis in the local context. We will learn more about the `assumption` tactic in the next chapter.

We can also ask Lean to fill in the proof by writing `‹p›`, where `p` is the proposition whose proof we want Lean to find in the context. You can type these corner quotes using `\f<` and `\f>`, respectively. The letter "f" is for "French," since the unicode symbols can also be used as French quotation marks. In fact, the notation is defined in Lean as follows:

```
notation "‹" p "›" => show p by assumption
```

This approach is more robust than using `by assumption`, because the type of the assumption that needs to be inferred is given explicitly. It also makes proofs more readable. Here is a more elaborate example:

```
variable (f : Nat → Nat)
variable (h : ∀ x : Nat, f x ≤ f (x + 1))

example : f 0 ≥ f 1 → f 1 ≥ f 2 → f 0 = f 2 :=
  fun _ : f 0 ≥ f 1 =>
  fun _ : f 1 ≥ f 2 =>
  have : f 0 ≥ f 2 := Nat.le_trans ‹f 1 ≥ f 2› ‹f 0 ≥ f 1›
  have : f 0 ≤ f 2 := Nat.le_trans (h 0) (h 1)
  show f 0 = f 2 from Nat.le_antisymm this ‹f 0 ≥ f 2›
```

Keep in mind that you can use the French quotation marks in this way to refer to *anything* in the context, not just things that were introduced anonymously. Its use is also not limited to propositions, though using it for data is somewhat odd:

```
example (n : Nat) : Nat := ‹Nat›
```

Later, we show how you can extend the proof language using the Lean macro system.

Exercises

1. Prove these equivalences:

```
variable (α : Type) (p q : α → Prop)

example : (∀ x, p x ∧ q x) ↔ (∀ x, p x) ∧ (∀ x, q x) := sorry
example : (∀ x, p x → q x) → (∀ x, p x) → (∀ x, q x) := sorry
example : (∀ x, p x) ∨ (∀ x, q x) → ∀ x, p x ∨ q x := sorry
```

You should also try to understand why the reverse implication is not derivable in the last example.

2. It is often possible to bring a component of a formula outside a universal quantifier, when it does not depend on the quantified variable. Try proving these (one direction of the second of these requires classical logic):

```
variable (α : Type) (p q : α → Prop)
variable (r : Prop)

example : α → ((∀ x : α, r) ↔ r) := sorry
example : (∀ x, p x ∨ r) ↔ (∀ x, p x) ∨ r := sorry
example : (∀ x, r → p x) ↔ (r → ∀ x, p x) := sorry
```

3. Consider the "barber paradox," that is, the claim that in a certain town there is a (male) barber that shaves all and only the men who do not shave themselves. Prove that this is a contradiction:

```
variable (men : Type) (barber : men)
variable (shaves : men → men → Prop)

example (h : ∀ x : men, shaves barber x ↔ ¬ shaves x x) : False := sorry
```

4. Remember that, without any parameters, an expression of type `Prop` is just an assertion. Fill in the definitions of `prime` and `Fermat_prime` below, and construct each of the given assertions. For example, you can say that there are infinitely many primes by asserting that for every natural number `n`, there is a prime number greater than `n`. Goldbach's weak conjecture states that every odd number greater than 5 is the sum of three primes. Look up the definition of a Fermat prime or any of the other statements, if necessary.

```
def even (n : Nat) : Prop := sorry

def prime (n : Nat) : Prop := sorry

def infinitely_many_primes : Prop := sorry

def Fermat_prime (n : Nat) : Prop := sorry

def infinitely_many_Fermat_primes : Prop := sorry

def goldbach_conjecture : Prop := sorry

def Goldbach's_weak_conjecture : Prop := sorry

def Fermat's_last_theorem : Prop := sorry
```

5. Prove as many of the identities listed in the Existential Quantifier section as you can.

Tactics

In this chapter, we describe an alternative approach to constructing proofs, using *tactics*. A proof term is a representation of a mathematical proof; tactics are commands, or instructions, that describe how to build such a proof. Informally, you might begin a mathematical proof by saying "to prove the forward direction, unfold the definition, apply the previous lemma, and simplify." Just as these are instructions that tell the reader how to find the relevant proof, tactics are instructions that tell Lean how to construct a proof term. They naturally support an incremental style of writing proofs, in which you decompose a proof and work on goals one step at a time.

We will describe proofs that consist of sequences of tactics as "tactic-style" proofs, to contrast with the ways of writing proof terms we have seen so far, which we will call "term-style" proofs. Each style has its own advantages and disadvantages. For example, tactic-style proofs can be harder to read, because they require the reader to predict or guess the results of each instruction. But they can also be shorter and easier to write. Moreover, tactics offer a gateway to using Lean's automation, since automated procedures are themselves tactics.

Entering Tactic Mode

Conceptually, stating a theorem or introducing a `have` statement creates a goal, namely, the goal of constructing a term with the expected type. For example, the following creates the goal of constructing a term of type `p ∧ q ∧ p`, in a context with constants `p q : Prop`, `hp : p` and `hq : q`:

```
theorem test (p q : Prop) (hp : p) (hq : q) : p ∧ q ∧ p :=
  sorry
```

You can write this goal as follows:

```
    p : Prop, q : Prop, hp : p, hq : q ⊢ p ∧ q ∧ p
```

Indeed, if you replace the "sorry" by an underscore in the example above, Lean will report that it is exactly this goal that has been left unsolved.

Ordinarily, you meet such a goal by writing an explicit term. But wherever a term is expected, Lean allows us to insert instead a `by <tactics>` block, where `<tactics>` is a sequence of commands, separated by semicolons or line breaks. You can prove the theorem above in that way:

```
theorem test (p q : Prop) (hp : p) (hq : q) : p ∧ q ∧ p :=
  by apply And.intro
     exact hp
     apply And.intro
     exact hq
     exact hp
```

We often put the `by` keyword on the preceding line, and write the example above as:

```
theorem test (p q : Prop) (hp : p) (hq : q) : p ∧ q ∧ p := by
  apply And.intro
  exact hp
  apply And.intro
  exact hq
  exact hp
```

The `apply` tactic applies an expression, viewed as denoting a function with zero or more arguments. It unifies the conclusion with the expression in the current goal, and creates new goals for the remaining arguments, provided that no later arguments depend on them. In the example above, the command `apply And.intro` yields two subgoals:

```
    case left
    p q : Prop
    hp : p
    hq : q
    ⊢ p

    case right
    p q : Prop
    hp : p
    hq : q
    ⊢ q ∧ p
```

The first goal is met with the command `exact hp`. The `exact` command is just a variant of `apply` which signals that the expression given should fill the goal exactly. It is good form to use it in a tactic proof, since its failure signals that something has gone wrong. It is also more robust than `apply`, since the elaborator takes the expected type, given by the target of the

goal, into account when processing the expression that is being applied. In this case, however, `apply` would work just as well.

You can see the resulting proof term with the `#print` command:

```
#print test
```

You can write a tactic script incrementally. In VS Code, you can open a window to display messages by pressing `Ctrl-Shift-Enter`, and that window will then show you the current goal whenever the cursor is in a tactic block. In Emacs, you can see the goal at the end of any line by pressing `C-c C-g`, or see the remaining goal in an incomplete proof by putting the cursor after the first character of the last tactic. If the proof is incomplete, the token `by` is decorated with a red squiggly line, and the error message contains the remaining goals.

Tactic commands can take compound expressions, not just single identifiers. The following is a shorter version of the preceding proof:

```
theorem test (p q : Prop) (hp : p) (hq : q) : p ∧ q ∧ p := by
  apply And.intro hp
  exact And.intro hq hp
```

Unsurprisingly, it produces exactly the same proof term:

```
#print test
```

Multiple tactic applications can be written in a single line by concatenating with a semicolon.

```
theorem test (p q : Prop) (hp : p) (hq : q) : p ∧ q ∧ p := by
  apply And.intro hp; exact And.intro hq hp
```

Tactics that may produce multiple subgoals often tag them. For example, the tactic `apply And.intro` tagged the first subgoal as `left`, and the second as `right`. In the case of the `apply` tactic, the tags are inferred from the parameters' names used in the `And.intro` declaration. You can structure your tactics using the notation `case <tag> => <tactics>`. The following is a structured version of our first tactic proof in this chapter.

```
theorem test (p q : Prop) (hp : p) (hq : q) : p ∧ q ∧ p := by
  apply And.intro
  case left => exact hp
  case right =>
    apply And.intro
    case left => exact hq
    case right => exact hp
```

You can solve the subgoal `right` before `left` using the `case` notation:

```
theorem test (p q : Prop) (hp : p) (hq : q) : p ∧ q ∧ p := by
  apply And.intro
  case right =>
    apply And.intro
    case left => exact hq
    case right => exact hp
  case left => exact hp
```

Note that Lean hides the other goals inside the `case` block. We say it is "focusing" on the selected goal. Moreover, Lean flags an error if the selected goal is not fully solved at the end of the `case` block.

For simple subgoals, it may not be worth selecting a subgoal using its tag, but you may still want to structure the proof. Lean also provides the "bullet" notation `. <tactics>` (or `· <tactics>`) for structuring proofs:

```
theorem test (p q : Prop) (hp : p) (hq : q) : p ∧ q ∧ p := by
  apply And.intro
  . exact hp
  . apply And.intro
    . exact hq
    . exact hp
```

Basic Tactics

In addition to `apply` and `exact`, another useful tactic is `intro`, which introduces a hypothesis. What follows is an example of an identity from propositional logic that we proved in a previous chapter, now proved using tactics.

```
example (p q r : Prop) : p ∧ (q ∨ r) ↔ (p ∧ q) ∨ (p ∧ r) := by
  apply Iff.intro
  . intro h
    apply Or.elim (And.right h)
    . intro hq
      apply Or.inl
      apply And.intro
      . exact And.left h
      . exact hq
    . intro hr
      apply Or.inr
      apply And.intro
      . exact And.left h
      . exact hr
  . intro h
    apply Or.elim h
    . intro hpq
      apply And.intro
      . exact And.left hpq
      . apply Or.inl
        exact And.right hpq
    . intro hpr
      apply And.intro
      . exact And.left hpr
      . apply Or.inr
        exact And.right hpr
```

The `intro` command can more generally be used to introduce a variable of any type:

```
example (α : Type) : α → α := by
  intro a
  exact a

example (α : Type) : ∀ x : α, x = x := by
  intro x
  exact Eq.refl x
```

You can use it to introduce several variables:

```
example : ∀ a b c : Nat, a = b → a = c → c = b := by
  intro a b c h₁ h₂
  exact Eq.trans (Eq.symm h₂) h₁
```

As the `apply` tactic is a command for constructing function applications interactively, the `intro` tactic is a command for constructing function abstractions interactively (i.e., terms of the form `fun x => e`). As with lambda abstraction notation, the `intro` tactic allows us to use an implicit `match`.

```
example (α : Type) (p q : α → Prop) : (∃ x, p x ∧ q x) → ∃ x, q x ∧ p x := by
  intro ⟨w, hpw, hqw⟩
  exact ⟨w, hqw, hpw⟩
```

You can also provide multiple alternatives like in the `match` expression.

```
example (α : Type) (p q : α → Prop) : (∃ x, p x ∨ q x) → ∃ x, q x ∨ p x := by
  intro
  | ⟨w, Or.inl h⟩ => exact ⟨w, Or.inr h⟩
  | ⟨w, Or.inr h⟩ => exact ⟨w, Or.inl h⟩
```

The `intros` tactic can be used without any arguments, in which case, it chooses names and introduces as many variables as it can. You will see an example of this in a moment.

The `assumption` tactic looks through the assumptions in context of the current goal, and if there is one matching the conclusion, it applies it.

```
example (x y z w : Nat) (h₁ : x = y) (h₂ : y = z) (h₃ : z = w) : x = w := by
  apply Eq.trans h₁
  apply Eq.trans h₂
  assumption   -- applied h₃
```

It will unify metavariables in the conclusion if necessary:

```
example (x y z w : Nat) (h₁ : x = y) (h₂ : y = z) (h₃ : z = w) : x = w := by
  apply Eq.trans
  assumption      -- solves x = ?b with h₁
  apply Eq.trans
  assumption      -- solves y = ?h₂.b with h₂
  assumption      -- solves z = w with h₃
```

The following example uses the `intros` command to introduce the three variables and two hypotheses automatically:

```
example : ∀ a b c : Nat, a = b → a = c → c = b := by
  intros
  apply Eq.trans
  apply Eq.symm
  assumption
  assumption
```

Note that names automatically generated by Lean are inaccessible by default. The motivation is to ensure your tactic proofs do not rely on automatically generated names, and are consequently more robust. However, you can use the combinator `unhygienic` to disable this restriction.

```
example : ∀ a b c : Nat, a = b → a = c → c = b := by unhygienic
  intros
  apply Eq.trans
  apply Eq.symm
  exact a_2
  exact a_1
```

You can also use the `rename_i` tactic to rename the most recent inaccessible names in your context. In the following example, the tactic `rename_i h1 _ h2` renames two of the last three hypotheses in your context.

```
example : ∀ a b c d : Nat, a = b → a = d → a = c → c = b := by
  intros
  rename_i h1 _ h2
  apply Eq.trans
  apply Eq.symm
  exact h2
  exact h1
```

The `rfl` tactic is syntactic sugar for `exact rfl`:

```
example (y : Nat) : (fun x : Nat => 0) y = 0 :=
  by rfl
```

The `repeat` combinator can be used to apply a tactic several times:

```
example : ∀ a b c : Nat, a = b → a = c → c = b := by
  intros
  apply Eq.trans
  apply Eq.symm
  repeat assumption
```

Another tactic that is sometimes useful is the `revert` tactic, which is, in a sense, an inverse to `intro`:

```
example (x : Nat) : x = x := by
  revert x
  -- goal is ⊢ ∀ (x : Nat), x = x
  intro y
  -- goal is y : Nat ⊢ y = y
  rfl
```

Moving a hypothesis into the goal yields an implication:

```
example (x y : Nat) (h : x = y) : y = x := by
  revert h
  -- goal is x y : Nat ⊢ x = y → y = x
  intro h₁
  -- goal is x y : Nat, h₁ : x = y ⊢ y = x
  apply Eq.symm
  assumption
```

But `revert` is even more clever, in that it will revert not only an element of the context but also all the subsequent elements of the context that depend on it. For example, reverting `x` in the example above brings `h` along with it:

```
example (x y : Nat) (h : x = y) : y = x := by
  revert x
  -- goal is y : Nat ⊢ ∀ (x : Nat), x = y → y = x
  intros
  apply Eq.symm
  assumption
```

You can also revert multiple elements of the context at once:

```
example (x y : Nat) (h : x = y) : y = x := by
  revert x y
  -- goal is ⊢ ∀ (x y : Nat), x = y → y = x
  intros
  apply Eq.symm
  assumption
```

You can only `revert` an element of the local context, that is, a local variable or hypothesis. But you can replace an arbitrary expression in the goal by a fresh variable using the `generalize` tactic:

```
example : 3 = 3 := by
  generalize 3 = x
  -- goal is x : Nat ⊢ x = x
  revert x
  -- goal is ⊢ ∀ (x : Nat), x = x
  intro y
  -- goal is y : Nat ⊢ y = y
  rfl
```

The mnemonic in the notation above is that you are generalizing the goal by setting `3` to an arbitrary variable `x`. Be careful: not every generalization preserves the validity of the goal. Here, `generalize` replaces a goal that could be proved using `rfl` with one that is not provable:

```
example : 2 + 3 = 5 := by
  generalize 3 = x
  -- goal is x : Nat ⊢ 2 + x = 5
  admit
```

In this example, the `admit` tactic is the analogue of the `sorry` proof term. It closes the current goal, producing the usual warning that `sorry` has been used. To preserve the validity of the previous goal, the `generalize` tactic allows us to record the fact that `3` has been replaced by `x`. All you need to do is to provide a label, and `generalize` uses it to store the assignment in the local context:

```
example : 2 + 3 = 5 := by
  generalize h : 3 = x
  -- goal is x : Nat, h : 3 = x ⊢ 2 + x = 5
  rw [← h]
```

Here the `rewrite` tactic, abbreviated `rw`, uses `h` to replace `x` by `3` again. The `rewrite` tactic will be discussed below.

More Tactics

Some additional tactics are useful for constructing and destructing propositions and data. For example, when applied to a goal of the form `p ∨ q`, you use tactics such as `apply Or.inl` and `apply Or.inr`. Conversely, the `cases` tactic can be used to decompose a disjunction:

```
example (p q : Prop) : p ∨ q → q ∨ p := by
  intro h
  cases h with
  | inl hp => apply Or.inr; exact hp
  | inr hq => apply Or.inl; exact hq
```

Note that the syntax is similar to the one used in `match` expressions. The new subgoals can be solved in any order:

```
example (p q : Prop) : p ∨ q → q ∨ p := by
  intro h
  cases h with
  | inr hq => apply Or.inl; exact hq
  | inl hp => apply Or.inr; exact hp
```

You can also use a (unstructured) `cases` without the `with` and a tactic for each alternative:

```
example (p q : Prop) : p ∨ q → q ∨ p := by
  intro h
  cases h
  apply Or.inr
  assumption
  apply Or.inl
  assumption
```

The (unstructured) `cases` is particularly useful when you can close several subgoals using the same tactic:

```
example (p : Prop) : p ∨ p → p := by
  intro h
  cases h
  repeat assumption
```

You can also use the combinator `tac1 <;> tac2` to apply `tac2` to each subgoal produced by tactic `tac1`:

```
example (p : Prop) : p ∨ p → p := by
  intro h
  cases h <;> assumption
```

You can combine the unstructured `cases` tactic with the `case` and `.` notation:

```
example (p q : Prop) : p ∨ q → q ∨ p := by
  intro h
  cases h
  . apply Or.inr
    assumption
  . apply Or.inl
    assumption

example (p q : Prop) : p ∨ q → q ∨ p := by
  intro h
  cases h
  case inr h =>
    apply Or.inl
    assumption
  case inl h =>
    apply Or.inr
    assumption

example (p q : Prop) : p ∨ q → q ∨ p := by
  intro h
  cases h
  case inr h =>
    apply Or.inl
    assumption
  . apply Or.inr
    assumption
```

The `cases` tactic can also be used to decompose a conjunction:

```
example (p q : Prop) : p ∧ q → q ∧ p := by
  intro h
  cases h with
  | intro hp hq => constructor; exact hq; exact hp
```

In this example, there is only one goal after the `cases` tactic is applied, with `h : p ∧ q` replaced by a pair of assumptions, `hp : p` and `hq : q`. The `constructor` tactic applies the unique constructor for conjunction, `And.intro`.

With these tactics, an example from the previous section can be rewritten as follows:

```
example (p q r : Prop) : p ∧ (q ∨ r) ↔ (p ∧ q) ∨ (p ∧ r) := by
  apply Iff.intro
  . intro h
    cases h with
    | intro hp hqr =>
      cases hqr
      . apply Or.inl; constructor <;> assumption
      . apply Or.inr; constructor <;> assumption
  . intro h
    cases h with
    | inl hpq =>
      cases hpq with
      | intro hp hq => constructor; exact hp; apply Or.inl; exact hq
    | inr hpr =>
      cases hpr with
      | intro hp hr => constructor; exact hp; apply Or.inr; exact hr
```

You will see in Chapter Inductive Types that these tactics are quite general. The `cases` tactic can be used to decompose any element of an inductively defined type; `constructor` always applies the first applicable constructor of an inductively defined type. For example, you can use `cases` and `constructor` with an existential quantifier:

```
example (p q : Nat → Prop) : (∃ x, p x) → ∃ x, p x ∨ q x := by
  intro h
  cases h with
  | intro x px => constructor; apply Or.inl; exact px
```

Here, the `constructor` tactic leaves the first component of the existential assertion, the value of `x`, implicit. It is represented by a metavariable, which should be instantiated later on. In the previous example, the proper value of the metavariable is determined by the tactic `exact px`, since `px` has type `p x`. If you want to specify a witness to the existential quantifier explicitly, you can use the `exists` tactic instead:

```
example (p q : Nat → Prop) : (∃ x, p x) → ∃ x, p x ∨ q x := by
  intro h
  cases h with
  | intro x px => exists x; apply Or.inl; exact px
```

Here is another example:

```
example (p q : Nat → Prop) : (∃ x, p x ∧ q x) → ∃ x, q x ∧ p x := by
  intro h
  cases h with
  | intro x hpq =>
    cases hpq with
    | intro hp hq =>
      exists x
```

These tactics can be used on data just as well as propositions. In the next example, they are used to define functions which swap the components of the product and sum types:

```
def swap_pair : α × β → β × α := by
  intro p
  cases p
  constructor <;> assumption

def swap_sum : Sum α β → Sum β α := by
  intro p
  cases p
  . apply Sum.inr; assumption
  . apply Sum.inl; assumption
```

Note that up to the names we have chosen for the variables, the definitions are identical to the proofs of the analogous propositions for conjunction and disjunction. The `cases` tactic will also do a case distinction on a natural number:

```
open Nat
example (P : Nat → Prop) (h₀ : P 0) (h₁ : ∀ n, P (succ n)) (m : Nat) : P m := by
  cases m with
  | zero   => exact h₀
  | succ m' => exact h₁ m'
```

The `cases` tactic, and its companion, the `induction` tactic, are discussed in greater detail in the Tactics for Inductive Types section.

The `contradiction` tactic searches for a contradiction among the hypotheses of the current goal:

```
example (p q : Prop) : p ∧ ¬ p → q := by
  intro h
  cases h
  contradiction
```

You can also use `match` in tactic blocks.

```
example (p q r : Prop) : p ∧ (q ∨ r) ↔ (p ∧ q) ∨ (p ∧ r) := by
  apply Iff.intro
  . intro h
    match h with
    | ⟨_, Or.inl _⟩ => apply Or.inl; constructor <;> assumption
    | ⟨_, Or.inr _⟩ => apply Or.inr; constructor <;> assumption
  . intro h
    match h with
    | Or.inl ⟨hp, hq⟩ => constructor; exact hp; apply Or.inl; exact hq
    | Or.inr ⟨hp, hr⟩ => constructor; exact hp; apply Or.inr; exact hr
```

You can "combine" `intro h` with `match h ...` and write the previous examples as follows:

```
example (p q r : Prop) : p ∧ (q ∨ r) ↔ (p ∧ q) ∨ (p ∧ r) := by
  apply Iff.intro
  . intro
    | ⟨hp, Or.inl hq⟩ => apply Or.inl; constructor <;> assumption
    | ⟨hp, Or.inr hr⟩ => apply Or.inr; constructor <;> assumption
  . intro
    | Or.inl ⟨hp, hq⟩ => constructor; assumption; apply Or.inl; assumption
    | Or.inr ⟨hp, hr⟩ => constructor; assumption; apply Or.inr; assumption
```

Structuring Tactic Proofs

Tactics often provide an efficient way of building a proof, but long sequences of instructions can obscure the structure of the argument. In this section, we describe some means that help provide structure to a tactic-style proof, making such proofs more readable and robust.

One thing that is nice about Lean's proof-writing syntax is that it is possible to mix term-style and tactic-style proofs, and pass between the two freely. For example, the tactics `apply` and `exact` expect arbitrary terms, which you can write using `have`, `show`, and so on. Conversely, when writing an arbitrary Lean term, you can always invoke the tactic mode by inserting a `by` block. The following is a somewhat toy example:

```
example (p q r : Prop) : p ∧ (q ∨ r) → (p ∧ q) ∨ (p ∧ r) := by
  intro h
  exact
    have hp : p := h.left
    have hqr : q ∨ r := h.right
    show (p ∧ q) ∨ (p ∧ r) by
      cases hqr with
      | inl hq => exact Or.inl ⟨hp, hq⟩
      | inr hr => exact Or.inr ⟨hp, hr⟩
```

The following is a more natural example:

```
example (p q r : Prop) : p ∧ (q ∨ r) ↔ (p ∧ q) ∨ (p ∧ r) := by
  apply Iff.intro
  . intro h
    cases h.right with
    | inl hq => exact Or.inl ⟨h.left, hq⟩
    | inr hr => exact Or.inr ⟨h.left, hr⟩
  . intro h
    cases h with
    | inl hpq => exact ⟨hpq.left, Or.inl hpq.right⟩
    | inr hpr => exact ⟨hpr.left, Or.inr hpr.right⟩
```

In fact, there is a `show` tactic, which is similar to the `show` expression in a proof term. It simply declares the type of the goal that is about to be solved, while remaining in tactic mode.

```
example (p q r : Prop) : p ∧ (q ∨ r) ↔ (p ∧ q) ∨ (p ∧ r) := by
  apply Iff.intro
  . intro h
    cases h.right with
    | inl hq =>
      show (p ∧ q) ∨ (p ∧ r)
      exact Or.inl ⟨h.left, hq⟩
    | inr hr =>
      show (p ∧ q) ∨ (p ∧ r)
      exact Or.inr ⟨h.left, hr⟩
  . intro h
    cases h with
    | inl hpq =>
      show p ∧ (q ∨ r)
      exact ⟨hpq.left, Or.inl hpq.right⟩
    | inr hpr =>
      show p ∧ (q ∨ r)
      exact ⟨hpr.left, Or.inr hpr.right⟩
```

The `show` tactic can actually be used to rewrite a goal to something definitionally equivalent:

```
example (n : Nat) : n + 1 = Nat.succ n := by
  show Nat.succ n = Nat.succ n
  rfl
```

There is also a `have` tactic, which introduces a new subgoal, just as when writing proof terms:

```
example (p q r : Prop) : p ∧ (q ∨ r) → (p ∧ q) ∨ (p ∧ r) := by
  intro ⟨hp, hqr⟩
  show (p ∧ q) ∨ (p ∧ r)
  cases hqr with
  | inl hq =>
    have hpq : p ∧ q := And.intro hp hq
    apply Or.inl
    exact hpq
  | inr hr =>
    have hpr : p ∧ r := And.intro hp hr
    apply Or.inr
    exact hpr
```

As with proof terms, you can omit the label in the `have` tactic, in which case, the default label `this` is used:

```
example (p q r : Prop) : p ∧ (q ∨ r) → (p ∧ q) ∨ (p ∧ r) := by
  intro ⟨hp, hqr⟩
  show (p ∧ q) ∨ (p ∧ r)
  cases hqr with
  | inl hq =>
    have : p ∧ q := And.intro hp hq
    apply Or.inl
    exact this
  | inr hr =>
    have : p ∧ r := And.intro hp hr
    apply Or.inr
    exact this
```

The types in a `have` tactic can be omitted, so you can write `have hp := h.left` and `have hqr := h.right`. In fact, with this notation, you can even omit both the type and the label, in which case the new fact is introduced with the label `this`:

```
example (p q r : Prop) : p ∧ (q ∨ r) → (p ∧ q) ∨ (p ∧ r) := by
  intro ⟨hp, hqr⟩
  cases hqr with
  | inl hq =>
    have := And.intro hp hq
    apply Or.inl; exact this
  | inr hr =>
    have := And.intro hp hr
    apply Or.inr; exact this
```

Lean also has a `let` tactic, which is similar to the `have` tactic, but is used to introduce local definitions instead of auxiliary facts. It is the tactic analogue of a `let` in a proof term:

```
example : ∃ x, x + 2 = 8 := by
  let a : Nat := 3 * 2
  exists a
```

As with `have`, you can leave the type implicit by writing `let a := 3 * 2`. The difference between `let` and `have` is that `let` introduces a local definition in the context, so that the definition of the local declaration can be unfolded in the proof.

We have used `.` to create nested tactic blocks. In a nested block, Lean focuses on the first goal, and generates an error if it has not been fully solved at the end of the block. This can be helpful in indicating the separate proofs of multiple subgoals introduced by a tactic. The notation `.` is whitespace sensitive and relies on the indentation to detect whether the tactic block ends. Alternatively, you can define tactic blocks using curly braces and semicolons:

```
example (p q r : Prop) : p ∧ (q ∨ r) ↔ (p ∧ q) ∨ (p ∧ r) := by
  apply Iff.intro
  { intro h;
    cases h.right;
    { show (p ∧ q) ∨ (p ∧ r);
      exact Or.inl ⟨h.left, ‹q›⟩ }
    { show (p ∧ q) ∨ (p ∧ r);
      exact Or.inr ⟨h.left, ‹r›⟩ } }
  { intro h;
    cases h;
    { show p ∧ (q ∨ r);
      rename_i hpq;
      exact ⟨hpq.left, Or.inl hpq.right⟩ }
    { show p ∧ (q ∨ r);
      rename_i hpr;
      exact ⟨hpr.left, Or.inr hpr.right⟩ } }
```

It is useful to use indentation to structure proof: every time a tactic leaves more than one subgoal, we separate the remaining subgoals by enclosing them in blocks and indenting. Thus if the application of theorem `foo` to a single goal produces four subgoals, one would expect the proof to look like this:

```
  apply foo
  . <proof of first goal>
  . <proof of second goal>
  . <proof of third goal>
  . <proof of final goal>
```

or

```
  apply foo
  case <tag of first goal>  => <proof of first goal>
  case <tag of second goal> => <proof of second goal>
  case <tag of third goal>  => <proof of third goal>
  case <tag of final goal>  => <proof of final goal>
```

or

```
  apply foo
  { <proof of first goal>  }
  { <proof of second goal> }
  { <proof of third goal>  }
  { <proof of final goal>  }
```

Tactic Combinators

Tactic combinators are operations that form new tactics from old ones. A sequencing combinator is already implicit in the `by` block:

```
example (p q : Prop) (hp : p) : p ∨ q :=
  by apply Or.inl; assumption
```

Here, `apply Or.inl; assumption` is functionally equivalent to a single tactic which first applies `apply Or.inl` and then applies `assumption`.

In `t₁ <;> t₂`, the `<;>` operator provides a *parallel* version of the sequencing operation: `t₁` is applied to the current goal, and then `t₂` is applied to *all* the resulting subgoals:

```
example (p q : Prop) (hp : p) (hq : q) : p ∧ q :=
  by constructor <;> assumption
```

This is especially useful when the resulting goals can be finished off in a uniform way, or, at least, when it is possible to make progress on all of them uniformly.

The `first | t₁ | t₂ | ... | tₙ` applies each `tᵢ` until one succeeds, or else fails:

```
example (p q : Prop) (hp : p) : p ∨ q := by
  first | apply Or.inl; assumption | apply Or.inr; assumption

example (p q : Prop) (hq : q) : p ∨ q := by
  first | apply Or.inl; assumption | apply Or.inr; assumption
```

In the first example, the left branch succeeds, whereas in the second one, it is the right one that succeeds. In the next three examples, the same compound tactic succeeds in each case:

```
example (p q r : Prop) (hp : p) : p ∨ q ∨ r :=
  by repeat (first | apply Or.inl; assumption | apply Or.inr | assumption)

example (p q r : Prop) (hq : q) : p ∨ q ∨ r :=
  by repeat (first | apply Or.inl; assumption | apply Or.inr | assumption)

example (p q r : Prop) (hr : r) : p ∨ q ∨ r :=
  by repeat (first | apply Or.inl; assumption | apply Or.inr | assumption)
```

The tactic tries to solve the left disjunct immediately by assumption; if that fails, it tries to focus on the right disjunct; and if that doesn't work, it invokes the assumption tactic.

You will have no doubt noticed by now that tactics can fail. Indeed, it is the "failure" state that causes the *first* combinator to backtrack and try the next tactic. The `try` combinator builds a tactic that always succeeds, though possibly in a trivial way: `try t` executes `t` and reports success, even if `t` fails. It is equivalent to `first | t | skip`, where `skip` is a tactic that does nothing (and succeeds in doing so). In the next example, the second `constructor` succeeds on the right conjunct `q ∧ r` (remember that disjunction and conjunction associate to the right) but fails on the first. The `try` tactic ensures that the sequential composition succeeds:

```
example (p q r : Prop) (hp : p) (hq : q) (hr : r) : p ∧ q ∧ r := by
  constructor <;> (try constructor) <;> assumption
```

Be careful: `repeat (try t)` will loop forever, because the inner tactic never fails.

In a proof, there are often multiple goals outstanding. Parallel sequencing is one way to arrange it so that a single tactic is applied to multiple goals, but there are other ways to do this. For example, `all_goals t` applies `t` to all open goals:

```
example (p q r : Prop) (hp : p) (hq : q) (hr : r) : p ∧ q ∧ r := by
  constructor
  all_goals (try constructor)
  all_goals assumption
```

In this case, the `any_goals` tactic provides a more robust solution. It is similar to `all_goals`, except it succeeds if its argument succeeds on at least one goal:

```
example (p q r : Prop) (hp : p) (hq : q) (hr : r) : p ∧ q ∧ r := by
  constructor
  any_goals constructor
  any_goals assumption
```

The first tactic in the `by` block below repeatedly splits conjunctions:

```
example (p q r : Prop) (hp : p) (hq : q) (hr : r) :
      p ∧ ((p ∧ q) ∧ r) ∧ (q ∧ r ∧ p) := by
  repeat (any_goals constructor)
  all_goals assumption
```

In fact, we can compress the full tactic down to one line:

```
example (p q r : Prop) (hp : p) (hq : q) (hr : r) :
      p ∧ ((p ∧ q) ∧ r) ∧ (q ∧ r ∧ p) := by
  repeat (any_goals (first | constructor | assumption))
```

The combinator `focus t` ensures that `t` only effects the current goal, temporarily hiding the others from the scope. So, if `t` ordinarily only effects the current goal, `focus (all_goals t)` has the same effect as `t`.

Rewriting

The `rewrite` tactic (abbreviated `rw`) and the `simp` tactic were introduced briefly in Calculational Proofs. In this section and the next, we discuss them in greater detail.

The `rewrite` tactic provides a basic mechanism for applying substitutions to goals and hypotheses, providing a convenient and efficient way of working with equality. The most basic form of the tactic is `rewrite [t]`, where `t` is a term whose type asserts an equality. For example, `t` can be a hypothesis `h : x = y` in the context; it can be a general lemma, like `add_comm : ∀ x y, x + y = y + x`, in which the rewrite tactic tries to find suitable

instantiations of `x` and `y`; or it can be any compound term asserting a concrete or general equation. In the following example, we use this basic form to rewrite the goal using a hypothesis.

```
example (f : Nat → Nat) (k : Nat) (h₁ : f 0 = 0) (h₂ : k = 0) : f k = 0 := by
  rw [h₂] -- replace k with 0
  rw [h₁] -- replace f 0 with 0
```

In the example above, the first use of `rw` replaces `k` with `0` in the goal `f k = 0`. Then, the second one replaces `f 0` with `0`. The tactic automatically closes any goal of the form `t = t`. Here is an example of rewriting using a compound expression:

```
example (x y : Nat) (p : Nat → Prop) (q : Prop) (h : q → x = y)
        (h' : p y) (hq : q) : p x := by
  rw [h hq]; assumption
```

Here, `h hq` establishes the equation `x = y`.

Multiple rewrites can be combined using the notation `rw [t_1, ..., t_n]`, which is just shorthand for `rw [t_1]; ...; rw [t_n]`. The previous example can be written as follows:

```
example (f : Nat → Nat) (k : Nat) (h₁ : f 0 = 0) (h₂ : k = 0) : f k = 0 := by
  rw [h₂, h₁]
```

By default, `rw` uses an equation in the forward direction, matching the left-hand side with an expression, and replacing it with the right-hand side. The notation `←t` can be used to instruct the tactic to use the equality `t` in the reverse direction.

```
example (f : Nat → Nat) (a b : Nat) (h₁ : a = b) (h₂ : f a = 0) : f b = 0 := by
  rw [←h₁, h₂]
```

In this example, the term `←h₁` instructs the rewriter to replace `b` with `a`. In the editors, you can type the backwards arrow as `\l`. You can also use the ascii equivalent, `<-`.

Sometimes the left-hand side of an identity can match more than one subterm in the pattern, in which case the `rw` tactic chooses the first match it finds when traversing the term. If that is not the one you want, you can use additional arguments to specify the appropriate subterm.

```
example (a b c : Nat) : a + b + c = a + c + b := by
  rw [Nat.add_assoc, Nat.add_comm b, ← Nat.add_assoc]

example (a b c : Nat) : a + b + c = a + c + b := by
  rw [Nat.add_assoc, Nat.add_assoc, Nat.add_comm b]

example (a b c : Nat) : a + b + c = a + c + b := by
  rw [Nat.add_assoc, Nat.add_assoc, Nat.add_comm _ b]
```

In the first example above, the first step rewrites `a + b + c` to `a + (b + c)`. The next step applies commutativity to the term `b + c`; without specifying the argument, the tactic would instead rewrite `a + (b + c)` to `(b + c) + a`. Finally, the last step applies associativity in the reverse direction, rewriting `a + (c + b)` to `a + c + b`. The next two examples instead apply associativity to move the parenthesis to the right on both sides, and then switch `b` and `c`. Notice that the last example specifies that the rewrite should take place on the right-hand side by specifying the second argument to `Nat.add_comm`.

By default, the `rewrite` tactic affects only the goal. The notation `rw [t] at h` applies the rewrite `t` at hypothesis `h`.

```
example (f : Nat → Nat) (a : Nat) (h : a + 0 = 0) : f a = f 0 := by
  rw [Nat.add_zero] at h
  rw [h]
```

The first step, `rw [Nat.add_zero] at h`, rewrites the hypothesis `a + 0 = 0` to `a = 0`. Then the new hypothesis `a = 0` is used to rewrite the goal to `f 0 = f 0`.

The `rewrite` tactic is not restricted to propositions. In the following example, we use `rw [h] at t` to rewrite the hypothesis `t : Tuple α n` to `t : Tuple α 0`.

```
def Tuple (α : Type) (n : Nat) :=
  { as : List α // as.length = n }

example (n : Nat) (h : n = 0) (t : Tuple α n) : Tuple α 0 := by
  rw [h] at t
  exact t
```

Using the Simplifier

Whereas `rewrite` is designed as a surgical tool for manipulating a goal, the simplifier offers a more powerful form of automation. A number of identities in Lean's library have been tagged with the `[simp]` attribute, and the `simp` tactic uses them to iteratively rewrite subterms in an expression.

```
example (x y z : Nat) : (x + 0) * (0 + y * 1 + z * 0) = x * y := by
  simp

example (x y z : Nat) (p : Nat → Prop) (h : p (x * y))
        : p ((x + 0) * (0 + y * 1 + z * 0)) := by
  simp; assumption
```

In the first example, the left-hand side of the equality in the goal is simplified using the usual identities involving 0 and 1, reducing the goal to `x * y = x * y`. At that point, `simp` applies reflexivity to finish it off. In the second example, `simp` reduces the goal to `p (x * y)`, at which point the assumption `h` finishes it off. Here are some more examples with lists:

```
open List

example (xs : List Nat)
        : reverse (xs ++ [1, 2, 3]) = [3, 2, 1] ++ reverse xs := by
  simp

example (xs ys : List α)
        : length (reverse (xs ++ ys)) = length xs + length ys := by
  simp [Nat.add_comm]
```

As with `rw`, you can use the keyword `at` to simplify a hypothesis:

```
example (x y z : Nat) (p : Nat → Prop)
        (h : p ((x + 0) * (0 + y * 1 + z * 0))) : p (x * y) := by
  simp at h; assumption
```

Moreover, you can use a "wildcard" asterisk to simplify all the hypotheses and the goal:

```
attribute [local simp] Nat.mul_comm Nat.mul_assoc Nat.mul_left_comm
attribute [local simp] Nat.add_assoc Nat.add_comm Nat.add_left_comm

example (w x y z : Nat) (p : Nat → Prop)
        (h : p (x * y + z * w * x)) : p (x * w * z + y * x) := by
  simp at *; assumption

example (x y z : Nat) (p : Nat → Prop)
        (h₁ : p (1 * x + y)) (h₂ : p (x * z * 1))
        : p (y + 0 + x) ∧ p (z * x) := by
  simp at * <;> constructor <;> assumption
```

For operations that are commutative and associative, like multiplication on the natural numbers, the simplifier uses these two facts to rewrite an expression, as well as *left commutativity*. In the case of multiplication the latter is expressed as follows: `x * (y * z) = y * (x * z)`. The `local` modifier tells the simplifier to use these rules in the current file (or section or namespace, as the case may be). It may seem that commutativity and left-commutativity are problematic, in that repeated application of either causes looping. But the simplifier detects identities that permute their arguments, and uses a technique known as *ordered rewriting*. This means that the system maintains an internal ordering of terms, and only applies the identity if doing so decreases the order. With the three identities mentioned above, this has the effect that all the parentheses in an expression are associated to the right, and the expressions are ordered in a canonical (though somewhat arbitrary) way. Two expressions that are equivalent up to associativity and commutativity are then rewritten to the same canonical form.

```
example (w x y z : Nat) (p : Nat → Prop)
        : x * y + z * w * x = x * w * z + y * x := by
  simp

example (w x y z : Nat) (p : Nat → Prop)
        (h : p (x * y + z * w * x)) : p (x * w * z + y * x) := by
  simp; simp at h; assumption
```

As with `rewrite`, you can send `simp` a list of facts to use, including general lemmas, local hypotheses, definitions to unfold, and compound expressions. The `simp` tactic also recognizes the `←t` syntax that `rewrite` does. In any case, the additional rules are added to the collection of identities that are used to simplify a term.

```
def f (m n : Nat) : Nat :=
  m + n + m

example {m n : Nat} (h : n = 1) (h' : 0 = m) : (f m n) = n := by
  simp [h, ←h', f]
```

A common idiom is to simplify a goal using local hypotheses:

```
example (f : Nat → Nat) (k : Nat) (h₁ : f 0 = 0) (h₂ : k = 0) : f k = 0 := by
  simp [h₁, h₂]
```

To use all the hypotheses present in the local context when simplifying, we can use the wildcard symbol, `*`:

```
example (f : Nat → Nat) (k : Nat) (h₁ : f 0 = 0) (h₂ : k = 0) : f k = 0 := by
  simp [*]
```

Here is another example:

```
example (u w x y z : Nat) (h₁ : x = y + z) (h₂ : w = u + x)
        : w = z + y + u := by
  simp [*, Nat.add_assoc, Nat.add_comm, Nat.add_left_comm]
```

The simplifier will also do propositional rewriting. For example, using the hypothesis `p`, it rewrites `p ∧ q` to `q` and `p ∨ q` to `true`, which it then proves trivially. Iterating such rewrites produces nontrivial propositional reasoning.

```
example (p q : Prop) (hp : p) : p ∧ q ↔ q := by
  simp [*]

example (p q : Prop) (hp : p) : p ∨ q := by
  simp [*]

example (p q r : Prop) (hp : p) (hq : q) : p ∧ (q ∨ r) := by
  simp [*]
```

The next example simplifies all the hypotheses, and then uses them to prove the goal.

```
example (u w x x' y y' z : Nat) (p : Nat → Prop)
        (h₁ : x + 0 = x') (h₂ : y + 0 = y')
        : x + y + 0 = x' + y' := by
  simp at *
  simp [*]
```

One thing that makes the simplifier especially useful is that its capabilities can grow as a library develops. For example, suppose we define a list operation that symmetrizes its input by appending its reversal:

```
def mk_symm (xs : List α) :=
  xs ++ xs.reverse
```

Then for any list `xs`, `reverse (mk_symm xs)` is equal to `mk_symm xs`, which can easily be proved by unfolding the definition:

```
theorem reverse_mk_symm (xs : List α)
        : (mk_symm xs).reverse = mk_symm xs := by
  simp [mk_symm]
```

We can now use this theorem to prove new results:

```
example (xs ys : List Nat)
        : (xs ++ mk_symm ys).reverse = mk_symm ys ++ xs.reverse := by
  simp [reverse_mk_symm]

example (xs ys : List Nat) (p : List Nat → Prop)
        (h : p (xs ++ mk_symm ys).reverse)
        : p (mk_symm ys ++ xs.reverse) := by
  simp [reverse_mk_symm] at h; assumption
```

But using `reverse_mk_symm` is generally the right thing to do, and it would be nice if users did not have to invoke it explicitly. You can achieve that by marking it as a simplification rule when the theorem is defined:

```
@[simp] theorem reverse_mk_symm (xs : List α)
        : (mk_symm xs).reverse = mk_symm xs := by
  simp [mk_symm]

example (xs ys : List Nat)
        : (xs ++ mk_symm ys).reverse = mk_symm ys ++ xs.reverse := by
  simp

example (xs ys : List Nat) (p : List Nat → Prop)
        (h : p (xs ++ mk_symm ys).reverse)
        : p (mk_symm ys ++ xs.reverse) := by
  simp at h; assumption
```

The notation `@[simp]` declares `reverse_mk_symm` to have the `[simp]` attribute, and can be spelled out more explicitly:

```
theorem reverse_mk_symm (xs : List α)
        : (mk_symm xs).reverse = mk_symm xs := by
  simp [mk_symm]

attribute [simp] reverse_mk_symm

example (xs ys : List Nat)
        : (xs ++ mk_symm ys).reverse = mk_symm ys ++ xs.reverse := by
  simp

example (xs ys : List Nat) (p : List Nat → Prop)
        (h : p (xs ++ mk_symm ys).reverse)
        : p (mk_symm ys ++ xs.reverse) := by
  simp at h; assumption
```

The attribute can also be applied any time after the theorem is declared:

```
theorem reverse_mk_symm (xs : List α)
        : (mk_symm xs).reverse = mk_symm xs := by
  simp [mk_symm]

example (xs ys : List Nat)
        : (xs ++ mk_symm ys).reverse = mk_symm ys ++ xs.reverse := by
  simp [reverse_mk_symm]

attribute [simp] reverse_mk_symm

example (xs ys : List Nat) (p : List Nat → Prop)
        (h : p (xs ++ mk_symm ys).reverse)
        : p (mk_symm ys ++ xs.reverse) := by
  simp at h; assumption
```

Once the attribute is applied, however, there is no way to permanently remove it; it persists in any file that imports the one where the attribute is assigned. As we will discuss further in Attributes, one can limit the scope of an attribute to the current file or section using the `local` modifier:

```
theorem reverse_mk_symm (xs : List α)
        : (mk_symm xs).reverse = mk_symm xs := by
  simp [mk_symm]

section
attribute [local simp] reverse_mk_symm

example (xs ys : List Nat)
        : (xs ++ mk_symm ys).reverse = mk_symm ys ++ xs.reverse := by
  simp

example (xs ys : List Nat) (p : List Nat → Prop)
        (h : p (xs ++ mk_symm ys).reverse)
        : p (mk_symm ys ++ xs.reverse) := by
  simp at h; assumption
end
```

Outside the section, the simplifier will no longer use `reverse_mk_symm` by default.

Note that the various `simp` options we have discussed --- giving an explicit list of rules, and using `at` to specify the location --- can be combined, but the order they are listed is rigid. You can see the correct order in an editor by placing the cursor on the `simp` identifier to see the documentation string that is associated with it.

There are two additional modifiers that are useful. By default, `simp` includes all theorems that have been marked with the attribute `[simp]`. Writing `simp only` excludes these defaults, allowing you to use a more explicitly crafted list of rules. In the examples below, the minus sign and `only` are used to block the application of `reverse_mk_symm`.

```
def mk_symm (xs : List α) :=
  xs ++ xs.reverse
@[simp] theorem reverse_mk_symm (xs : List α)
        : (mk_symm xs).reverse = mk_symm xs := by
  simp [mk_symm]

example (xs ys : List Nat) (p : List Nat → Prop)
        (h : p (xs ++ mk_symm ys).reverse)
        : p (mk_symm ys ++ xs.reverse) := by
  simp at h; assumption

example (xs ys : List Nat) (p : List Nat → Prop)
        (h : p (xs ++ mk_symm ys).reverse)
        : p ((mk_symm ys).reverse ++ xs.reverse) := by
  simp [-reverse_mk_symm] at h; assumption

example (xs ys : List Nat) (p : List Nat → Prop)
        (h : p (xs ++ mk_symm ys).reverse)
        : p ((mk_symm ys).reverse ++ xs.reverse) := by
  simp only [List.reverse_append] at h; assumption
```

The `simp` tactic has many configuration options. For example, we can enable contextual simplifications as follows:

```
example : if x = 0 then y + x = y else x ≠ 0 := by
  simp (config := { contextual := true })
```

With `contextual := true`, the `simp` tactic uses the fact that `x = 0` when simplifying `y + x = y`, and `x ≠ 0` when simplifying the other branch. Here is another example:

```
example : ∀ (x : Nat) (h : x = 0), y + x = y := by
  simp (config := { contextual := true })
```

Another useful configuration option is `arith := true` which enables arithmetical simplifications. It is so useful that `simp_arith` is a shorthand for `simp (config := { arith := true })`:

```
example : 0 < 1 + x ∧ x + y + 2 ≥ y + 1 := by
  simp_arith
```

Split Tactic

The `split` tactic is useful for breaking nested `if-then-else` and `match` expressions in cases. For a `match` expression with `n` cases, the `split` tactic generates at most `n` subgoals. Here is an example:

```
def f (x y z : Nat) : Nat :=
  match x, y, z with
  | 5, _, _ => y
  | _, 5, _ => y
  | _, _, 5 => y
  | _, _, _ => 1

example (x y z : Nat) : x ≠ 5 → y ≠ 5 → z ≠ 5 → z = w → f x y w = 1 := by
  intros
  simp [f]
  split
  . contradiction
  . contradiction
  . contradiction
  . rfl
```

We can compress the tactic proof above as follows.

```
example (x y z : Nat) : x ≠ 5 → y ≠ 5 → z ≠ 5 → z = w → f x y w = 1 := by
  intros; simp [f]; split <;> first | contradiction | rfl
```

The tactic `split <;> first | contradiction | rfl` first applies the `split` tactic, and then for each generated goal it tries `contradiction`, and then `rfl` if `contradiction` fails. Like `simp`, we can apply `split` to a particular hypothesis:

```
def g (xs ys : List Nat) : Nat :=
  match xs, ys with
  | [a, b], _ => a+b+1
  | _, [b, c] => b+1
  | _, _      => 1

example (xs ys : List Nat) (h : g xs ys = 0) : False := by
  simp [g] at h; split at h <;> simp_arith at h
```

Extensible Tactics

In the following example, we define the notation `triv` using the command `syntax`. Then, we use the command `macro_rules` to specify what should be done when `triv` is used. You can provide different expansions, and the tactic interpreter will try all of them until one succeeds:

```
-- Define a new tactic notation
syntax "triv" : tactic

macro_rules
  | `(tactic| triv) => `(tactic| assumption)

example (h : p) : p := by
  triv

-- You cannot prove the following theorem using `triv`
-- example (x : α) : x = x := by
--  triv

-- Let's extend `triv`. The tactic interpreter
-- tries all possible macro extensions for `triv` until one succeeds
macro_rules
  | `(tactic| triv) => `(tactic| rfl)

example (x : α) : x = x := by
  triv

example (x : α) (h : p) : x = x ∧ p := by
  apply And.intro <;> triv

-- We now add a (recursive) extension
macro_rules | `(tactic| triv) => `(tactic| apply And.intro <;> triv)

example (x : α) (h : p) : x = x ∧ p := by
  triv
```

Exercises

1. Go back to the exercises in Chapter Propositions and Proofs and Chapter Quantifiers and Equality and redo as many as you can now with tactic proofs, using also `rw` and `simp` as appropriate.

2. Use tactic combinators to obtain a one line proof of the following:

```
example (p q r : Prop) (hp : p)
        : (p ∨ q ∨ r) ∧ (q ∨ p ∨ r) ∧ (q ∨ r ∨ p) := by
  admit
```

Interacting with Lean

You are now familiar with the fundamentals of dependent type theory, both as a language for defining mathematical objects and a language for constructing proofs. The one thing you are missing is a mechanism for defining new data types. We will fill this gap in the next chapter, which introduces the notion of an *inductive data type*. But first, in this chapter, we take a break from the mechanics of type theory to explore some pragmatic aspects of interacting with Lean.

Not all of the information found here will be useful to you right away. We recommend skimming this section to get a sense of Lean's features, and then returning to it as necessary.

Importing Files

The goal of Lean's front end is to interpret user input, construct formal expressions, and check that they are well-formed and type-correct. Lean also supports the use of various editors, which provide continuous checking and feedback. More information can be found on the Lean documentation pages.

The definitions and theorems in Lean's standard library are spread across multiple files. Users may also wish to make use of additional libraries, or develop their own projects across multiple files. When Lean starts, it automatically imports the contents of the library `Init` folder, which includes a number of fundamental definitions and constructions. As a result, most of the examples we present here work "out of the box."

If you want to use additional files, however, they need to be imported manually, via an `import` statement at the beginning of a file. The command

```
import Bar.Baz.Blah
```

imports the file `Bar/Baz/Blah.olean`, where the descriptions are interpreted relative to the Lean *search path*. Information as to how the search path is determined can be found on the documentation pages. By default, it includes the standard library directory, and (in some contexts) the root of the user's local project.

Importing is transitive. In other words, if you import `Foo` and `Foo` imports `Bar`, then you also have access to the contents of `Bar`, and do not need to import it explicitly.

More on Sections

Lean provides various sectioning mechanisms to help structure a theory. You saw in Variables and Sections that the `section` command makes it possible not only to group together elements of a theory that go together, but also to declare variables that are inserted as arguments to theorems and definitions, as necessary. Remember that the point of the `variable` command is to declare variables for use in theorems, as in the following example:

```
section
variable (x y : Nat)

def double := x + x

#check double y
#check double (2 * x)

attribute [local simp] Nat.add_assoc Nat.add_comm Nat.add_left_comm

theorem t1 : double (x + y) = double x + double y := by
  simp [double]

#check t1 y
#check t1 (2 * x)

theorem t2 : double (x * y) = double x * y := by
  simp [double, Nat.add_mul]

end
```

The definition of `double` does not have to declare `x` as an argument; Lean detects the dependence and inserts it automatically. Similarly, Lean detects the occurrence of `x` in `t1` and `t2`, and inserts it automatically there, too. Note that `double` does *not* have `y` as argument. Variables are only included in declarations where they are actually used.

More on Namespaces

In Lean, identifiers are given by hierarchical *names* like `Foo.Bar.baz`. We saw in Namespaces that Lean provides mechanisms for working with hierarchical names. The command `namespace foo` causes `foo` to be prepended to the name of each definition and theorem until `end foo` is encountered. The command `open foo` then creates temporary *aliases* to definitions and theorems that begin with prefix `foo`.

```
namespace Foo
def bar : Nat := 1
end Foo

open Foo

#check bar
#check Foo.bar
```

The following definition

```
def Foo.bar : Nat := 1
```

is treated as a macro, and expands to

```
namespace Foo
def bar : Nat := 1
end Foo
```

Although the names of theorems and definitions have to be unique, the aliases that identify them do not. When we open a namespace, an identifier may be ambiguous. Lean tries to use type information to disambiguate the meaning in context, but you can always disambiguate by giving the full name. To that end, the string `_root_` is an explicit description of the empty prefix.

```
def String.add (a b : String) : String :=
  a ++ b

def Bool.add (a b : Bool) : Bool :=
  a != b

def add (α β : Type) : Type := Sum α β

open Bool
open String
-- #check add -- ambiguous
#check String.add           -- String → String → String
#check Bool.add             -- Bool → Bool → Bool
#check _root_.add           -- Type → Type → Type

#check add "hello" "world"  -- String
#check add true false       -- Bool
#check add Nat Nat          -- Type
```

We can prevent the shorter alias from being created by using the `protected` keyword:

```
protected def Foo.bar : Nat := 1

open Foo

-- #check bar -- error
#check Foo.bar
```

This is often used for names like `Nat.rec` and `Nat.recOn`, to prevent overloading of common names.

The `open` command admits variations. The command

```
open Nat (succ zero gcd)
#check zero     -- Nat
#eval gcd 15 6  -- 3
```

creates aliases for only the identifiers listed. The command

```
open Nat hiding succ gcd
#check zero     -- Nat
-- #eval gcd 15 6  -- error
#eval Nat.gcd 15 6  -- 3
```

creates aliases for everything in the `Nat` namespace *except* the identifiers listed.

```
open Nat renaming mul → times, add → plus
#eval plus (times 2 2) 3  -- 7
```

creates aliases renaming `Nat.mul` to `times` and `Nat.add` to `plus`.

It is sometimes useful to `export` aliases from one namespace to another, or to the top level. The command

```
export Nat (succ add sub)
```

creates aliases for `succ`, `add`, and `sub` in the current namespace, so that whenever the namespace is open, these aliases are available. If this command is used outside a namespace, the aliases are exported to the top level.

Attributes

The main function of Lean is to translate user input to formal expressions that are checked by the kernel for correctness and then stored in the environment for later use. But some commands have other effects on the environment, either assigning attributes to objects in the environment, defining notation, or declaring instances of type classes, as described in Chapter Type Classes. Most of these commands have global effects, which is to say, they remain in effect not only in the current file, but also in any file that imports it. However, such commands often support the `local` modifier, which indicates that they only have effect until the current `section` or `namespace` is closed, or until the end of the current file.

In Section Using the Simplifier, we saw that theorems can be annotated with the `[simp]` attribute, which makes them available for use by the simplifier. The following example

defines the prefix relation on lists, proves that this relation is reflexive, and assigns the `[simp]` attribute to that theorem.

```
def isPrefix (l₁ : List α) (l₂ : List α) : Prop :=
  ∃ t, l₁ ++ t = l₂

@[simp] theorem List.isPrefix_self (as : List α) : isPrefix as as :=
  ⟨[], by simp⟩

example : isPrefix [1, 2, 3] [1, 2, 3] := by
  simp
```

The simplifier then proves `isPrefix [1, 2, 3] [1, 2, 3]` by rewriting it to `True`.

One can also assign the attribute any time after the definition takes place:

```
theorem List.isPrefix_self (as : List α) : isPrefix as as :=
  ⟨[], by simp⟩

attribute [simp] List.isPrefix_self
```

In all these cases, the attribute remains in effect in any file that imports the one in which the declaration occurs. Adding the `local` modifier restricts the scope:

```
section

theorem List.isPrefix_self (as : List α) : isPrefix as as :=
  ⟨[], by simp⟩

attribute [local simp] List.isPrefix_self

example : isPrefix [1, 2, 3] [1, 2, 3] := by
  simp

end

-- Error:
-- example : isPrefix [1, 2, 3] [1, 2, 3] := by
--   simp
```

For another example, we can use the `instance` command to assign the notation `≤` to the `isPrefix` relation. That command, which will be explained in Chapter Type Classes, works by assigning an `[instance]` attribute to the associated definition.

```
def isPrefix (l₁ : List α) (l₂ : List α) : Prop :=
  ∃ t, l₁ ++ t = l₂

instance : LE (List α) where
  le := isPrefix

theorem List.isPrefix_self (as : List α) : as ≤ as :=
  ⟨[], by simp⟩
```

That assignment can also be made local:

```
def instLe : LE (List α) :=
  { le := isPrefix }

section
attribute [local instance] instLe

example (as : List α) : as ≤ as :=
  ⟨[], by simp⟩

end

-- Error:
-- example (as : List α) : as ≤ as :=
--   ⟨[], by simp⟩
```

In Section Notation below, we will discuss Lean's mechanisms for defining notation, and see that they also support the `local` modifier. However, in Section Setting Options, we will discuss Lean's mechanisms for setting options, which does *not* follow this pattern: options can *only* be set locally, which is to say, their scope is always restricted to the current section or current file.

More on Implicit Arguments

In Section Implicit Arguments, we saw that if Lean displays the type of a term `t` as `{x : α} → β x`, then the curly brackets indicate that `x` has been marked as an *implicit argument* to `t`. This means that whenever you write `t`, a placeholder, or "hole," is inserted, so that `t` is replaced by `@t _`. If you don't want that to happen, you have to write `@t` instead.

Notice that implicit arguments are inserted eagerly. Suppose we define a function `f (x : Nat) {y : Nat} (z : Nat)` with the arguments shown. Then, when we write the expression `f 7` without further arguments, it is parsed as `f 7 _`. Lean offers a weaker annotation, `{{y : Nat}}`, which specifies that a placeholder should only be added *before* a subsequent explicit argument. This annotation can also be written using as `⦃y : Nat⦄`, where the unicode brackets are entered as `\{{` and `\}}`, respectively. With this annotation, the expression `f 7` would be parsed as is, whereas `f 7 3` would be parsed as `f 7 _ 3`, just as it would be with the strong annotation.

To illustrate the difference, consider the following example, which shows that a reflexive euclidean relation is both symmetric and transitive.

```
def reflexive {α : Type u} (r : α → α → Prop) : Prop :=
  ∀ (a : α), r a a

def symmetric {α : Type u} (r : α → α → Prop) : Prop :=
  ∀ {a b : α}, r a b → r b a

def transitive {α : Type u} (r : α → α → Prop) : Prop :=
  ∀ {a b c : α}, r a b → r b c → r a c

def euclidean {α : Type u} (r : α → α → Prop) : Prop :=
  ∀ {a b c : α}, r a b → r a c → r b c

theorem th1 {α : Type u} {r : α → α → Prop}
            (reflr : reflexive r) (euclr : euclidean r)
            : symmetric r :=
  fun {a b : α} =>
  fun (h : r a b) =>
  show r b a from euclr h (reflr _)

theorem th2 {α : Type u} {r : α → α → Prop}
            (symmr : symmetric r) (euclr : euclidean r)
            : transitive r :=
  fun {a b c : α} =>
  fun (rab : r a b) (rbc : r b c) =>
  euclr (symmr rab) rbc

theorem th3 {α : Type u} {r : α → α → Prop}
            (reflr : reflexive r) (euclr : euclidean r)
            : transitive r :=
 th2 (th1 reflr @euclr) @euclr

variable (r : α → α → Prop)
variable (euclr : euclidean r)

#check euclr  -- r ?m1 ?m2 → r ?m1 ?m3 → r ?m2 ?m3
```

The results are broken down into small steps: `th1` shows that a relation that is reflexive and euclidean is symmetric, and `th2` shows that a relation that is symmetric and euclidean is transitive. Then `th3` combines the two results. But notice that we have to manually disable the implicit arguments in `euclr`, because otherwise too many implicit arguments are inserted. The problem goes away if we use weak implicit arguments:

```
def reflexive {α : Type u} (r : α → α → Prop) : Prop :=
  ∀ (a : α), r a a

def symmetric {α : Type u} (r : α → α → Prop) : Prop :=
  ∀ {{a b : α}}, r a b → r b a

def transitive {α : Type u} (r : α → α → Prop) : Prop :=
  ∀ {{a b c : α}}, r a b → r b c → r a c

def euclidean {α : Type u} (r : α → α → Prop) : Prop :=
  ∀ {{a b c : α}}, r a b → r a c → r b c

theorem th1 {α : Type u} {r : α → α → Prop}
            (reflr : reflexive r) (euclr : euclidean r)
            : symmetric r :=
  fun {a b : α} =>
  fun (h : r a b) =>
  show r b a from euclr h (reflr _)

theorem th2 {α : Type u} {r : α → α → Prop}
            (symmr : symmetric r) (euclr : euclidean r)
            : transitive r :=
  fun {a b c : α} =>
  fun (rab : r a b) (rbc : r b c) =>
  euclr (symmr rab) rbc

theorem th3 {α : Type u} {r : α → α → Prop}
            (reflr : reflexive r) (euclr : euclidean r)
            : transitive r :=
  th2 (th1 reflr euclr) euclr

variable (r : α → α → Prop)
variable (euclr : euclidean r)

#check euclr  -- euclidean r
```

There is a third kind of implicit argument that is denoted with square brackets, `[` and `]`. These are used for type classes, as explained in Chapter Type Classes.

Notation

Identifiers in Lean can include any alphanumeric characters, including Greek characters (other than ∀ , Σ , and λ , which, as we have seen, have a special meaning in the dependent type theory). They can also include subscripts, which can be entered by typing `\_` followed by the desired subscripted character.

Lean's parser is extensible, which is to say, we can define new notation.

Lean's syntax can be extended and customized by users at every level, ranging from basic "mixfix" notations to custom elaborators. In fact, all builtin syntax is parsed and processed

using the same mechanisms and APIs open to users. In this section, we will describe and explain the various extension points.

While introducing new notations is a relatively rare feature in programming languages and sometimes even frowned upon because of its potential to obscure code, it is an invaluable tool in formalization for expressing established conventions and notations of the respective field succinctly in code. Going beyond basic notations, Lean's ability to factor out common boilerplate code into (well-behaved) macros and to embed entire custom domain specific languages (DSLs) to textually encode subproblems efficiently and readably can be of great benefit to both programmers and proof engineers alike.

Notations and Precedence

The most basic syntax extension commands allow introducing new (or overloading existing) prefix, infix, and postfix operators.

```
infixl:65   " + " => HAdd.hAdd  -- left-associative
infix:50    " = " => Eq         -- non-associative
infixr:80   " ^ " => HPow.hPow  -- right-associative
prefix:100  "-"   => Neg.neg
postfix:max "⁻¹"  => Inv.inv
```

After the initial command name describing the operator kind (its "fixity"), we give the *parsing precedence* of the operator preceded by a colon `:`, then a new or existing token surrounded by double quotes (the whitespace is used for pretty printing), then the function this operator should be translated to after the arrow `=>`.

The precedence is a natural number describing how "tightly" an operator binds to its arguments, encoding the order of operations. We can make this more precise by looking at the commands the above unfold to:

```
notation:65 lhs:65 " + " rhs:66 => HAdd.hAdd lhs rhs
notation:50 lhs:51 " = " rhs:51 => Eq lhs rhs
notation:80 lhs:81 " ^ " rhs:80 => HPow.hPow lhs rhs
notation:100 "-" arg:100 => Neg.neg arg
notation:1024 arg:1024 "⁻¹" => Inv.inv arg  -- `max` is a shorthand for
precedence 1024
```

It turns out that all commands from the first code block are in fact command *macros* translating to the more general `notation` command. We will learn about writing such macros below. Instead of a single token, the `notation` command accepts a mixed sequence of tokens and named term placeholders with precedences, which can be referenced on the right-hand side of `=>` and will be replaced by the respective term parsed at that position. A placeholder with precedence `p` accepts only notations with precedence at least `p` in that place. Thus the string `a + b + c` cannot be parsed as the equivalent of `a + (b + c)` because the right-hand side operand of an `infixl` notation has precedence one greater than the notation itself. In contrast, `infixr` reuses the notation's precedence for the right-

hand side operand, so `a ^ b ^ c` *can* be parsed as `a ^ (b ^ c)`. Note that if we used `notation` directly to introduce an infix notation like

```
notation:65 lhs:65 " ~ " rhs:65 => wobble lhs rhs
```

where the precedences do not sufficiently determine associativity, Lean's parser will default to right associativity. More precisely, Lean's parser follows a local *longest parse* rule in the presence of ambiguous grammars: when parsing the right-hand side of `a ~` in `a ~ b ~ c`, it will continue parsing as long as possible (as the current precedence allows), not stopping after `b` but parsing `~ c` as well. Thus the term is equivalent to `a ~ (b ~ c)`.

As mentioned above, the `notation` command allows us to define arbitrary *mixfix* syntax freely mixing tokens and placeholders.

```
notation:max "(" e ")" => e
notation:10 Γ " ⊢ " e " : " τ => Typing Γ e τ
```

Placeholders without precedence default to `0`, i.e. they accept notations of any precedence in their place. If two notations overlap, we again apply the longest parse rule:

```
notation:65 a " + " b:66 " + " c:66 => a + b - c
#eval 1 + 2 + 3  -- 0
```

The new notation is preferred to the binary notation since the latter, before chaining, would stop parsing after `1 + 2`. If there are multiple notations accepting the same longest parse, the choice will be delayed until elaboration, which will fail unless exactly one overload is type-correct.

Coercions

In Lean, the type of natural numbers, `Nat`, is different from the type of integers, `Int`. But there is a function `Int.ofNat` that embeds the natural numbers in the integers, meaning that we can view any natural number as an integer, when needed. Lean has mechanisms to detect and insert *coercions* of this sort.

```
variable (m n : Nat)
variable (i j : Int)

#check i + m      -- i + Int.ofNat m : Int
#check i + m + j  -- i + Int.ofNat m + j : Int
#check i + m + n  -- i + Int.ofNat m + Int.ofNat n : Int
```

Displaying Information

There are a number of ways in which you can query Lean for information about its current state and the objects and theorems that are available in the current context. You have already seen two of the most common ones, `#check` and `#eval`. Remember that `#check` is often used in conjunction with the `@` operator, which makes all of the arguments to a theorem or definition explicit. In addition, you can use the `#print` command to get information about any identifier. If the identifier denotes a definition or theorem, Lean prints the type of the symbol, and its definition. If it is a constant or an axiom, Lean indicates that fact, and shows the type.

```
-- examples with equality
#check Eq
#check @Eq
#check Eq.symm
#check @Eq.symm

#print Eq.symm

-- examples with And
#check And
#check And.intro
#check @And.intro

-- a user-defined function
def foo {α : Type u} (x : α) : α := x

#check foo
#check @foo
#print foo
```

Setting Options

Lean maintains a number of internal variables that can be set by users to control its behavior. The syntax for doing so is as follows:

```
set_option <name> <value>
```

One very useful family of options controls the way Lean's *pretty- printer* displays terms. The following options take an input of true or false:

```
pp.explicit  : display implicit arguments
pp.universes : display hidden universe parameters
pp.notation  : display output using defined notations
```

As an example, the following settings yield much longer output:

```
set_option pp.explicit true
set_option pp.universes true
set_option pp.notation false

#check 2 + 2 = 4
#reduce (fun x => x + 2) = (fun x => x + 3)
#check (fun x => x + 1) 1
```

The command `set_option pp.all true` carries out these settings all at once, whereas `set_option pp.all false` reverts to the previous values. Pretty printing additional information is often very useful when you are debugging a proof, or trying to understand a cryptic error message. Too much information can be overwhelming, though, and Lean's defaults are generally sufficient for ordinary interactions.

Using the Library

To use Lean effectively you will inevitably need to make use of definitions and theorems in the library. Recall that the `import` command at the beginning of a file imports previously compiled results from other files, and that importing is transitive; if you import `Foo` and `Foo` imports `Bar`, then the definitions and theorems from `Bar` are available to you as well. But the act of opening a namespace, which provides shorter names, does not carry over. In each file, you need to open the namespaces you wish to use.

In general, it is important for you to be familiar with the library and its contents, so you know what theorems, definitions, notations, and resources are available to you. Below we will see that Lean's editor modes can also help you find things you need, but studying the contents of the library directly is often unavoidable. Lean's standard library can be found online, on GitHub:

- https://github.com/leanprover/lean4/tree/master/src/Init
- https://github.com/leanprover/std4/tree/main/Std

You can see the contents of these directories and files using GitHub's browser interface. If you have installed Lean on your own computer, you can find the library in the `lean` folder, and explore it with your file manager. Comment headers at the top of each file provide additional information.

Lean's library developers follow general naming guidelines to make it easier to guess the name of a theorem you need, or to find it using tab completion in editors with a Lean mode that supports this, which is discussed in the next section. Identifiers are generally `camelCase`, and types are `CamelCase`. For theorem names, we rely on descriptive names where the different components are separated by `_`s. Often the name of theorem simply describes the conclusion:

```
#check Nat.succ_ne_zero
#check Nat.zero_add
#check Nat.mul_one
#check Nat.le_of_succ_le_succ
```

Remember that identifiers in Lean can be organized into hierarchical namespaces. For example, the theorem named `le_of_succ_le_succ` in the namespace `Nat` has full name `Nat.le_of_succ_le_succ`, but the shorter name is made available by the command `open Nat` (for names not marked as `protected`). We will see in Chapter Inductive Types and Chapter Structures and Records that defining structures and inductive data types in Lean generates associated operations, and these are stored in a namespace with the same name as the type under definition. For example, the product type comes with the following operations:

```
#check @Prod.mk
#check @Prod.fst
#check @Prod.snd
#check @Prod.rec
```

The first is used to construct a pair, whereas the next two, `Prod.fst` and `Prod.snd`, project the two elements. The last, `Prod.rec`, provides another mechanism for defining functions on a product in terms of a function on the two components. Names like `Prod.rec` are *protected*, which means that one has to use the full name even when the `Prod` namespace is open.

With the propositions as types correspondence, logical connectives are also instances of inductive types, and so we tend to use dot notation for them as well:

```
#check @And.intro
#check @And.casesOn
#check @And.left
#check @And.right
#check @Or.inl
#check @Or.inr
#check @Or.elim
#check @Exists.intro
#check @Exists.elim
#check @Eq.refl
#check @Eq.subst
```

Auto Bound Implicit Arguments

In the previous section, we have shown how implicit arguments make functions more convenient to use. However, functions such as `compose` are still quite verbose to define. Note that the universe polymorphic `compose` is even more verbose than the one previously defined.

```
universe u v w
def compose {α : Type u} {β : Type v} {γ : Type w}
            (g : β → γ) (f : α → β) (x : α) : γ :=
  g (f x)
```

You can avoid the `universe` command by providing the universe parameters when defining `compose`.

```
def compose.{u, v, w}
            {α : Type u} {β : Type v} {γ : Type w}
            (g : β → γ) (f : α → β) (x : α) : γ :=
  g (f x)
```

Lean 4 supports a new feature called *auto bound implicit arguments*. It makes functions such as `compose` much more convenient to write. When Lean processes the header of a declaration, any unbound identifier is automatically added as an implicit argument *if* it is a single lower case or greek letter. With this feature we can write `compose` as

```
def compose (g : β → γ) (f : α → β) (x : α) : γ :=
  g (f x)

#check @compose
-- {β : Sort u_1} → {γ : Sort u_2} → {α : Sort u_3} → (β → γ) → (α → β) → α → γ
```

Note that Lean inferred a more general type using `Sort` instead of `Type`.

Although we love this feature and use it extensively when implementing Lean, we realize some users may feel uncomfortable with it. Thus, you can disable it using the command `set_option autoImplicit false`.

```
set_option autoImplicit false
/- The following definition produces `unknown identifier` errors -/
-- def compose (g : β → γ) (f : α → β) (x : α) : γ :=
--   g (f x)
```

Implicit Lambdas

In Lean 3 stdlib, we find many instances of the dreadful `@` + `_` idiom. It is often used when the expected type is a function type with implicit arguments, and we have a constant (`reader_t.pure` in the example) which also takes implicit arguments. In Lean 4, the elaborator automatically introduces lambdas for consuming implicit arguments. We are still exploring this feature and analyzing its impact, but the experience so far has been very positive. Here is the example from the link above using Lean 4 implicit lambdas.

```
instance : Monad (ReaderT ρ m) where
  pure := ReaderT.pure
  bind := ReaderT.bind
```

Users can disable the implicit lambda feature by using `@` or writing a lambda expression with `{}` or `[]` binder annotations. Here are few examples

```
def id1 : {α : Type} → α → α :=
  fun x => x

def listId : List ({α : Type} → α → α) :=
  (fun x => x) :: []

-- In this example, implicit lambda introduction has been disabled because
-- we use `@` before `fun`
def id2 : {α : Type} → α → α :=
  @fun α (x : α) => id1 x

def id3 : {α : Type} → α → α :=
  @fun α x => id1 x

def id4 : {α : Type} → α → α :=
  fun x => id1 x

-- In this example, implicit lambda introduction has been disabled
-- because we used the binder annotation `{...}`
def id5 : {α : Type} → α → α :=
  fun {α} x => id1 x
```

Sugar for Simple Functions

In Lean 3, we can create simple functions from infix operators by using parentheses. For example, `(+1)` is sugar for `fun x, x + 1`. In Lean 4, we generalize this notation using `·` as a placeholder. Here are a few examples:

```
#check (· + 1)
-- fun a => a + 1
#check (2 - ·)
-- fun a => 2 - a
#eval [1, 2, 3, 4, 5].foldl (·*·) 1
-- 120

def f (x y z : Nat) :=
  x + y + z

#check (f · 1 ·)
-- fun a b => f a 1 b

#eval [(1, 2), (3, 4), (5, 6)].map (·.1)
-- [1, 3, 5]
```

As in Lean 3, the notation is activated using parentheses, and the lambda abstraction is created by collecting the nested `·`s. The collection is interrupted by nested parentheses. In the following example, two different lambda expressions are created.

```
#check (Prod.mk · (· + 1))
-- fun a => (a, fun b => b + 1)
```

Named Arguments

Named arguments enable you to specify an argument for a parameter by matching the argument with its name rather than with its position in the parameter list. If you don't remember the order of the parameters but know their names, you can send the arguments in any order. You may also provide the value for an implicit parameter when Lean failed to infer it. Named arguments also improve the readability of your code by identifying what each argument represents.

```
def sum (xs : List Nat) :=
  xs.foldl (init := 0) (·+·)

#eval sum [1, 2, 3, 4]
-- 10

example {a b : Nat} {p : Nat → Nat → Nat → Prop} (h₁ : p a b b) (h₂ : b = a)
    : p a a b :=
  Eq.subst (motive := fun x => p a x b) h₂ h₁
```

In the following examples, we illustrate the interaction between named and default arguments.

```
def f (x : Nat) (y : Nat := 1) (w : Nat := 2) (z : Nat) :=
  x + y + w - z

example (x z : Nat) : f (z := z) x = x + 1 + 2 - z := rfl

example (x z : Nat) : f x (z := z) = x + 1 + 2 - z := rfl

example (x y : Nat) : f x y = fun z => x + y + 2 - z := rfl

example : f = (fun x z => x + 1 + 2 - z) := rfl

example (x : Nat) : f x = fun z => x + 1 + 2 - z := rfl

example (y : Nat) : f (y := 5) = fun x z => x + 5 + 2 - z := rfl

def g {α} [Add α] (a : α) (b? : Option α := none) (c : α) : α :=
  match b? with
  | none   => a + c
  | some b => a + b + c

variable {α} [Add α]

example : g = fun (a c : α) => a + c := rfl

example (x : α) : g (c := x) = fun (a : α) => a + x := rfl

example (x : α) : g (b? := some x) = fun (a c : α) => a + x + c := rfl

example (x : α) : g x = fun (c : α) => x + c := rfl

example (x y : α) : g x y = fun (c : α) => x + y + c := rfl
```

You can use `..` to provide missing explicit arguments as `_`. This feature combined with named arguments is useful for writing patterns. Here is an example:

```
inductive Term where
  | var    (name : String)
  | num    (val : Nat)
  | app    (fn : Term) (arg : Term)
  | lambda (name : String) (type : Term) (body : Term)

def getBinderName : Term → Option String
  | Term.lambda (name := n) .. => some n
  | _ => none

def getBinderType : Term → Option Term
  | Term.lambda (type := t) .. => some t
  | _ => none
```

Ellipses are also useful when explicit arguments can be automatically inferred by Lean, and we want to avoid a sequence of `_`s.

```
example (f : Nat → Nat) (a b c : Nat) : f (a + b + c) = f (a + (b + c)) :=
  congrArg f (Nat.add_assoc ..)
```

Inductive Types

We have seen that Lean's formal foundation includes basic types, `Prop, Type 0, Type 1, Type 2, ...`, and allows for the formation of dependent function types, `(x : α) → β`. In the examples, we have also made use of additional types like `Bool`, `Nat`, and `Int`, and type constructors, like `List`, and product, `×`. In fact, in Lean's library, every concrete type other than the universes and every type constructor other than dependent arrows is an instance of a general family of type constructions known as *inductive types*. It is remarkable that it is possible to construct a substantial edifice of mathematics based on nothing more than the type universes, dependent arrow types, and inductive types; everything else follows from those.

Intuitively, an inductive type is built up from a specified list of constructors. In Lean, the syntax for specifying such a type is as follows:

```
inductive Foo where
  | constructor₁ : ... → Foo
  | constructor₂ : ... → Foo
  ...
  | constructorₙ : ... → Foo
```

The intuition is that each constructor specifies a way of building new objects of `Foo`, possibly from previously constructed values. The type `Foo` consists of nothing more than the objects that are constructed in this way. The first character `|` in an inductive declaration is optional. We can also separate constructors using a comma instead of `|`.

We will see below that the arguments of the constructors can include objects of type `Foo`, subject to a certain "positivity" constraint, which guarantees that elements of `Foo` are built from the bottom up. Roughly speaking, each `...` can be any arrow type constructed from `Foo` and previously defined types, in which `Foo` appears, if at all, only as the "target" of the dependent arrow type.

We will provide a number of examples of inductive types. We will also consider slight generalizations of the scheme above, to mutually defined inductive types, and so-called *inductive families*.

As with the logical connectives, every inductive type comes with introduction rules, which show how to construct an element of the type, and elimination rules, which show how to "use" an element of the type in another construction. The analogy to the logical connectives should not come as a surprise; as we will see below, they, too, are examples of inductive type constructions. You have already seen the introduction rules for an inductive type: they are just the constructors that are specified in the definition of the type. The elimination rules provide for a principle of recursion on the type, which includes, as a special case, a principle of induction as well.

In the next chapter, we will describe Lean's function definition package, which provides even more convenient ways to define functions on inductive types and carry out inductive proofs. But because the notion of an inductive type is so fundamental, we feel it is important to start with a low-level, hands-on understanding. We will start with some basic examples of inductive types, and work our way up to more elaborate and complex examples.

Enumerated Types

The simplest kind of inductive type is a type with a finite, enumerated list of elements.

```
inductive Weekday where
  | sunday : Weekday
  | monday : Weekday
  | tuesday : Weekday
  | wednesday : Weekday
  | thursday : Weekday
  | friday : Weekday
  | saturday : Weekday
```

The `inductive` command creates a new type, `Weekday`. The constructors all live in the `Weekday` namespace.

```
#check Weekday.sunday
#check Weekday.monday

open Weekday

#check sunday
#check monday
```

You can omit `: Weekday` when declaring the `Weekday` inductive type.

```
inductive Weekday where
  | sunday
  | monday
  | tuesday
  | wednesday
  | thursday
  | friday
  | saturday
```

Think of `sunday`, `monday`, ... , `saturday` as being distinct elements of `Weekday`, with no other distinguishing properties. The elimination principle, `Weekday.rec`, is defined along with the type `Weekday` and its constructors. It is also known as a *recursor*, and it is what makes the type "inductive": it allows us to define a function on `Weekday` by assigning values corresponding to each constructor. The intuition is that an inductive type is exhaustively generated by the constructors, and has no elements beyond those they construct.

We will use the `match` expression to define a function from `Weekday` to the natural numbers:

```
open Weekday

def numberOfDay (d : Weekday) : Nat :=
  match d with
  | sunday    => 1
  | monday    => 2
  | tuesday   => 3
  | wednesday => 4
  | thursday  => 5
  | friday    => 6
  | saturday  => 7

#eval numberOfDay Weekday.sunday  -- 1
#eval numberOfDay Weekday.monday  -- 2
#eval numberOfDay Weekday.tuesday -- 3
```

Note that the `match` expression is compiled using the *recursor* `Weekday.rec` generated when you declare the inductive type.

```
open Weekday

def numberOfDay (d : Weekday) : Nat :=
  match d with
  | sunday    => 1
  | monday    => 2
  | tuesday   => 3
  | wednesday => 4
  | thursday  => 5
  | friday    => 6
  | saturday  => 7

set_option pp.all true
#print numberOfDay
-- ... numberOfDay.match_1
#print numberOfDay.match_1
-- ... Weekday.casesOn ...
#print Weekday.casesOn
-- ... Weekday.rec ...
#check @Weekday.rec
/-
@Weekday.rec.{u}
 : {motive : Weekday → Sort u} →
    motive Weekday.sunday →
    motive Weekday.monday →
    motive Weekday.tuesday →
    motive Weekday.wednesday →
    motive Weekday.thursday →
    motive Weekday.friday →
    motive Weekday.saturday →
    (t : Weekday) → motive t
-/
```

When declaring an inductive datatype, you can use `deriving Repr` to instruct Lean to generate a function that converts `Weekday` objects into text. This function is used by the `#eval` command to display `Weekday` objects.

```
inductive Weekday where
  | sunday
  | monday
  | tuesday
  | wednesday
  | thursday
  | friday
  | saturday
  deriving Repr

open Weekday

#eval tuesday   -- Weekday.tuesday
```

It is often useful to group definitions and theorems related to a structure in a namespace with the same name. For example, we can put the `numberOfDay` function in the `Weekday` namespace. We are then allowed to use the shorter name when we open the namespace.

We can define functions from `Weekday` to `Weekday`:

```
namespace Weekday
def next (d : Weekday) : Weekday :=
  match d with
  | sunday    => monday
  | monday    => tuesday
  | tuesday   => wednesday
  | wednesday => thursday
  | thursday  => friday
  | friday    => saturday
  | saturday  => sunday

def previous (d : Weekday) : Weekday :=
  match d with
  | sunday    => saturday
  | monday    => sunday
  | tuesday   => monday
  | wednesday => tuesday
  | thursday  => wednesday
  | friday    => thursday
  | saturday  => friday

#eval next (next tuesday)      -- Weekday.thursday
#eval next (previous tuesday)  -- Weekday.tuesday

example : next (previous tuesday) = tuesday :=
  rfl

end Weekday
```

How can we prove the general theorem that `next (previous d) = d` for any Weekday `d`? You can use `match` to provide a proof of the claim for each constructor:

```
def next_previous (d : Weekday) : next (previous d) = d :=
  match d with
  | sunday    => rfl
  | monday    => rfl
  | tuesday   => rfl
  | wednesday => rfl
  | thursday  => rfl
  | friday    => rfl
  | saturday  => rfl
```

Using a tactic proof, we can be even more concise:

```
def next_previous (d : Weekday) : next (previous d) = d := by
  cases d <;> rfl
```

Tactics for Inductive Types below will introduce additional tactics that are specifically designed to make use of inductive types.

Notice that, under the propositions-as-types correspondence, we can use `match` to prove theorems as well as define functions. In other words, under the propositions-as-types correspondence, the proof by cases is a kind of definition by cases, where what is being "defined" is a proof instead of a piece of data.

The `Bool` type in the Lean library is an instance of enumerated type.

```
inductive Bool where
  | false : Bool
  | true  : Bool
```

(To run these examples, we put them in a namespace called `Hidden`, so that a name like `Bool` does not conflict with the `Bool` in the standard library. This is necessary because these types are part of the Lean "prelude" that is automatically imported when the system is started.)

As an exercise, you should think about what the introduction and elimination rules for these types do. As a further exercise, we suggest defining boolean operations `and`, `or`, `not` on the `Bool` type, and verifying common identities. Note that you can define a binary operation like `and` using `match`:

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

Similarly, most identities can be proved by introducing suitable `match`, and then using `rfl`.

Constructors with Arguments

Enumerated types are a very special case of inductive types, in which the constructors take no arguments at all. In general, a "construction" can depend on data, which is then represented in the constructed argument. Consider the definitions of the product type and sum type in the library:

```
inductive Prod (α : Type u) (β : Type v)
  | mk : α → β → Prod α β

inductive Sum (α : Type u) (β : Type v) where
  | inl : α → Sum α β
  | inr : β → Sum α β
```

Consider what is going on in these examples. The product type has one constructor, `Prod.mk`, which takes two arguments. To define a function on `Prod α β`, we can assume the input is of the form `Prod.mk a b`, and we have to specify the output, in terms of `a` and `b`. We can use this to define the two projections for `Prod`. Remember that the standard library defines notation `α × β` for `Prod α β` and `(a, b)` for `Prod.mk a b`.

```
def fst {α : Type u} {β : Type v} (p : Prod α β) : α :=
  match p with
  | Prod.mk a b => a

def snd {α : Type u} {β : Type v} (p : Prod α β) : β :=
  match p with
  | Prod.mk a b => b
```

The function `fst` takes a pair, `p`. The `match` interprets `p` as a pair, `Prod.mk a b`. Recall also from Dependent Type Theory that to give these definitions the greatest generality possible, we allow the types `α` and `β` to belong to any universe.

Here is another example where we use the recursor `Prod.casesOn` instead of `match`.

```
def prod_example (p : Bool × Nat) : Nat :=
  Prod.casesOn (motive := fun _ => Nat) p (fun b n => cond b (2 * n) (2 * n +
1))

#eval prod_example (true, 3)
#eval prod_example (false, 3)
```

The argument `motive` is used to specify the type of the object you want to construct, and it is a function because it may depend on the pair. The `cond` function is a boolean conditional: `cond b t1 t2` returns `t1` if `b` is true, and `t2` otherwise. The function `prod_example` takes a pair consisting of a boolean, `b`, and a number, `n`, and returns either `2 * n` or `2 * n + 1` according to whether `b` is true or false.

In contrast, the sum type has *two* constructors, `inl` and `inr` (for "insert left" and "insert right"), each of which takes *one* (explicit) argument. To define a function on `Sum α β`, we

have to handle two cases: either the input is of the form `inl a`, in which case we have to specify an output value in terms of `a`, or the input is of the form `inr b`, in which case we have to specify an output value in terms of `b`.

```
def sum_example (s : Sum Nat Nat) : Nat :=
  Sum.casesOn (motive := fun _ => Nat) s
    (fun n => 2 * n)
    (fun n => 2 * n + 1)

#eval sum_example (Sum.inl 3)
#eval sum_example (Sum.inr 3)
```

This example is similar to the previous one, but now an input to `sum_example` is implicitly either of the form `inl n` or `inr n`. In the first case, the function returns `2 * n`, and the second case, it returns `2 * n + 1`.

Notice that the product type depends on parameters `α β : Type` which are arguments to the constructors as well as `Prod`. Lean detects when these arguments can be inferred from later arguments to a constructor or the return type, and makes them implicit in that case.

In Section Defining the Natural Numbers we will see what happens when the constructor of an inductive type takes arguments from the inductive type itself. What characterizes the examples we consider in this section is that each constructor relies only on previously specified types.

Notice that a type with multiple constructors is disjunctive: an element of `Sum α β` is either of the form `inl a` *or* of the form `inl b`. A constructor with multiple arguments introduces conjunctive information: from an element `Prod.mk a b` of `Prod α β` we can extract `a` *and* `b`. An arbitrary inductive type can include both features, by having any number of constructors, each of which takes any number of arguments.

As with function definitions, Lean's inductive definition syntax will let you put named arguments to the constructors before the colon:

```
inductive Prod (α : Type u) (β : Type v) where
  | mk (fst : α) (snd : β) : Prod α β

inductive Sum (α : Type u) (β : Type v) where
  | inl (a : α) : Sum α β
  | inr (b : β) : Sum α β
```

The results of these definitions are essentially the same as the ones given earlier in this section.

A type, like `Prod`, that has only one constructor is purely conjunctive: the constructor simply packs the list of arguments into a single piece of data, essentially a tuple where the type of subsequent arguments can depend on the type of the initial argument. We can also think of such a type as a "record" or a "structure". In Lean, the keyword `structure` can be used to define such an inductive type as well as its projections, at the same time.

```
structure Prod (α : Type u) (β : Type v) where
  mk :: (fst : α) (snd : β)
```

This example simultaneously introduces the inductive type, `Prod`, its constructor, `mk`, the usual eliminators (`rec` and `recOn`), as well as the projections, `fst` and `snd`, as defined above.

If you do not name the constructor, Lean uses `mk` as a default. For example, the following defines a record to store a color as a triple of RGB values:

```
structure Color where
  (red : Nat) (green : Nat) (blue : Nat)
  deriving Repr

def yellow := Color.mk 255 255 0

#eval Color.red yellow
```

The definition of `yellow` forms the record with the three values shown, and the projection `Color.red` returns the red component.

You can avoid the parentheses if you add a line break between each field.

```
structure Color where
  red : Nat
  green : Nat
  blue : Nat
  deriving Repr
```

The `structure` command is especially useful for defining algebraic structures, and Lean provides substantial infrastructure to support working with them. Here, for example, is the definition of a semigroup:

```
structure Semigroup where
  carrier : Type u
  mul : carrier → carrier → carrier
  mul_assoc : ∀ a b c, mul (mul a b) c = mul a (mul b c)
```

We will see more examples in Chapter Structures and Records.

We have already discussed the dependent product type `Sigma`:

```
inductive Sigma {α : Type u} (β : α → Type v) where
  | mk : (a : α) → β a → Sigma β
```

Two more examples of inductive types in the library are the following:

```
inductive Option (α : Type u) where
  | none : Option α
  | some : α → Option α

inductive Inhabited (α : Type u) where
  | mk : α → Inhabited α
```

In the semantics of dependent type theory, there is no built-in notion of a partial function. Every element of a function type `α → β` or a dependent function type `(a : α) → β` is assumed to have a value at every input. The `Option` type provides a way of representing partial functions. An element of `Option β` is either `none` or of the form `some b`, for some value `b : β`. Thus we can think of an element `f` of the type `α → Option β` as being a partial function from `α` to `β`: for every `a : α`, `f a` either returns `none`, indicating `f a` is "undefined", or `some b`.

An element of `Inhabited α` is simply a witness to the fact that there is an element of `α`. Later, we will see that `Inhabited` is an example of a *type class* in Lean: Lean can be instructed that suitable base types are inhabited, and can automatically infer that other constructed types are inhabited on that basis.

As exercises, we encourage you to develop a notion of composition for partial functions from `α` to `β` and `β` to `γ`, and show that it behaves as expected. We also encourage you to show that `Bool` and `Nat` are inhabited, that the product of two inhabited types is inhabited, and that the type of functions to an inhabited type is inhabited.

Inductively Defined Propositions

Inductively defined types can live in any type universe, including the bottom-most one, `Prop`. In fact, this is exactly how the logical connectives are defined.

```
inductive False : Prop

inductive True : Prop where
  | intro : True

inductive And (a b : Prop) : Prop where
  | intro : a → b → And a b

inductive Or (a b : Prop) : Prop where
  | inl : a → Or a b
  | inr : b → Or a b
```

You should think about how these give rise to the introduction and elimination rules that you have already seen. There are rules that govern what the eliminator of an inductive type can eliminate *to*, that is, what kinds of types can be the target of a recursor. Roughly speaking, what characterizes inductive types in `Prop` is that one can only eliminate to other

types in `Prop`. This is consistent with the understanding that if `p : Prop`, an element `hp : p` carries no data. There is a small exception to this rule, however, which we will discuss below, in Section Inductive Families.

Even the existential quantifier is inductively defined:

```
inductive Exists {α : Sort u} (p : α → Prop) : Prop where
  | intro (w : α) (h : p w) : Exists p
```

Keep in mind that the notation `∃ x : α, p` is syntactic sugar for `Exists (fun x : α => p)`.

The definitions of `False`, `True`, `And`, and `Or` are perfectly analogous to the definitions of `Empty`, `Unit`, `Prod`, and `Sum`. The difference is that the first group yields elements of `Prop`, and the second yields elements of `Type u` for some `u`. In a similar way, `∃ x : α, p` is a `Prop`-valued variant of `Σ x : α, p`.

This is a good place to mention another inductive type, denoted `{x : α // p}`, which is sort of a hybrid between `∃ x : α, P` and `Σ x : α, P`.

```
inductive Subtype {α : Type u} (p : α → Prop) where
  | mk : (x : α) → p x → Subtype p
```

In fact, in Lean, `Subtype` is defined using the structure command:

```
structure Subtype {α : Sort u} (p : α → Prop) where
  val : α
  property : p val
```

The notation `{x : α // p x}` is syntactic sugar for `Subtype (fun x : α => p x)`. It is modeled after subset notation in set theory: the idea is that `{x : α // p x}` denotes the collection of elements of `α` that have property `p`.

Defining the Natural Numbers

The inductively defined types we have seen so far are "flat": constructors wrap data and insert it into a type, and the corresponding recursor unpacks the data and acts on it. Things get much more interesting when the constructors act on elements of the very type being defined. A canonical example is the type `Nat` of natural numbers:

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

There are two constructors. We start with `zero : Nat`; it takes no arguments, so we have it from the start. In contrast, the constructor `succ` can only be applied to a previously constructed `Nat`. Applying it to `zero` yields `succ zero : Nat`. Applying it again yields `succ (succ zero) : Nat`, and so on. Intuitively, `Nat` is the "smallest" type with these constructors, meaning that it is exhaustively (and freely) generated by starting with `zero` and applying `succ` repeatedly.

As before, the recursor for `Nat` is designed to define a dependent function `f` from `Nat` to any domain, that is, an element `f` of `(n : Nat) → motive n` for some `motive : Nat → Sort u`. It has to handle two cases: the case where the input is `zero`, and the case where the input is of the form `succ n` for some `n : Nat`. In the first case, we simply specify a target value with the appropriate type, as before. In the second case, however, the recursor can assume that a value of `f` at `n` has already been computed. As a result, the next argument to the recursor specifies a value for `f (succ n)` in terms of `n` and `f n`. If we check the type of the recursor,

```
#check @Nat.rec
```

you find the following:

```
  {motive : Nat → Sort u}
  → motive Nat.zero
  → ((n : Nat) → motive n → motive (Nat.succ n))
  → (t : Nat) → motive t
```

The implicit argument, `motive`, is the codomain of the function being defined. In type theory it is common to say `motive` is the *motive* for the elimination/recursion, since it describes the kind of object we wish to construct. The next two arguments specify how to compute the zero and successor cases, as described above. They are also known as the *minor premises*. Finally, the `t : Nat`, is the input to the function. It is also known as the *major premise*.

The `Nat.recOn` is similar to `Nat.rec` but the major premise occurs before the minor premises.

```
@Nat.recOn :
  {motive : Nat → Sort u}
  → (t : Nat)
  → motive Nat.zero
  → ((n : Nat) → motive n → motive (Nat.succ n))
  → motive t
```

Consider, for example, the addition function `add m n` on the natural numbers. Fixing `m`, we can define addition by recursion on `n`. In the base case, we set `add m zero` to `m`. In the successor step, assuming the value `add m n` is already determined, we define `add m (succ n)` to be `succ (add m n)`.

```
inductive Nat where
  | zero : Nat
  | succ : Nat → Nat
  deriving Repr

def add (m n : Nat) : Nat :=
  match n with
  | Nat.zero   => m
  | Nat.succ n => Nat.succ (add m n)

open Nat

#eval add (succ (succ zero)) (succ zero)
```

It is useful to put such definitions into a namespace, `Nat`. We can then go on to define familiar notation in that namespace. The two defining equations for addition now hold definitionally:

```
namespace Nat

def add (m n : Nat) : Nat :=
  match n with
  | Nat.zero   => m
  | Nat.succ n => Nat.succ (add m n)

instance : Add Nat where
  add := add

theorem add_zero (m : Nat) : m + zero = m := rfl
theorem add_succ (m n : Nat) : m + succ n = succ (m + n) := rfl

end Nat
```

We will explain how the `instance` command works in Chapter Type Classes. In the examples below, we will use Lean's version of the natural numbers.

Proving a fact like `zero + m = m`, however, requires a proof by induction. As observed above, the induction principle is just a special case of the recursion principle, when the codomain `motive n` is an element of `Prop`. It represents the familiar pattern of an inductive proof: to prove `∀ n, motive n`, first prove `motive 0`, and then, for arbitrary `n`, assume `ih : motive n` and prove `motive (succ n)`.

```
open Nat

theorem zero_add (n : Nat) : 0 + n = n :=
  Nat.recOn (motive := fun x => 0 + x = x)
   n
   (show 0 + 0 = 0 from rfl)
   (fun (n : Nat) (ih : 0 + n = n) =>
    show 0 + succ n = succ n from
    calc 0 + succ n
      _ = succ (0 + n) := rfl
      _ = succ n       := by rw [ih])
```

Notice that, once again, when `Nat.recOn` is used in the context of a proof, it is really the induction principle in disguise. The `rewrite` and `simp` tactics tend to be very effective in proofs like these. In this case, each can be used to reduce the proof to:

```
open Nat

theorem zero_add (n : Nat) : 0 + n = n :=
  Nat.recOn (motive := fun x => 0 + x = x) n
    rfl
    (fun n ih => by simp [add_succ, ih])
```

As another example, let us prove the associativity of addition, `∀ m n k, m + n + k = m + (n + k)`. (The notation `+`, as we have defined it, associates to the left, so `m + n + k` is really `(m + n) + k`.) The hardest part is figuring out which variable to do the induction on. Since addition is defined by recursion on the second argument, `k` is a good guess, and once we make that choice the proof almost writes itself:

```
open Nat
theorem add_assoc (m n k : Nat) : m + n + k = m + (n + k) :=
  Nat.recOn (motive := fun k => m + n + k = m + (n + k)) k
    (show m + n + 0 = m + (n + 0) from rfl)
    (fun k (ih : m + n + k = m + (n + k)) =>
      show m + n + succ k = m + (n + succ k) from
      calc m + n + succ k
        _ = succ (m + n + k)   := rfl
        _ = succ (m + (n + k)) := by rw [ih]
        _ = m + succ (n + k)   := rfl
        _ = m + (n + succ k)   := rfl)
```

Once again, you can reduce the proof to:

```
open Nat
theorem add_assoc (m n k : Nat) : m + n + k = m + (n + k) :=
  Nat.recOn (motive := fun k => m + n + k = m + (n + k)) k
    rfl
    (fun k ih => by simp [Nat.add_succ, ih])
```

Suppose we try to prove the commutativity of addition. Choosing induction on the second argument, we might begin as follows:

```
open Nat
theorem add_comm (m n : Nat) : m + n = n + m :=
  Nat.recOn (motive := fun x => m + x = x + m) n
   (show m + 0 = 0 + m by rw [Nat.zero_add, Nat.add_zero])
   (fun (n : Nat) (ih : m + n = n + m) =>
    show m + succ n = succ n + m from
    calc m + succ n
      _ = succ (m + n) := rfl
      _ = succ (n + m) := by rw [ih]
      _ = succ n + m   := sorry)
```

At this point, we see that we need another supporting fact, namely, that `succ (n + m) = succ n + m`. You can prove this by induction on `m`:

```
open Nat

theorem succ_add (n m : Nat) : succ n + m = succ (n + m) :=
  Nat.recOn (motive := fun x => succ n + x = succ (n + x)) m
    (show succ n + 0 = succ (n + 0) from rfl)
    (fun (m : Nat) (ih : succ n + m = succ (n + m)) =>
     show succ n + succ m = succ (n + succ m) from
     calc succ n + succ m
       _ = succ (succ n + m)   := rfl
       _ = succ (succ (n + m)) := by rw [ih]
       _ = succ (n + succ m)   := rfl)
```

You can then replace the `sorry` in the previous proof with `succ_add`. Yet again, the proofs can be compressed:

```
open Nat
theorem succ_add (n m : Nat) : succ n + m = succ (n + m) :=
  Nat.recOn (motive := fun x => succ n + x = succ (n + x)) m
    rfl
    (fun m ih => by simp only [add_succ, ih])

theorem add_comm (m n : Nat) : m + n = n + m :=
  Nat.recOn (motive := fun x => m + x = x + m) n
    (by simp)
    (fun m ih => by simp [add_succ, succ_add, ih])
```

Other Recursive Data Types

Let us consider some more examples of inductively defined types. For any type, `α`, the type `List α` of lists of elements of `α` is defined in the library.

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

namespace List

def append (as bs : List α) : List α :=
  match as with
  | nil       => bs
  | cons a as => cons a (append as bs)

theorem nil_append (as : List α) : append nil as = as :=
  rfl

theorem cons_append (a : α) (as bs : List α)
                    : append (cons a as) bs = cons a (append as bs) :=
  rfl

end List
```

A list of elements of type `α` is either the empty list, `nil`, or an element `h : α` followed by a list `t : List α`. The first element, `h`, is commonly known as the "head" of the list, and the remainder, `t`, is known as the "tail."

As an exercise, prove the following:

```
theorem append_nil (as : List α) : append as nil = as :=
  sorry

theorem append_assoc (as bs cs : List α)
        : append (append as bs) cs = append as (append bs cs) :=
  sorry
```

Try also defining the function `length : {α : Type u} → List α → Nat` that returns the length of a list, and prove that it behaves as expected (for example, `length (append as bs) = length as + length bs`).

For another example, we can define the type of binary trees:

```
inductive BinaryTree where
  | leaf : BinaryTree
  | node : BinaryTree → BinaryTree → BinaryTree
```

In fact, we can even define the type of countably branching trees:

```
inductive CBTree where
  | leaf : CBTree
  | sup : (Nat → CBTree) → CBTree

namespace CBTree

def succ (t : CBTree) : CBTree :=
  sup (fun _ => t)

def toCBTree : Nat → CBTree
  | 0 => leaf
  | n+1 => succ (toCBTree n)

def omega : CBTree :=
  sup toCBTree

end CBTree
```

Tactics for Inductive Types

Given the fundamental importance of inductive types in Lean, it should not be surprising that there are a number of tactics designed to work with them effectively. We describe some of them here.

The `cases` tactic works on elements of an inductively defined type, and does what the name suggests: it decomposes the element according to each of the possible constructors. In its most basic form, it is applied to an element `x` in the local context. It then reduces the goal to cases in which `x` is replaced by each of the constructions.

```
example (p : Nat → Prop) (hz : p 0) (hs : ∀ n, p (Nat.succ n)) : ∀ n, p n := by
  intro n
  cases n
  . exact hz  -- goal is p 0
  . apply hs  -- goal is a : Nat ⊢ p (succ a)
```

There are extra bells and whistles. For one thing, `cases` allows you to choose the names for each alternative using a `with` clause. In the next example, for example, we choose the name `m` for the argument to `succ`, so that the second case refers to `succ m`. More importantly, the cases tactic will detect any items in the local context that depend on the target variable. It reverts these elements, does the split, and reintroduces them. In the example below, notice that the hypothesis `h : n ≠ 0` becomes `h : 0 ≠ 0` in the first branch, and `h : succ m ≠ 0` in the second.

```
open Nat

example (n : Nat) (h : n ≠ 0) : succ (pred n) = n := by
  cases n with
  | zero =>
    -- goal: h : 0 ≠ 0 ⊢ succ (pred 0) = 0
    apply absurd rfl h
  | succ m =>
    -- second goal: h : succ m ≠ 0 ⊢ succ (pred (succ m)) = succ m
    rfl
```

Notice that `cases` can be used to produce data as well as prove propositions.

```
def f (n : Nat) : Nat := by
  cases n; exact 3; exact 7

example : f 0 = 3 := rfl
example : f 5 = 7 := rfl
```

Once again, cases will revert, split, and then reintroduce dependencies in the context.

```
def Tuple (α : Type) (n : Nat) :=
  { as : List α // as.length = n }

def f {n : Nat} (t : Tuple α n) : Nat := by
  cases n; exact 3; exact 7

def myTuple : Tuple Nat 3 :=
  ⟨[0, 1, 2], rfl⟩

example : f myTuple = 7 :=
  rfl
```

Here is an example of multiple constructors with arguments.

```
inductive Foo where
  | bar1 : Nat → Nat → Foo
  | bar2 : Nat → Nat → Nat → Foo

def silly (x : Foo) : Nat := by
  cases x with
  | bar1 a b => exact b
  | bar2 c d e => exact e
```

The alternatives for each constructor don't need to be solved in the order the constructors were declared.

```
def silly (x : Foo) : Nat := by
  cases x with
  | bar2 c d e => exact e
  | bar1 a b => exact b
```

The syntax of the `with` is convenient for writing structured proofs. Lean also provides a complementary `case` tactic, which allows you to focus on goal assign variable names.

```
def silly (x : Foo) : Nat := by
  cases x
  case bar1 a b => exact b
  case bar2 c d e => exact e
```

The `case` tactic is clever, in that it will match the constructor to the appropriate goal. For example, we can fill the goals above in the opposite order:

```
def silly (x : Foo) : Nat := by
  cases x
  case bar2 c d e => exact e
  case bar1 a b => exact b
```

You can also use `cases` with an arbitrary expression. Assuming that expression occurs in the goal, the cases tactic will generalize over the expression, introduce the resulting universally quantified variable, and case on that.

```
open Nat

example (p : Nat → Prop) (hz : p 0) (hs : ∀ n, p (succ n)) (m k : Nat)
        : p (m + 3 * k) := by
  cases m + 3 * k
  exact hz   -- goal is p 0
  apply hs   -- goal is a : Nat ⊢ p (succ a)
```

Think of this as saying "split on cases as to whether `m + 3 * k` is zero or the successor of some number." The result is functionally equivalent to the following:

```
open Nat

example (p : Nat → Prop) (hz : p 0) (hs : ∀ n, p (succ n)) (m k : Nat)
        : p (m + 3 * k) := by
  generalize m + 3 * k = n
  cases n
  exact hz   -- goal is p 0
  apply hs   -- goal is a : Nat ⊢ p (succ a)
```

Notice that the expression `m + 3 * k` is erased by `generalize`; all that matters is whether it is of the form `0` or `succ a`. This form of `cases` will *not* revert any hypotheses that also mention the expression in the equation (in this case, `m + 3 * k`). If such a term appears in a hypothesis and you want to generalize over that as well, you need to `revert` it explicitly.

If the expression you case on does not appear in the goal, the `cases` tactic uses `have` to put the type of the expression into the context. Here is an example:

```
example (p : Prop) (m n : Nat)
        (h₁ : m < n → p) (h₂ : m ≥ n → p) : p := by
  cases Nat.lt_or_ge m n
  case inl hlt => exact h₁ hlt
  case inr hge => exact h₂ hge
```

The theorem `Nat.lt_or_ge m n` says `m < n ∨ m ≥ n`, and it is natural to think of the proof above as splitting on these two cases. In the first branch, we have the hypothesis `hlt : m < n`, and in the second we have the hypothesis `hge : m ≥ n`. The proof above is functionally equivalent to the following:

```
example (p : Prop) (m n : Nat)
        (h₁ : m < n → p) (h₂ : m ≥ n → p) : p := by
  have h : m < n ∨ m ≥ n := Nat.lt_or_ge m n
  cases h
  case inl hlt => exact h₁ hlt
  case inr hge => exact h₂ hge
```

After the first two lines, we have `h : m < n ∨ m ≥ n` as a hypothesis, and we simply do cases on that.

Here is another example, where we use the decidability of equality on the natural numbers to split on the cases `m = n` and `m ≠ n`.

```
#check Nat.sub_self

example (m n : Nat) : m - n = 0 ∨ m ≠ n := by
  cases Decidable.em (m = n) with
  | inl heq => rw [heq]; apply Or.inl; exact Nat.sub_self n
  | inr hne => apply Or.inr; exact hne
```

Remember that if you `open Classical`, you can use the law of the excluded middle for any proposition at all. But using type class inference (see Chapter Type Classes), Lean can actually find the relevant decision procedure, which means that you can use the case split in a computable function.

Just as the `cases` tactic can be used to carry out proof by cases, the `induction` tactic can be used to carry out proofs by induction. The syntax is similar to that of `cases`, except that the argument can only be a term in the local context. Here is an example:

```
theorem zero_add (n : Nat) : 0 + n = n := by
  induction n with
  | zero => rfl
  | succ n ih => rw [Nat.add_succ, ih]
```

As with `cases`, we can use the `case` tactic instead of `with`.

```
theorem zero_add (n : Nat) : 0 + n = n := by
  induction n
  case zero => rfl
  case succ n ih => rw [Nat.add_succ, ih]
```

Here are some additional examples:

```
open Nat

theorem zero_add (n : Nat) : 0 + n = n := by
  induction n <;> simp [*, add_zero, add_succ]

theorem succ_add (m n : Nat) : succ m + n = succ (m + n) := by
  induction n <;> simp [*, add_zero, add_succ]

theorem add_comm (m n : Nat) : m + n = n + m := by
  induction n <;> simp [*, add_zero, add_succ, succ_add, zero_add]

theorem add_assoc (m n k : Nat) : m + n + k = m + (n + k) := by
  induction k <;> simp [*, add_zero, add_succ]
```

The `induction` tactic also supports user-defined induction principles with multiple targets (aka major premises).

```
/-
theorem Nat.mod.inductionOn
      {motive : Nat → Nat → Sort u}
      (x y  : Nat)
      (ind  : ∀ x y, 0 < y ∧ y ≤ x → motive (x - y) y → motive x y)
      (base : ∀ x y, ¬(0 < y ∧ y ≤ x) → motive x y)
      : motive x y :=
-/

example (x : Nat) {y : Nat} (h : y > 0) : x % y < y := by
  induction x, y using Nat.mod.inductionOn with
  | ind x y h₁ ih =>
    rw [Nat.mod_eq_sub_mod h₁.2]
    exact ih h
  | base x y h₁ =>
    have : ¬ 0 < y ∨ ¬ y ≤ x := Iff.mp (Decidable.not_and_iff_or_not ..) h₁
    match this with
    | Or.inl h₁ => exact absurd h h₁
    | Or.inr h₁ =>
      have hgt : y > x := Nat.gt_of_not_le h₁
      rw [← Nat.mod_eq_of_lt hgt] at hgt
      assumption
```

You can use the `match` notation in tactics too:

```
example : p ∨ q → q ∨ p := by
  intro h
  match h with
  | Or.inl _  => apply Or.inr; assumption
  | Or.inr h2 => apply Or.inl; exact h2
```

As a convenience, pattern-matching has been integrated into tactics such as `intro` and `funext`.

```
example : s ∧ q ∧ r → p ∧ r → q ∧ p := by
  intro ⟨_, ⟨hq, _⟩⟩ ⟨hp, _⟩
  exact ⟨hq, hp⟩

example :
    (fun (x : Nat × Nat) (y : Nat × Nat) => x.1 + y.2)
    =
    (fun (x : Nat × Nat) (z : Nat × Nat) => z.2 + x.1) := by
  funext (a, b) (c, d)
  show a + d = d + a
  rw [Nat.add_comm]
```

We close this section with one last tactic that is designed to facilitate working with inductive types, namely, the `injection` tactic. By design, the elements of an inductive type are freely generated, which is to say, the constructors are injective and have disjoint ranges. The `injection` tactic is designed to make use of this fact:

```
open Nat

example (m n k : Nat) (h : succ (succ m) = succ (succ n))
        : n + k = m + k := by
  injection h with h'
  injection h' with h''
  rw [h'']
```

The first instance of the tactic adds `h' : succ m = succ n` to the context, and the second adds `h'' : m = n`.

The `injection` tactic also detects contradictions that arise when different constructors are set equal to one another, and uses them to close the goal.

```
open Nat

example (m n : Nat) (h : succ m = 0) : n = n + 7 := by
  injection h

example (m n : Nat) (h : succ m = 0) : n = n + 7 := by
  contradiction

example (h : 7 = 4) : False := by
  contradiction
```

As the second example shows, the `contradiction` tactic also detects contradictions of this form.

Inductive Families

We are almost done describing the full range of inductive definitions accepted by Lean. So far, you have seen that Lean allows you to introduce inductive types with any number of recursive constructors. In fact, a single inductive definition can introduce an indexed *family* of inductive types, in a manner we now describe.

An inductive family is an indexed family of types defined by a simultaneous induction of the following form:

```
inductive foo : ... → Sort u where
  | constructor₁ : ... → foo ...
  | constructor₂ : ... → foo ...
  ...
  | constructorₙ : ... → foo ...
```

In contrast to an ordinary inductive definition, which constructs an element of some `Sort u`, the more general version constructs a function `... → Sort u`, where "`...`" denotes a sequence of argument types, also known as *indices*. Each constructor then constructs an element of some member of the family. One example is the definition of `Vector α n`, the type of vectors of elements of `α` of length `n`:

```
inductive Vector (α : Type u) : Nat → Type u where
  | nil  : Vector α 0
  | cons : α → {n : Nat} → Vector α n → Vector α (n+1)
```

Notice that the `cons` constructor takes an element of `Vector α n` and returns an element of `Vector α (n+1)`, thereby using an element of one member of the family to build an element of another.

A more exotic example is given by the definition of the equality type in Lean:

```
inductive Eq {α : Sort u} (a : α) : α → Prop where
  | refl : Eq a a
```

For each fixed `α : Sort u` and `a : α`, this definition constructs a family of types `Eq a x`, indexed by `x : α`. Notably, however, there is only one constructor, `refl`, which is an element of `Eq a a`. Intuitively, the only way to construct a proof of `Eq a x` is to use reflexivity, in the case where `x` is `a`. Note that `Eq a a` is the only inhabited type in the family of types `Eq a x`. The elimination principle generated by Lean is as follows:

```
universe u v

#check (@Eq.rec : {α : Sort u} → {a : α} → {motive : (x : α) → a = x → Sort v}
                  → motive a rfl → {b : α} → (h : a = b) → motive b h)
```

It is a remarkable fact that all the basic axioms for equality follow from the constructor, `refl`, and the eliminator, `Eq.rec`. The definition of equality is atypical, however; see the

discussion in Section Axiomatic Details.

The recursor `Eq.rec` is also used to define substitution:

```
theorem subst {α : Type u} {a b : α} {p : α → Prop} (h₁ : Eq a b) (h₂ : p a) : p
b :=
  Eq.rec (motive := fun x _ => p x) h₂ h₁
```

You can also define `subst` using `match`.

```
theorem subst {α : Type u} {a b : α} {p : α → Prop} (h₁ : Eq a b) (h₂ : p a) : p
b :=
  match h₁ with
  | rfl => h₂
```

Actually, Lean compiles the `match` expressions using a definition based on `Eq.rec`.

```
theorem subst {α : Type u} {a b : α} {p : α → Prop} (h₁ : Eq a b) (h₂ : p a) : p
b :=
  match h₁ with
  | rfl => h₂

set_option pp.all true
#print subst
  -- ... subst.match_1 ...
#print subst.match_1
  -- ... Eq.casesOn ...
#print Eq.casesOn
  -- ... Eq.rec ...
```

Using the recursor or `match` with `h₁ : a = b`, we may assume `a` and `b` are the same, in which case, `p b` and `p a` are the same.

It is not hard to prove that `Eq` is symmetric and transitive. In the following example, we prove `symm` and leave as exercises the theorems `trans` and `congr` (congruence).

```
theorem symm {α : Type u} {a b : α} (h : Eq a b) : Eq b a :=
  match h with
  | rfl => rfl

theorem trans {α : Type u} {a b c : α} (h₁ : Eq a b) (h₂ : Eq b c) : Eq a c :=
  sorry

theorem congr {α β : Type u} {a b : α} (f : α → β) (h : Eq a b) : Eq (f a) (f b)
:=
  sorry
```

In the type theory literature, there are further generalizations of inductive definitions, for example, the principles of *induction-recursion* and *induction-induction*. These are not supported by Lean.

Axiomatic Details

We have described inductive types and their syntax through examples. This section provides additional information for those interested in the axiomatic foundations.

We have seen that the constructor to an inductive type takes *parameters* --- intuitively, the arguments that remain fixed throughout the inductive construction --- and *indices*, the arguments parameterizing the family of types that is simultaneously under construction. Each constructor should have a type, where the argument types are built up from previously defined types, the parameter and index types, and the inductive family currently being defined. The requirement is that if the latter is present at all, it occurs only *strictly positively*. This means simply that any argument to the constructor in which it occurs is a dependent arrow type in which the inductive type under definition occurs only as the resulting type, where the indices are given in terms of constants and previous arguments.

Since an inductive type lives in `Sort u` for some `u`, it is reasonable to ask *which* universe levels `u` can be instantiated to. Each constructor `c` in the definition of a family `C` of inductive types is of the form

```
c : (a : α) → (b : β[a]) → C a p[a,b]
```

where `a` is a sequence of data type parameters, `b` is the sequence of arguments to the constructors, and `p[a, b]` are the indices, which determine which element of the inductive family the construction inhabits. (Note that this description is somewhat misleading, in that the arguments to the constructor can appear in any order as long as the dependencies make sense.) The constraints on the universe level of `C` fall into two cases, depending on whether or not the inductive type is specified to land in `Prop` (that is, `Sort 0`).

Let us first consider the case where the inductive type is *not* specified to land in `Prop`. Then the universe level `u` is constrained to satisfy the following:

> For each constructor `c` as above, and each `βk[a]` in the sequence `β[a]`, if `βk[a] : Sort v`, we have `u` ≥ `v`.

In other words, the universe level `u` is required to be at least as large as the universe level of each type that represents an argument to a constructor.

When the inductive type is specified to land in `Prop`, there are no constraints on the universe levels of the constructor arguments. But these universe levels do have a bearing on the elimination rule. Generally speaking, for an inductive type in `Prop`, the motive of the elimination rule is required to be in `Prop`.

There is an exception to this last rule: we are allowed to eliminate from an inductively defined `Prop` to an arbitrary `Sort` when there is only one constructor and each constructor

argument is either in `Prop` or an index. The intuition is that in this case the elimination does not make use of any information that is not already given by the mere fact that the type of argument is inhabited. This special case is known as *singleton elimination*.

We have already seen singleton elimination at play in applications of `Eq.rec`, the eliminator for the inductively defined equality type. We can use an element `h : Eq a b` to cast an element `t' : p a` to `p b` even when `p a` and `p b` are arbitrary types, because the cast does not produce new data; it only reinterprets the data we already have. Singleton elimination is also used with heterogeneous equality and well-founded recursion, which will be discussed in a Chapter Induction and Recursion.

Mutual and Nested Inductive Types

We now consider two generalizations of inductive types that are often useful, which Lean supports by "compiling" them down to the more primitive kinds of inductive types described above. In other words, Lean parses the more general definitions, defines auxiliary inductive types based on them, and then uses the auxiliary types to define the ones we really want. Lean's equation compiler, described in the next chapter, is needed to make use of these types effectively. Nonetheless, it makes sense to describe the declarations here, because they are straightforward variations on ordinary inductive definitions.

First, Lean supports *mutually defined* inductive types. The idea is that we can define two (or more) inductive types at the same time, where each one refers to the other(s).

```
mutual
  inductive Even : Nat → Prop where
    | even_zero : Even 0
    | even_succ : (n : Nat) → Odd n → Even (n + 1)

  inductive Odd : Nat → Prop where
    | odd_succ : (n : Nat) → Even n → Odd (n + 1)
end
```

In this example, two types are defined simultaneously: a natural number `n` is `Even` if it is `0` or one more than an `Odd` number, and `Odd` if it is one more than an `Even` number. In the exercises below, you are asked to spell out the details.

A mutual inductive definition can also be used to define the notation of a finite tree with nodes labelled by elements of `α`:

```
mutual
    inductive Tree (α : Type u) where
      | node : α → TreeList α → Tree α

    inductive TreeList (α : Type u) where
      | nil  : TreeList α
      | cons : Tree α → TreeList α → TreeList α
end
```

With this definition, one can construct an element of `Tree α` by giving an element of `α` together with a list of subtrees, possibly empty. The list of subtrees is represented by the type `TreeList α`, which is defined to be either the empty list, `nil`, or the `cons` of a tree and an element of `TreeList α`.

This definition is inconvenient to work with, however. It would be much nicer if the list of subtrees were given by the type `List (Tree α)`, especially since Lean's library contains a number of functions and theorems for working with lists. One can show that the type `TreeList α` is *isomorphic* to `List (Tree α)`, but translating results back and forth along this isomorphism is tedious.

In fact, Lean allows us to define the inductive type we really want:

```
inductive Tree (α : Type u) where
  | mk : α → List (Tree α) → Tree α
```

This is known as a *nested* inductive type. It falls outside the strict specification of an inductive type given in the last section because `Tree` does not occur strictly positively among the arguments to `mk`, but, rather, nested inside the `List` type constructor. Lean then automatically builds the isomorphism between `TreeList α` and `List (Tree α)` in its kernel, and defines the constructors for `Tree` in terms of the isomorphism.

Exercises

1. Try defining other operations on the natural numbers, such as multiplication, the predecessor function (with `pred 0 = 0`), truncated subtraction (with `n - m = 0` when `m` is greater than or equal to `n`), and exponentiation. Then try proving some of their basic properties, building on the theorems we have already proved.

   Since many of these are already defined in Lean's core library, you should work within a namespace named `Hidden`, or something like that, in order to avoid name clashes.

2. Define some operations on lists, like a `length` function or the `reverse` function. Prove some properties, such as the following:

   a. `length (s ++ t) = length s + length t`

b. `length (reverse t) = length t`

c. `reverse (reverse t) = t`

3. Define an inductive data type consisting of terms built up from the following constructors:

   - `const n`, a constant denoting the natural number `n`
   - `var n`, a variable, numbered `n`
   - `plus s t`, denoting the sum of `s` and `t`
   - `times s t`, denoting the product of `s` and `t`

   Recursively define a function that evaluates any such term with respect to an assignment of values to the variables.

4. Similarly, define the type of propositional formulas, as well as functions on the type of such formulas: an evaluation function, functions that measure the complexity of a formula, and a function that substitutes another formula for a given variable.

Induction and Recursion

In the previous chapter, we saw that inductive definitions provide a powerful means of introducing new types in Lean. Moreover, the constructors and the recursors provide the only means of defining functions on these types. By the propositions-as-types correspondence, this means that induction is the fundamental method of proof.

Lean provides natural ways of defining recursive functions, performing pattern matching, and writing inductive proofs. It allows you to define a function by specifying equations that it should satisfy, and it allows you to prove a theorem by specifying how to handle various cases that can arise. Behind the scenes, these descriptions are "compiled" down to primitive recursors, using a procedure that we refer to as the "equation compiler." The equation compiler is not part of the trusted code base; its output consists of terms that are checked independently by the kernel.

Pattern Matching

The interpretation of schematic patterns is the first step of the compilation process. We have seen that the `casesOn` recursor can be used to define functions and prove theorems by cases, according to the constructors involved in an inductively defined type. But complicated definitions may use several nested `casesOn` applications, and may be hard to read and understand. Pattern matching provides an approach that is more convenient, and familiar to users of functional programming languages.

Consider the inductively defined type of natural numbers. Every natural number is either `zero` or `succ x`, and so you can define a function from the natural numbers to an arbitrary type by specifying a value in each of those cases:

```
open Nat

def sub1 : Nat → Nat
  | zero   => zero
  | succ x => x

def isZero : Nat → Bool
  | zero   => true
  | succ x => false
```

The equations used to define these functions hold definitionally:

```
example : sub1 0 = 0 := rfl
example (x : Nat) : sub1 (succ x) = x := rfl

example : isZero 0 = true := rfl
example (x : Nat) : isZero (succ x) = false := rfl

example : sub1 7 = 6 := rfl
example (x : Nat) : isZero (x + 3) = false := rfl
```

Instead of `zero` and `succ`, we can use more familiar notation:

```
def sub1 : Nat → Nat
  | 0   => 0
  | x+1 => x

def isZero : Nat → Bool
  | 0   => true
  | x+1 => false
```

Because addition and the zero notation have been assigned the `[match_pattern]` attribute, they can be used in pattern matching. Lean simply normalizes these expressions until the constructors `zero` and `succ` are exposed.

Pattern matching works with any inductive type, such as products and option types:

```
def swap : α × β → β × α
  | (a, b) => (b, a)

def foo : Nat × Nat → Nat
  | (m, n) => m + n

def bar : Option Nat → Nat
  | some n => n + 1
  | none   => 0
```

Here we use it not only to define a function, but also to carry out a proof by cases:

```
def not : Bool → Bool
  | true  => false
  | false => true

theorem not_not : ∀ (b : Bool), not (not b) = b
  | true  => rfl  -- proof that not (not true) = true
  | false => rfl  -- proof that not (not false) = false
```

Pattern matching can also be used to destruct inductively defined propositions:

```
example (p q : Prop) : p ∧ q → q ∧ p
  | And.intro h₁ h₂ => And.intro h₂ h₁

example (p q : Prop) : p ∨ q → q ∨ p
  | Or.inl hp => Or.inr hp
  | Or.inr hq => Or.inl hq
```

This provides a compact way of unpacking hypotheses that make use of logical connectives.

In all these examples, pattern matching was used to carry out a single case distinction. More interestingly, patterns can involve nested constructors, as in the following examples.

```
def sub2 : Nat → Nat
  | 0   => 0
  | 1   => 0
  | x+2 => x
```

The equation compiler first splits on cases as to whether the input is `zero` or of the form `succ x`. It then does a case split on whether `x` is of the form `zero` or `succ x`. It determines the necessary case splits from the patterns that are presented to it, and raises an error if the patterns fail to exhaust the cases. Once again, we can use arithmetic notation, as in the version below. In either case, the defining equations hold definitionally.

```
example : sub2 0 = 0 := rfl
example : sub2 1 = 0 := rfl
example : sub2 (x+2) = x := rfl

example : sub2 5 = 3 := rfl
```

You can write `#print sub2` to see how the function was compiled to recursors. (Lean will tell you that `sub2` has been defined in terms of an internal auxiliary function, `sub2.match_1`, but you can print that out too.) Lean uses these auxiliary functions to compile `match` expressions. Actually, the definition above is expanded to

```
def sub2 : Nat → Nat :=
  fun x =>
    match x with
    | 0   => 0
    | 1   => 0
    | x+2 => x
```

Here are some more examples of nested pattern matching:

```
example (p q : α → Prop)
        : (∃ x, p x ∨ q x) → (∃ x, p x) ∨ (∃ x, q x)
  | Exists.intro x (Or.inl px) => Or.inl (Exists.intro x px)
  | Exists.intro x (Or.inr qx) => Or.inr (Exists.intro x qx)

def foo : Nat × Nat → Nat
  | (0, n)     => 0
  | (m+1, 0)   => 1
  | (m+1, n+1) => 2
```

The equation compiler can process multiple arguments sequentially. For example, it would be more natural to define the previous example as a function of two arguments:

```
def foo : Nat → Nat → Nat
  | 0,   n   => 0
  | m+1, 0   => 1
  | m+1, n+1 => 2
```

Here is another example:

```
def bar : List Nat → List Nat → Nat
  | [],      []      => 0
  | a :: as, []      => a
  | [],      b :: bs => b
  | a :: as, b :: bs => a + b
```

Note that the patterns are separated by commas.

In each of the following examples, splitting occurs on only the first argument, even though the others are included among the list of patterns.

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

def or : Bool → Bool → Bool
  | true,  _ => true
  | false, a => a

def cond : Bool → α → α → α
  | true,  x, y => x
  | false, x, y => y
```

Notice also that, when the value of an argument is not needed in the definition, you can use an underscore instead. This underscore is known as a *wildcard pattern*, or an *anonymous variable*. In contrast to usage outside the equation compiler, here the underscore does *not* indicate an implicit argument. The use of underscores for wildcards is common in functional programming languages, and so Lean adopts that notation. Section Wildcards and Overlapping Patterns expands on the notion of a wildcard, and Section Inaccessible Patterns explains how you can use implicit arguments in patterns as well.

As described in Chapter Inductive Types, inductive data types can depend on parameters. The following example defines the `tail` function using pattern matching. The argument `α : Type u` is a parameter and occurs before the colon to indicate it does not participate in the pattern matching. Lean also allows parameters to occur after `:`, but it cannot pattern match on them.

```
def tail1 {α : Type u} : List α → List α
  | []      => []
  | a :: as => as

def tail2 : {α : Type u} → List α → List α
  | α, []      => []
  | α, a :: as => as
```

Despite the different placement of the parameter `α` in these two examples, in both cases it is treated in the same way, in that it does not participate in a case split.

Lean can also handle more complex forms of pattern matching, in which arguments to dependent types pose additional constraints on the various cases. Such examples of *dependent pattern matching* are considered in the Section Dependent Pattern Matching.

Wildcards and Overlapping Patterns

Consider one of the examples from the last section:

```
def foo : Nat → Nat → Nat
  | 0,   n   => 0
  | m+1, 0   => 1
  | m+1, n+1 => 2
```

An alternative presentation is:

```
def foo : Nat → Nat → Nat
  | 0, n => 0
  | m, 0 => 1
  | m, n => 2
```

In the second presentation, the patterns overlap; for example, the pair of arguments `0 0` matches all three cases. But Lean handles the ambiguity by using the first applicable equation, so in this example the net result is the same. In particular, the following equations hold definitionally:

```
example : foo 0     0     = 0 := rfl
example : foo 0     (n+1) = 0 := rfl
example : foo (m+1) 0     = 1 := rfl
example : foo (m+1) (n+1) = 2 := rfl
```

Since the values of `m` and `n` are not needed, we can just as well use wildcard patterns instead.

```
def foo : Nat → Nat → Nat
  | 0, _ => 0
  | _, 0 => 1
  | _, _ => 2
```

You can check that this definition of `foo` satisfies the same definitional identities as before.

Some functional programming languages support *incomplete patterns*. In these languages, the interpreter produces an exception or returns an arbitrary value for incomplete cases. We can simulate the arbitrary value approach using the `Inhabited` type class. Roughly, an element of `Inhabited α` is a witness to the fact that there is an element of `α`; in the Chapter Type Classes we will see that Lean can be instructed that suitable base types are inhabited, and can automatically infer that other constructed types are inhabited. On this basis, the standard library provides a default element, `default`, of any inhabited type.

We can also use the type `Option α` to simulate incomplete patterns. The idea is to return `some a` for the provided patterns, and use `none` for the incomplete cases. The following example demonstrates both approaches.

```
def f1 : Nat → Nat → Nat
  | 0, _  => 1
  | _, 0  => 2
  | _, _  => default  -- the "incomplete" case

example : f1 0     0     = 1       := rfl
example : f1 0     (a+1) = 1       := rfl
example : f1 (a+1) 0     = 2       := rfl
example : f1 (a+1) (b+1) = default := rfl

def f2 : Nat → Nat → Option Nat
  | 0, _  => some 1
  | _, 0  => some 2
  | _, _  => none     -- the "incomplete" case

example : f2 0     0     = some 1 := rfl
example : f2 0     (a+1) = some 1 := rfl
example : f2 (a+1) 0     = some 2 := rfl
example : f2 (a+1) (b+1) = none   := rfl
```

The equation compiler is clever. If you leave out any of the cases in the following definition, the error message will let you know what has not been covered.

```
def bar : Nat → List Nat → Bool → Nat
  | 0,   _,      false => 0
  | 0,   b :: _, _     => b
  | 0,   [],     true  => 7
  | a+1, [],     false => a
  | a+1, [],     true  => a + 1
  | a+1, b :: _, _     => a + b
```

It will also use an "if ... then ... else" instead of a `casesOn` in appropriate situations.

```
def foo : Char → Nat
  | 'A' => 1
  | 'B' => 2
  | _   => 3

#print foo.match_1
```

Structural Recursion and Induction

What makes the equation compiler powerful is that it also supports recursive definitions. In the next three sections, we will describe, respectively:

- structurally recursive definitions
- well-founded recursive definitions
- mutually recursive definitions

Generally speaking, the equation compiler processes input of the following form:

```
def foo (a : α) : (b : β) → γ
  | [patterns₁] => t₁
  ...
  | [patternsₙ] => tₙ
```

Here `(a : α)` is a sequence of parameters, `(b : β)` is the sequence of arguments on which pattern matching takes place, and `γ` is any type, which can depend on `a` and `b`. Each line should contain the same number of patterns, one for each element of `β`. As we have seen, a pattern is either a variable, a constructor applied to other patterns, or an expression that normalizes to something of that form (where the non-constructors are marked with the `[match_pattern]` attribute). The appearances of constructors prompt case splits, with the arguments to the constructors represented by the given variables. In Section Dependent Pattern Matching, we will see that it is sometimes necessary to include explicit terms in patterns that are needed to make an expression type check, though they do not play a role in pattern matching. These are called "inaccessible patterns" for that reason. But we will not need to use such inaccessible patterns before Section Dependent Pattern Matching.

As we saw in the last section, the terms `t₁, ..., tₙ` can make use of any of the parameters `a`, as well as any of the variables that are introduced in the corresponding patterns. What makes recursion and induction possible is that they can also involve recursive calls to `foo`. In this section, we will deal with *structural recursion*, in which the arguments to `foo` occurring on the right-hand side of the `=>` are subterms of the patterns on the left-hand side. The idea is that they are structurally smaller, and hence appear in the inductive type at an earlier stage. Here are some examples of structural recursion from the last chapter, now defined using the equation compiler:

```
open Nat
def add : Nat → Nat → Nat
  | m, zero   => m
  | m, succ n => succ (add m n)

theorem add_zero (m : Nat)   : add m zero = m := rfl
theorem add_succ (m n : Nat) : add m (succ n) = succ (add m n) := rfl

theorem zero_add : ∀ n, add zero n = n
  | zero   => rfl
  | succ n => congrArg succ (zero_add n)

def mul : Nat → Nat → Nat
  | n, zero   => zero
  | n, succ m => add (mul n m) n
```

The proof of `zero_add` makes it clear that proof by induction is really a form of recursion in Lean.

The example above shows that the defining equations for `add` hold definitionally, and the same is true of `mul`. The equation compiler tries to ensure that this holds whenever possible, as is the case with straightforward structural induction. In other situations, however, reductions hold only *propositionally*, which is to say, they are equational theorems that must be applied explicitly. The equation compiler generates such theorems internally. They are not meant to be used directly by the user; rather, the `simp` tactic is configured to use them when necessary. Thus both of the following proofs of `zero_add` work:

```
open Nat
theorem zero_add : ∀ n, add zero n = n
  | zero   => by simp [add]
  | succ n => by simp [add, zero_add]
```

As with definition by pattern matching, parameters to a structural recursion or induction may appear before the colon. Such parameters are simply added to the local context before the definition is processed. For example, the definition of addition may also be written as follows:

```
open Nat
def add (m : Nat) : Nat → Nat
  | zero   => m
  | succ n => succ (add m n)
```

You can also write the example above using `match`.

```
open Nat
def add (m n : Nat) : Nat :=
  match n with
  | zero   => m
  | succ n => succ (add m n)
```

A more interesting example of structural recursion is given by the Fibonacci function `fib`.

```
def fib : Nat → Nat
  | 0   => 1
  | 1   => 1
  | n+2 => fib (n+1) + fib n

example : fib 0 = 1 := rfl
example : fib 1 = 1 := rfl
example : fib (n + 2) = fib (n + 1) + fib n := rfl

example : fib 7 = 21 := rfl
```

Here, the value of the `fib` function at `n + 2` (which is definitionally equal to `succ (succ n)`) is defined in terms of the values at `n + 1` (which is definitionally equivalent to `succ n`) and the value at `n`. This is a notoriously inefficient way of computing the Fibonacci function, however, with an execution time that is exponential in `n`. Here is a better way:

```
def fibFast (n : Nat) : Nat :=
  (loop n).2
where
  loop : Nat → Nat × Nat
    | 0   => (0, 1)
    | n+1 => let p := loop n; (p.2, p.1 + p.2)

#eval fibFast 100
```

Here is the same definition using a `let rec` instead of a `where`.

```
def fibFast (n : Nat) : Nat :=
  let rec loop : Nat → Nat × Nat
    | 0   => (0, 1)
    | n+1 => let p := loop n; (p.2, p.1 + p.2)
  (loop n).2
```

In both cases, Lean generates the auxiliary function `fibFast.loop`.

To handle structural recursion, the equation compiler uses *course-of-values* recursion, using constants `below` and `brecOn` that are automatically generated with each inductively defined type. You can get a sense of how it works by looking at the types of `Nat.below` and `Nat.brecOn`:

```
variable (C : Nat → Type u)

#check (@Nat.below C : Nat → Type u)

#reduce @Nat.below C (3 : Nat)

#check (@Nat.brecOn C : (n : Nat) → ((n : Nat) → @Nat.below C n → C n) → C n)
```

The type `@Nat.below C (3 : nat)` is a data structure that stores elements of `C 0`, `C 1`, and `C 2`. The course-of-values recursion is implemented by `Nat.brecOn`. It enables us to define the value of a dependent function of type `(n : Nat) → C n` at a particular input `n` in terms of all the previous values of the function, presented as an element of `@Nat.below C n`.

The use of course-of-values recursion is one of the techniques the equation compiler uses to justify to the Lean kernel that a function terminates. It does not affect the code generator which compiles recursive functions as other functional programming language compilers. Recall that `#eval fib <n>` is exponential on `<n>`. On the other hand, `#reduce fib <n>` is efficient because it uses the definition sent to the kernel that is based on the `brecOn` construction.

```
def fib : Nat → Nat
  | 0   => 1
  | 1   => 1
  | n+2 => fib (n+1) + fib n

-- #eval fib 50 -- slow
#reduce fib 50  -- fast

#print fib
```

Another good example of a recursive definition is the list `append` function.

```
def append : List α → List α → List α
  | [],    bs => bs
  | a::as, bs => a :: append as bs

example : append [1, 2, 3] [4, 5] = [1, 2, 3, 4, 5] := rfl
```

Here is another: it adds elements of the first list to elements of the second list, until one of the two lists runs out.

```
def listAdd [Add α] : List α → List α → List α
  | [],      _       => []
  | _,       []      => []
  | a :: as, b :: bs => (a + b) :: listAdd as bs

#eval listAdd [1, 2, 3] [4, 5, 6, 6, 9, 10]
-- [5, 7, 9]
```

You are encouraged to experiment with similar examples in the exercises below.

Local recursive declarations

You can define local recursive declarations using the `let rec` keyword.

```
def replicate (n : Nat) (a : α) : List α :=
  let rec loop : Nat → List α → List α
    | 0,   as => as
    | n+1, as => loop n (a::as)
  loop n []

#check @replicate.loop
-- {α : Type} → α → Nat → List α → List α
```

Lean creates an auxiliary declaration for each `let rec`. In the example above, it created the declaration `replicate.loop` for the `let rec loop` occurring at `replicate`. Note that, Lean "closes" the declaration by adding any local variable occurring in the `let rec` declaration as additional parameters. For example, the local variable `a` occurs at `let rec loop`.

You can also use `let rec` in tactic mode and for creating proofs by induction.

```
theorem length_replicate (n : Nat) (a : α) : (replicate n a).length = n := by
  let rec aux (n : Nat) (as : List α)
              : (replicate.loop a n as).length = n + as.length := by
    match n with
    | 0   => simp [replicate.loop]
    | n+1 => simp [replicate.loop, aux n, Nat.add_succ, Nat.succ_add]
  exact aux n []
```

You can also introduce auxiliary recursive declarations using `where` clause after your definition. Lean converts them into a `let rec`.

```
def replicate (n : Nat) (a : α) : List α :=
  loop n []
where
  loop : Nat → List α → List α
    | 0,   as => as
    | n+1, as => loop n (a::as)

theorem length_replicate (n : Nat) (a : α) : (replicate n a).length = n := by
  exact aux n []
where
  aux (n : Nat) (as : List α)
      : (replicate.loop a n as).length = n + as.length := by
    match n with
    | 0   => simp [replicate.loop]
    | n+1 => simp [replicate.loop, aux n, Nat.add_succ, Nat.succ_add]
```

Well-Founded Recursion and Induction

When structural recursion cannot be used, we can prove termination using well-founded recursion. We need a well-founded relation and a proof that each recursive application is decreasing with respect to this relation. Dependent type theory is powerful enough to encode and justify well-founded recursion. Let us start with the logical background that is needed to understand how it works.

Lean's standard library defines two predicates, `Acc r a` and `WellFounded r`, where `r` is a binary relation on a type `α`, and `a` is an element of type `α`.

```
variable (α : Sort u)
variable (r : α → α → Prop)

#check (Acc r : α → Prop)
#check (WellFounded r : Prop)
```

The first, `Acc`, is an inductively defined predicate. According to its definition, `Acc r x` is equivalent to `∀ y, r y x → Acc r y`. If you think of `r y x` as denoting a kind of order relation `y < x`, then `Acc r x` says that `x` is accessible from below, in the sense that all its predecessors are accessible. In particular, if `x` has no predecessors, it is accessible. Given any type `α`, we should be able to assign a value to each accessible element of `α`, recursively, by assigning values to all its predecessors first.

The statement that `r` is well-founded, denoted `WellFounded r`, is exactly the statement that every element of the type is accessible. By the above considerations, if `r` is a well-founded relation on a type `α`, we should have a principle of well-founded recursion on `α`, with respect to the relation `r`. And, indeed, we do: the standard library defines `WellFounded.fix`, which serves exactly that purpose.

```
noncomputable def f {α : Sort u}
      (r : α → α → Prop)
      (h : WellFounded r)
      (C : α → Sort v)
      (F : (x : α) → ((y : α) → r y x → C y) → C x)
      : (x : α) → C x := WellFounded.fix h F
```

There is a long cast of characters here, but the first block we have already seen: the type, `α`, the relation, `r`, and the assumption, `h`, that `r` is well-founded. The variable `C` represents the motive of the recursive definition: for each element `x : α`, we would like to construct an element of `C x`. The function `F` provides the inductive recipe for doing that: it tells us how to construct an element `C x`, given elements of `C y` for each predecessor `y` of `x`.

Note that `WellFounded.fix` works equally well as an induction principle. It says that if `<` is well-founded and you want to prove `∀ x, C x`, it suffices to show that for an arbitrary `x`, if we have `∀ y < x, C y`, then we have `C x`.

In the example above we use the modifier `noncomputable` because the code generator currently does not support `WellFounded.fix`. The function `WellFounded.fix` is another tool Lean uses to justify that a function terminates.

Lean knows that the usual order `<` on the natural numbers is well founded. It also knows a number of ways of constructing new well founded orders from others, for example, using lexicographic order.

Here is essentially the definition of division on the natural numbers that is found in the standard library.

```
open Nat

theorem div_lemma {x y : Nat} : 0 < y ∧ y ≤ x → x - y < x :=
  fun h => sub_lt (Nat.lt_of_lt_of_le h.left h.right) h.left

def div.F (x : Nat) (f : (x₁ : Nat) → x₁ < x → Nat → Nat) (y : Nat) : Nat :=
  if h : 0 < y ∧ y ≤ x then
    f (x - y) (div_lemma h) y + 1
  else
    zero

noncomputable def div := WellFounded.fix (measure id).wf div.F

#reduce div 8 2 -- 4
```

The definition is somewhat inscrutable. Here the recursion is on `x`, and `div.F x f : Nat → Nat` returns the "divide by `y`" function for that fixed `x`. You have to remember that the second argument to `div.F`, the recipe for the recursion, is a function that is supposed to return the divide by `y` function for all values `x₁` smaller than `x`.

The elaborator is designed to make definitions like this more convenient. It accepts the following:

```
def div (x y : Nat) : Nat :=
  if h : 0 < y ∧ y ≤ x then
    have : x - y < x := Nat.sub_lt (Nat.lt_of_lt_of_le h.1 h.2) h.1
    div (x - y) y + 1
  else
    0
```

When Lean encounters a recursive definition, it first tries structural recursion, and only when that fails, does it fall back on well-founded recursion. Lean uses the tactic `decreasing_tactic` to show that the recursive applications are smaller. The auxiliary proposition `x - y < x` in the example above should be viewed as a hint for this tactic.

The defining equation for `div` does *not* hold definitionally, but we can unfold `div` using the `unfold` tactic. We use `conv` to select which `div` application we want to unfold.

```
example (x y : Nat) : div x y = if 0 < y ∧ y ≤ x then div (x - y) y + 1 else 0
:= by
  conv => lhs; unfold div -- unfold occurrence in the left-hand-side of the
equation

example (x y : Nat) (h : 0 < y ∧ y ≤ x) : div x y = div (x - y) y + 1 := by
  conv => lhs; unfold div
  simp [h]
```

The following example is similar: it converts any natural number to a binary expression, represented as a list of 0's and 1's. We have to provide evidence that the recursive call is decreasing, which we do here with a `sorry`. The `sorry` does not prevent the interpreter from evaluating the function successfully.

```
def natToBin : Nat → List Nat
  | 0     => [0]
  | 1     => [1]
  | n + 2 =>
    have : (n + 2) / 2 < n + 2 := sorry
    natToBin ((n + 2) / 2) ++ [n % 2]

#eval natToBin 1234567
```

As a final example, we observe that Ackermann's function can be defined directly, because it is justified by the well-foundedness of the lexicographic order on the natural numbers. The `termination_by` clause instructs Lean to use a lexicographic order. This clause is actually mapping the function arguments to elements of type `Nat × Nat`. Then, Lean uses typeclass resolution to synthesize an element of type `WellFoundedRelation (Nat × Nat)`.

```
def ack : Nat → Nat → Nat
  | 0,   y   => y+1
  | x+1, 0   => ack x 1
  | x+1, y+1 => ack x (ack (x+1) y)
termination_by x y => (x, y)
```

Note that a lexicographic order is used in the example above because the instance `WellFoundedRelation (α × β)` uses a lexicographic order. Lean also defines the instance

```
instance (priority := low) [SizeOf α] : WellFoundedRelation α :=
  sizeOfWFRel
```

In the following example, we prove termination by showing that `as.size - i` is decreasing in the recursive application.

```
def takeWhile (p : α → Bool) (as : Array α) : Array α :=
  go 0 #[]
where
  go (i : Nat) (r : Array α) : Array α :=
    if h : i < as.size then
      let a := as.get ⟨i, h⟩
      if p a then
        go (i+1) (r.push a)
      else
        r
    else
      r
  termination_by as.size - i
```

Note that, auxiliary function `go` is recursive in this example, but `takeWhile` is not.

By default, Lean uses the tactic `decreasing_tactic` to prove recursive applications are decreasing. The modifier `decreasing_by` allows us to provide our own tactic. Here is an example.

```
theorem div_lemma {x y : Nat} : 0 < y ∧ y ≤ x → x - y < x :=
  fun ⟨ypos, ylex⟩ => Nat.sub_lt (Nat.lt_of_lt_of_le ypos ylex) ypos

def div (x y : Nat) : Nat :=
  if h : 0 < y ∧ y ≤ x then
    div (x - y) y + 1
  else
    0
decreasing_by apply div_lemma; assumption
```

Note that `decreasing_by` is not replacement for `termination_by`, they complement each other. `termination_by` is used to specify a well-founded relation, and `decreasing_by` for providing our own tactic for showing recursive applications are decreasing. In the following example, we use both of them.

```
def ack : Nat → Nat → Nat
  | 0,   y   => y+1
  | x+1, 0   => ack x 1
  | x+1, y+1 => ack x (ack (x+1) y)
termination_by x y => (x, y)
decreasing_by
  all_goals simp_wf -- unfolds well-founded recursion auxiliary definitions
  · apply Prod.Lex.left; simp_arith
  · apply Prod.Lex.right; simp_arith
  · apply Prod.Lex.left; simp_arith
```

We can use `decreasing_by sorry` to instruct Lean to "trust" us that the function terminates.

```
def natToBin : Nat → List Nat
  | 0     => [0]
  | 1     => [1]
  | n + 2 => natToBin ((n + 2) / 2) ++ [n % 2]
decreasing_by sorry

#eval natToBin 1234567
```

Recall that using `sorry` is equivalent to using a new axiom, and should be avoided. In the following example, we used the `sorry` to prove `False`. The command `#print axioms unsound` shows that `unsound` depends on the unsound axiom `sorryAx` used to implement `sorry`.

```
def unsound (x : Nat) : False :=
  unsound (x + 1)
decreasing_by sorry

#check unsound 0
-- `unsound 0` is a proof of `False`

#print axioms unsound
-- 'unsound' depends on axioms: [sorryAx]
```

Summary:

- If there is no `termination_by`, a well-founded relation is derived (if possible) by selecting an argument and then using typeclass resolution to synthesize a well-founded relation for this argument's type.
- If `termination_by` is specified, it maps the arguments of the function to a type `α` and type class resolution is again used. Recall that, the default instance for `β × γ` is a lexicographic order based on the well-founded relations for `β` and `γ`.
- The default well-founded relation instance for `Nat` is `<`.
- By default, the tactic `decreasing_tactic` is used to show that recursive applications are smaller with respect to the selected well-founded relation. If `decreasing_tactic` fails, the error message includes the remaining goal `... |- G`. Note that, the `decreasing_tactic` uses `assumption`. So, you can include a `have`-expression to prove goal `G`. You can also provide your own tactic using `decreasing_by`.

Mutual Recursion

Lean also supports mutual recursive definitions. The syntax is similar to that for mutual inductive types. Here is an example:

```
mutual
  def even : Nat → Bool
    | 0   => true
    | n+1 => odd n

  def odd : Nat → Bool
    | 0   => false
    | n+1 => even n
end

example : even (a + 1) = odd a := by
  simp [even]

example : odd (a + 1) = even a := by
  simp [odd]

theorem even_eq_not_odd : ∀ a, even a = not (odd a) := by
  intro a; induction a
  . simp [even, odd]
  . simp [even, odd, *]
```

What makes this a mutual definition is that `even` is defined recursively in terms of `odd`, while `odd` is defined recursively in terms of `even`. Under the hood, this is compiled as a single recursive definition. The internally defined function takes, as argument, an element of a sum type, either an input to `even`, or an input to `odd`. It then returns an output appropriate to the input. To define that function, Lean uses a suitable well-founded measure. The internals are meant to be hidden from users; the canonical way to make use of such definitions is to use `simp` (or `unfold`), as we did above.

Mutual recursive definitions also provide natural ways of working with mutual and nested inductive types. Recall the definition of `Even` and `Odd` as mutual inductive predicates as presented before.

```
mutual
  inductive Even : Nat → Prop where
    | even_zero : Even 0
    | even_succ : ∀ n, Odd n → Even (n + 1)

  inductive Odd : Nat → Prop where
    | odd_succ : ∀ n, Even n → Odd (n + 1)
end
```

The constructors, `even_zero`, `even_succ`, and `odd_succ` provide positive means for showing that a number is even or odd. We need to use the fact that the inductive type is generated by these constructors to know that zero is not odd, and that the latter two implications reverse. As usual, the constructors are kept in a namespace that is named after the type being defined, and the command `open Even Odd` allows us to access them more conveniently.

```
open Even Odd

theorem not_odd_zero : ¬ Odd 0 :=
  fun h => nomatch h

theorem even_of_odd_succ : ∀ n, Odd (n + 1) → Even n
  | _, odd_succ n h => h

theorem odd_of_even_succ : ∀ n, Even (n + 1) → Odd n
  | _, even_succ n h => h
```

For another example, suppose we use a nested inductive type to define a set of terms inductively, so that a term is either a constant (with a name given by a string), or the result of applying a constant to a list of constants.

```
inductive Term where
  | const : String → Term
  | app   : String → List Term → Term
```

We can then use a mutual recursive definition to count the number of constants occurring in a term, as well as the number occurring in a list of terms.

```
namespace Term

mutual
  def numConsts : Term → Nat
    | const _ => 1
    | app _ cs => numConstsLst cs

  def numConstsLst : List Term → Nat
    | [] => 0
    | c :: cs => numConsts c + numConstsLst cs
end

def sample := app "f" [app "g" [const "x"], const "y"]

#eval numConsts sample

end Term
```

As a final example, we define a function `replaceConst a b e` that replaces a constant `a` with `b` in a term `e`, and then prove the number of constants is the same. Note that, our proof uses mutual recursion (aka induction).

```
mutual
  def replaceConst (a b : String) : Term → Term
    | const c => if a == c then const b else const c
    | app f cs => app f (replaceConstLst a b cs)

  def replaceConstLst (a b : String) : List Term → List Term
    | [] => []
    | c :: cs => replaceConst a b c :: replaceConstLst a b cs
end

mutual
  theorem numConsts_replaceConst (a b : String) (e : Term)
            : numConsts (replaceConst a b e) = numConsts e := by
    match e with
    | const c => simp [replaceConst]; split <;> simp [numConsts]
    | app f cs => simp [replaceConst, numConsts, numConsts_replaceConstLst a b
cs]

  theorem numConsts_replaceConstLst (a b : String) (es : List Term)
            : numConstsLst (replaceConstLst a b es) = numConstsLst es := by
    match es with
    | [] => simp [replaceConstLst, numConstsLst]
    | c :: cs =>
      simp [replaceConstLst, numConstsLst, numConsts_replaceConst a b c,
            numConsts_replaceConstLst a b cs]
end
```

Dependent Pattern Matching

All the examples of pattern matching we considered in Section Pattern Matching can easily be written using `casesOn` and `recOn`. However, this is often not the case with indexed inductive families such as `Vector α n`, since case splits impose constraints on the values of the indices. Without the equation compiler, we would need a lot of boilerplate code to define very simple functions such as `map`, `zip`, and `unzip` using recursors. To understand the difficulty, consider what it would take to define a function `tail` which takes a vector `v : Vector α (succ n)` and deletes the first element. A first thought might be to use the `casesOn` function:

```
inductive Vector (α : Type u) : Nat → Type u
  | nil  : Vector α 0
  | cons : α → {n : Nat} → Vector α n → Vector α (n+1)

namespace Vector

#check @Vector.casesOn
/-
  {α : Type u}
  → {motive : (a : Nat) → Vector α a → Sort v} →
  → {a : Nat} → (t : Vector α a)
  → motive 0 nil
  → ((a : α) → {n : Nat} → (a_1 : Vector α n) → motive (n + 1) (cons a a_1))
  → motive a t
-/

end Vector
```

But what value should we return in the `nil` case? Something funny is going on: if `v` has type `Vector α (succ n)`, it *can't* be nil, but it is not clear how to tell that to `casesOn`.

One solution is to define an auxiliary function:

```
def tailAux (v : Vector α m) : m = n + 1 → Vector α n :=
  Vector.casesOn (motive := fun x _ => x = n + 1 → Vector α n) v
    (fun h : 0 = n + 1 => Nat.noConfusion h)
    (fun (a : α) (m : Nat) (as : Vector α m) =>
     fun (h : m + 1 = n + 1) =>
       Nat.noConfusion h (fun h1 : m = n => h1 ▸ as))

def tail (v : Vector α (n+1)) : Vector α n :=
  tailAux v rfl
```

In the `nil` case, `m` is instantiated to `0`, and `noConfusion` makes use of the fact that `0 = succ n` cannot occur. Otherwise, `v` is of the form `a :: w`, and we can simply return `w`, after casting it from a vector of length `m` to a vector of length `n`.

The difficulty in defining `tail` is to maintain the relationships between the indices. The hypothesis `e : m = n + 1` in `tailAux` is used to communicate the relationship between `n` and the index associated with the minor premise. Moreover, the `zero = n + 1` case is unreachable, and the canonical way to discard such a case is to use `noConfusion`.

The `tail` function is, however, easy to define using recursive equations, and the equation compiler generates all the boilerplate code automatically for us. Here are a number of similar examples:

```
def head : {n : Nat} → Vector α (n+1) → α
  | n, cons a as => a

def tail : {n : Nat} → Vector α (n+1) → Vector α n
  | n, cons a as => as

theorem eta : ∀ {n : Nat} (v : Vector α (n+1)), cons (head v) (tail v) = v
  | n, cons a as => rfl

def map (f : α → β → γ) : {n : Nat} → Vector α n → Vector β n → Vector γ n
  | 0,   nil,       nil       => nil
  | n+1, cons a as, cons b bs => cons (f a b) (map f as bs)

def zip : {n : Nat} → Vector α n → Vector β n → Vector (α × β) n
  | 0,   nil,       nil       => nil
  | n+1, cons a as, cons b bs => cons (a, b) (zip as bs)
```

Note that we can omit recursive equations for "unreachable" cases such as `head nil`. The automatically generated definitions for indexed families are far from straightforward. For example:

```
def map (f : α → β → γ) : {n : Nat} → Vector α n → Vector β n → Vector γ n
  | 0,   nil,       nil       => nil
  | n+1, cons a as, cons b bs => cons (f a b) (map f as bs)

#print map
#print map.match_1
```

The `map` function is even more tedious to define by hand than the `tail` function. We encourage you to try it, using `recOn`, `casesOn` and `noConfusion`.

Inaccessible Patterns

Sometimes an argument in a dependent matching pattern is not essential to the definition, but nonetheless has to be included to specialize the type of the expression appropriately. Lean allows users to mark such subterms as *inaccessible* for pattern matching. These annotations are essential, for example, when a term occurring in the left-hand side is neither a variable nor a constructor application, because these are not suitable targets for pattern matching. We can view such inaccessible patterns as "don't care" components of the patterns. You can declare a subterm inaccessible by writing `.(t)`. If the inaccessible pattern can be inferred, you can also write `_`.

The following example, we declare an inductive type that defines the property of "being in the image of `f`". You can view an element of the type `ImageOf f b` as evidence that `b` is in the image of `f`, whereby the constructor `imf` is used to build such evidence. We can then define any function `f` with an "inverse" which takes anything in the image of `f` to an element that is mapped to it. The typing rules forces us to write `f a` for the first argument,

but this term is neither a variable nor a constructor application, and plays no role in the pattern-matching definition. To define the function `inverse` below, we *have to* mark `f a` inaccessible.

```
inductive ImageOf {α β : Type u} (f : α → β) : β → Type u where
  | imf : (a : α) → ImageOf f (f a)

open ImageOf

def inverse {f : α → β} : (b : β) → ImageOf f b → α
  | .(f a), imf a => a

def inverse' {f : α → β} : (b : β) → ImageOf f b → α
  | _, imf a => a
```

In the example above, the inaccessible annotation makes it clear that `f` is *not* a pattern matching variable.

Inaccessible patterns can be used to clarify and control definitions that make use of dependent pattern matching. Consider the following definition of the function `Vector.add`, which adds two vectors of elements of a type, assuming that type has an associated addition function:

```
inductive Vector (α : Type u) : Nat → Type u
  | nil  : Vector α 0
  | cons : α → {n : Nat} → Vector α n → Vector α (n+1)

namespace Vector

def add [Add α] : {n : Nat} → Vector α n → Vector α n → Vector α n
  | 0,   nil,       nil       => nil
  | n+1, cons a as, cons b bs => cons (a + b) (add as bs)

end Vector
```

The argument `{n : Nat}` appear after the colon, because it cannot be held fixed throughout the definition. When implementing this definition, the equation compiler starts with a case distinction as to whether the first argument is `0` or of the form `n+1`. This is followed by nested case splits on the next two arguments, and in each case the equation compiler rules out the cases are not compatible with the first pattern.

But, in fact, a case split is not required on the first argument; the `casesOn` eliminator for `Vector` automatically abstracts this argument and replaces it by `0` and `n + 1` when we do a case split on the second argument. Using inaccessible patterns, we can prompt the equation compiler to avoid the case split on `n`

```
def add [Add α] : {n : Nat} → Vector α n → Vector α n → Vector α n
  | .(_), nil,       nil       => nil
  | .(_), cons a as, cons b bs => cons (a + b) (add as bs)
```

Marking the position as an inaccessible pattern tells the equation compiler first, that the form of the argument should be inferred from the constraints posed by the other arguments, and, second, that the first argument should *not* participate in pattern matching.

The inaccessible pattern `.(_)` can be written as `_` for convenience.

```
def add [Add α] : {n : Nat} → Vector α n → Vector α n → Vector α n
  | _, nil,       nil       => nil
  | _, cons a as, cons b bs => cons (a + b) (add as bs)
```

As we mentioned above, the argument `{n : Nat}` is part of the pattern matching, because it cannot be held fixed throughout the definition. In previous Lean versions, users often found it cumbersome to have to include these extra discriminants. Thus, Lean 4 implements a new feature, *discriminant refinement*, which includes these extra discriminants automatically for us.

```
def add [Add α] {n : Nat} : Vector α n → Vector α n → Vector α n
  | nil,       nil       => nil
  | cons a as, cons b bs => cons (a + b) (add as bs)
```

When combined with the *auto bound implicits* feature, you can simplify the declare further and write:

```
def add [Add α] : Vector α n → Vector α n → Vector α n
  | nil,       nil       => nil
  | cons a as, cons b bs => cons (a + b) (add as bs)
```

Using these new features, you can write the other vector functions defined in the previous sections more compactly as follows:

```
def head : Vector α (n+1) → α
  | cons a as => a

def tail : Vector α (n+1) → Vector α n
  | cons a as => as

theorem eta : (v : Vector α (n+1)) → cons (head v) (tail v) = v
  | cons a as => rfl

def map (f : α → β → γ) : Vector α n → Vector β n → Vector γ n
  | nil,       nil       => nil
  | cons a as, cons b bs => cons (f a b) (map f as bs)

def zip : Vector α n → Vector β n → Vector (α × β) n
  | nil,       nil       => nil
  | cons a as, cons b bs => cons (a, b) (zip as bs)
```

Match Expressions

Lean also provides a compiler for *match-with* expressions found in many functional languages:

```
def isNotZero (m : Nat) : Bool :=
  match m with
  | 0   => false
  | n+1 => true
```

This does not look very different from an ordinary pattern matching definition, but the point is that a `match` can be used anywhere in an expression, and with arbitrary arguments.

```
def isNotZero (m : Nat) : Bool :=
  match m with
  | 0   => false
  | n+1 => true

def filter (p : α → Bool) : List α → List α
  | []      => []
  | a :: as =>
    match p a with
    | true => a :: filter p as
    | false => filter p as

example : filter isNotZero [1, 0, 0, 3, 0] = [1, 3] := rfl
```

Here is another example:

```
def foo (n : Nat) (b c : Bool) :=
  5 + match n - 5, b && c with
      | 0,   true  => 0
      | m+1, true  => m + 7
      | 0,   false => 5
      | m+1, false => m + 3

#eval foo 7 true false

example : foo 7 true false = 9 := rfl
```

Lean uses the `match` construct internally to implement pattern-matching in all parts of the system. Thus, all four of these definitions have the same net effect:

```
def bar₁ : Nat × Nat → Nat
  | (m, n) => m + n

def bar₂ (p : Nat × Nat) : Nat :=
  match p with
  | (m, n) => m + n

def bar₃ : Nat × Nat → Nat :=
  fun (m, n) => m + n

def bar₄ (p : Nat × Nat) : Nat :=
  let (m, n) := p; m + n
```

These variations are equally useful for destructing propositions:

```
variable (p q : Nat → Prop)

example : (∃ x, p x) → (∃ y, q y) → ∃ x y, p x ∧ q y
  | ⟨x, px⟩, ⟨y, qy⟩ => ⟨x, y, px, qy⟩

example (h₀ : ∃ x, p x) (h₁ : ∃ y, q y)
        : ∃ x y, p x ∧ q y :=
  match h₀, h₁ with
  | ⟨x, px⟩, ⟨y, qy⟩ => ⟨x, y, px, qy⟩

example : (∃ x, p x) → (∃ y, q y) → ∃ x y, p x ∧ q y :=
  fun ⟨x, px⟩ ⟨y, qy⟩ => ⟨x, y, px, qy⟩

example (h₀ : ∃ x, p x) (h₁ : ∃ y, q y)
        : ∃ x y, p x ∧ q y :=
  let ⟨x, px⟩ := h₀
  let ⟨y, qy⟩ := h₁
  ⟨x, y, px, qy⟩
```

Local Recursive Declarations

You can define local recursive declarations using the `let rec` keyword:

```
def replicate (n : Nat) (a : α) : List α :=
  let rec loop : Nat → List α → List α
    | 0,   as => as
    | n+1, as => loop n (a::as)
  loop n []

#check @replicate.loop
-- {α : Type} → α → Nat → List α → List α
```

Lean creates an auxiliary declaration for each `let rec`. In the example above, it created the declaration `replicate.loop` for the `let rec loop` occurring at `replicate`. Note that, Lean "closes" the declaration by adding any local variable occurring in the `let rec` declaration as additional parameters. For example, the local variable `a` occurs at `let rec loop`.

You can also use `let rec` in tactic mode and for creating proofs by induction:

```
theorem length_replicate (n : Nat) (a : α) : (replicate n a).length = n := by
  let rec aux (n : Nat) (as : List α)
            : (replicate.loop a n as).length = n + as.length := by
    match n with
    | 0   => simp [replicate.loop]
    | n+1 => simp [replicate.loop, aux n, Nat.add_succ, Nat.succ_add]
  exact aux n []
```

You can also introduce auxiliary recursive declarations using a `where` clause after your definition. Lean converts them into a `let rec`:

```
def replicate (n : Nat) (a : α) : List α :=
  loop n []
where
  loop : Nat → List α → List α
    | 0,   as => as
    | n+1, as => loop n (a::as)

theorem length_replicate (n : Nat) (a : α) : (replicate n a).length = n := by
  exact aux n []
where
  aux (n : Nat) (as : List α)
      : (replicate.loop a n as).length = n + as.length := by
    match n with
    | 0   => simp [replicate.loop]
    | n+1 => simp [replicate.loop, aux n, Nat.add_succ, Nat.succ_add]
```

Exercises

1. Open a namespace `Hidden` to avoid naming conflicts, and use the equation compiler to define addition, multiplication, and exponentiation on the natural numbers. Then use the equation compiler to derive some of their basic properties.

2. Similarly, use the equation compiler to define some basic operations on lists (like the `reverse` function) and prove theorems about lists by induction (such as the fact that `reverse (reverse xs) = xs` for any list `xs`).

3. Define your own function to carry out course-of-value recursion on the natural numbers. Similarly, see if you can figure out how to define `WellFounded.fix` on your own.

4. Following the examples in Section Dependent Pattern Matching, define a function that will append two vectors. This is tricky; you will have to define an auxiliary function.

5. Consider the following type of arithmetic expressions. The idea is that `var n` is a variable, v_n, and `const n` is the constant whose value is `n`.

```
inductive Expr where
  | const : Nat → Expr
  | var : Nat → Expr
  | plus : Expr → Expr → Expr
  | times : Expr → Expr → Expr
  deriving Repr

open Expr

def sampleExpr : Expr :=
  plus (times (var 0) (const 7)) (times (const 2) (var 1))
```

Here `sampleExpr` represents `(v₀ * 7) + (2 * v₁)`.

Write a function that evaluates such an expression, evaluating each `var n` to `v n`.

```
def eval (v : Nat → Nat) : Expr → Nat
  | const n     => sorry
  | var n       => v n
  | plus e₁ e₂  => sorry
  | times e₁ e₂ => sorry

def sampleVal : Nat → Nat
  | 0 => 5
  | 1 => 6
  | _ => 0

-- Try it out. You should get 47 here.
-- #eval eval sampleVal sampleExpr
```

Implement "constant fusion," a procedure that simplifies subterms like `5 + 7` to `12`. Using the auxiliary function `simpConst`, define a function "fuse": to simplify a plus or a times, first simplify the arguments recursively, and then apply `simpConst` to try to simplify the result.

```
def simpConst : Expr → Expr
  | plus (const n₁) (const n₂)  => const (n₁ + n₂)
  | times (const n₁) (const n₂) => const (n₁ * n₂)
  | e                           => e

def fuse : Expr → Expr := sorry

theorem simpConst_eq (v : Nat → Nat)
        : ∀ e : Expr, eval v (simpConst e) = eval v e :=
  sorry

theorem fuse_eq (v : Nat → Nat)
        : ∀ e : Expr, eval v (fuse e) = eval v e :=
  sorry
```

The last two theorems show that the definitions preserve the value.

Structures and Records

We have seen that Lean's foundational system includes inductive types. We have, moreover, noted that it is a remarkable fact that it is possible to construct a substantial edifice of mathematics based on nothing more than the type universes, dependent arrow types, and inductive types; everything else follows from those. The Lean standard library contains many instances of inductive types (e.g., `Nat`, `Prod`, `List`), and even the logical connectives are defined using inductive types.

Recall that a non-recursive inductive type that contains only one constructor is called a *structure* or *record*. The product type is a structure, as is the dependent product (Sigma) type. In general, whenever we define a structure `S`, we usually define *projection* functions that allow us to "destruct" each instance of `S` and retrieve the values that are stored in its fields. The functions `prod.fst` and `prod.snd`, which return the first and second elements of a pair, are examples of such projections.

When writing programs or formalizing mathematics, it is not uncommon to define structures containing many fields. The `structure` command, available in Lean, provides infrastructure to support this process. When we define a structure using this command, Lean automatically generates all the projection functions. The `structure` command also allows us to define new structures based on previously defined ones. Moreover, Lean provides convenient notation for defining instances of a given structure.

Declaring Structures

The structure command is essentially a "front end" for defining inductive data types. Every `structure` declaration introduces a namespace with the same name. The general form is as follows:

```
structure <name> <parameters> <parent-structures> where
  <constructor> :: <fields>
```

Most parts are optional. Here is an example:

```
structure Point (α : Type u) where
  mk :: (x : α) (y : α)
```

Values of type `Point` are created using `Point.mk a b`, and the fields of a point `p` are accessed using `Point.x p` and `Point.y p` (but `p.x` and `p.y` also work, see below). The structure command also generates useful recursors and theorems. Here are some of the constructions generated for the declaration above.

```
#check Point       -- a Type
#check @Point.rec  -- the eliminator
#check @Point.mk   -- the constructor
#check @Point.x    -- a projection
#check @Point.y    -- a projection
```

If the constructor name is not provided, then a constructor is named `mk` by default. You can also avoid the parentheses around field names if you add a line break between each field.

```
structure Point (α : Type u) where
  x : α
  y : α
```

Here are some simple theorems and expressions that use the generated constructions. As usual, you can avoid the prefix `Point` by using the command `open Point`.

```
#eval Point.x (Point.mk 10 20)
#eval Point.y (Point.mk 10 20)

open Point

example (a b : α) : x (mk a b) = a :=
  rfl

example (a b : α) : y (mk a b) = b :=
  rfl
```

Given `p : Point Nat`, the dot notation `p.x` is shorthand for `Point.x p`. This provides a convenient way of accessing the fields of a structure.

```
def p := Point.mk 10 20

#check p.x  -- Nat
#eval p.x   -- 10
#eval p.y   -- 20
```

The dot notation is convenient not just for accessing the projections of a record, but also for applying functions defined in a namespace with the same name. Recall from the Conjunction section if `p` has type `Point`, the expression `p.foo` is interpreted as `Point.foo p`, assuming that the first non-implicit argument to `foo` has type `Point`. The expression `p.add q` is therefore shorthand for `Point.add p q` in the example below.

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

def Point.add (p q : Point Nat) :=
  mk (p.x + q.x) (p.y + q.y)

def p : Point Nat := Point.mk 1 2
def q : Point Nat := Point.mk 3 4

#eval p.add q  -- {x := 4, y := 6}
```

In the next chapter, you will learn how to define a function like `add` so that it works generically for elements of `Point α` rather than just `Point Nat`, assuming `α` has an associated addition operation.

More generally, given an expression `p.foo x y z` where `p : Point`, Lean will insert `p` at the first argument to `Point.foo` of type `Point`. For example, with the definition of scalar multiplication below, `p.smul 3` is interpreted as `Point.smul 3 p`.

```
def Point.smul (n : Nat) (p : Point Nat) :=
  Point.mk (n * p.x) (n * p.y)

def p : Point Nat := Point.mk 1 2

#eval p.smul 3  -- {x := 3, y := 6}
```

It is common to use a similar trick with the `List.map` function, which takes a list as its second non-implicit argument:

```
#check @List.map

def xs : List Nat := [1, 2, 3]
def f : Nat → Nat := fun x => x * x

#eval xs.map f  -- [1, 4, 9]
```

Here `xs.map f` is interpreted as `List.map f xs`.

Objects

We have been using constructors to create elements of a structure type. For structures containing many fields, this is often inconvenient, because we have to remember the order in which the fields were defined. Lean therefore provides the following alternative notations for defining elements of a structure type.

```
    { (<field-name> := <expr>)* : structure-type }
    or
    { (<field-name> := <expr>)* }
```

The suffix `: structure-type` can be omitted whenever the name of the structure can be inferred from the expected type. For example, we use this notation to define "points." The order that the fields are specified does not matter, so all the expressions below define the same point.

```
structure Point (α : Type u) where
  x : α
  y : α

#check { x := 10, y := 20 : Point Nat }  -- Point ℕ
#check { y := 20, x := 10 : Point _ }
#check ({ x := 10, y := 20 } : Point Nat)

example : Point Nat :=
  { y := 20, x := 10 }
```

If the value of a field is not specified, Lean tries to infer it. If the unspecified fields cannot be inferred, Lean flags an error indicating the corresponding placeholder could not be synthesized.

```
structure MyStruct where
    {α : Type u}
    {β : Type v}
    a : α
    b : β

#check { a := 10, b := true : MyStruct }
```

Record update is another common operation which amounts to creating a new record object by modifying the value of one or more fields in an old one. Lean allows you to specify that unassigned fields in the specification of a record should be taken from a previously defined structure object `s` by adding the annotation `s with` before the field assignments. If more than one record object is provided, then they are visited in order until Lean finds one that contains the unspecified field. Lean raises an error if any of the field names remain unspecified after all the objects are visited.

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

def p : Point Nat :=
  { x := 1, y := 2 }

#eval { p with y := 3 }  -- { x := 1, y := 3 }
#eval { p with x := 4 }  -- { x := 4, y := 2 }

structure Point3 (α : Type u) where
  x : α
  y : α
  z : α

def q : Point3 Nat :=
  { x := 5, y := 5, z := 5 }

def r : Point3 Nat :=
  { p, q with x := 6 }

example : r.x = 6 := rfl
example : r.y = 2 := rfl
example : r.z = 5 := rfl
```

Inheritance

We can *extend* existing structures by adding new fields. This feature allows us to simulate a form of *inheritance*.

```
structure Point (α : Type u) where
  x : α
  y : α

inductive Color where
  | red | green | blue

structure ColorPoint (α : Type u) extends Point α where
  c : Color
```

In the next example, we define a structure using multiple inheritance, and then define an object using objects of the parent structures.

```
structure Point (α : Type u) where
  x : α
  y : α
  z : α

structure RGBValue where
  red : Nat
  green : Nat
  blue : Nat

structure RedGreenPoint (α : Type u) extends Point α, RGBValue where
  no_blue : blue = 0

def p : Point Nat :=
  { x := 10, y := 10, z := 20 }

def rgp : RedGreenPoint Nat :=
  { p with red := 200, green := 40, blue := 0, no_blue := rfl }

example : rgp.x   = 10 := rfl
example : rgp.red = 200 := rfl
```

Type Classes

Type classes were introduced as a principled way of enabling ad-hoc polymorphism in functional programming languages. We first observe that it would be easy to implement an ad-hoc polymorphic function (such as addition) if the function simply took the type-specific implementation of addition as an argument and then called that implementation on the remaining arguments. For example, suppose we declare a structure in Lean to hold implementations of addition.

```
structure Add (a : Type) where
  add : a → a → a

#check @Add.add
-- Add.add : {a : Type} → Add a → a → a → a
```

In the above Lean code, the field `add` has type `Add.add : {a : Type} → Add a → a → a → a` where the curly braces around the type `a` mean that it is an implicit argument. We could implement `double` by:

```
def double (s : Add a) (x : a) : a :=
  s.add x x

#eval double { add := Nat.add } 10
-- 20

#eval double { add := Nat.mul } 10
-- 100

#eval double { add := Int.add } 10
-- 20
```

Note that you can double a natural number `n` by `double { add := Nat.add } n`. Of course, it would be highly cumbersome for users to manually pass the implementations around in this way. Indeed, it would defeat most of the potential benefits of ad-hoc polymorphism.

The main idea behind type classes is to make arguments such as `Add a` implicit, and to use a database of user-defined instances to synthesize the desired instances automatically through a process known as typeclass resolution. In Lean, by changing `structure` to `class` in the example above, the type of `Add.add` becomes:

```
class Add (a : Type) where
  add : a → a → a

#check @Add.add
-- Add.add : {a : Type} → [self : Add a] → a → a → a
```

where the square brackets indicate that the argument of type `Add a` is *instance implicit*, i.e. that it should be synthesized using typeclass resolution. This version of `add` is the Lean analogue of the Haskell term `add :: Add a => a -> a -> a`. Similarly, we can register instances by:

```
instance : Add Nat where
  add := Nat.add

instance : Add Int where
  add := Int.add

instance : Add Float where
  add := Float.add
```

Then for `n : Nat` and `m : Nat`, the term `Add.add n m` triggers typeclass resolution with the goal of `Add Nat`, and typeclass resolution will synthesize the instance for `Nat` above. We can now reimplement `double` using an instance implicit by:

```
def double [Add a] (x : a) : a :=
  Add.add x x

#check @double
-- @double : {a : Type} → [inst : Add a] → a → a

#eval double 10
-- 20

#eval double (10 : Int)
-- 100

#eval double (7 : Float)
-- 14.000000

#eval double (239.0 + 2)
-- 482.000000
```

In general, instances may depend on other instances in complicated ways. For example, you can declare an (anonymous) instance stating that if `a` has addition, then `Array a` has addition:

```
instance [Add a] : Add (Array a) where
  add x y := Array.zipWith x y (· + ·)

#eval Add.add #[1, 2] #[3, 4]
-- #[4, 6]

#eval #[1, 2] + #[3, 4]
-- #[4, 6]
```

Note that `(· + ·)` is notation for `fun x y => x + y` in Lean.

The example above demonstrates how type classes are used to overload notation. Now, we explore another application. We often need an arbitrary element of a given type. Recall that types may not have any elements in Lean. It often happens that we would like a definition to return an arbitrary element in a "corner case." For example, we may like the expression `head xs` to be of type `a` when `xs` is of type `List a`. Similarly, many theorems hold under the additional assumption that a type is not empty. For example, if `a` is a type, `exists x : a, x = x` is true only if `a` is not empty. The standard library defines a type class `Inhabited` to enable type class inference to infer a "default" element of an inhabited type. Let us start with the first step of the program above, declaring an appropriate class:

```
class Inhabited (a : Type u) where
  default : a

#check @Inhabited.default
-- Inhabited.default : {a : Type u} → [self : Inhabited a] → a
```

Note `Inhabited.default` doesn't have any explicit arguments.

An element of the class `Inhabited a` is simply an expression of the form `Inhabited.mk x`, for some element `x : a`. The projection `Inhabited.default` will allow us to "extract" such an element of `a` from an element of `Inhabited a`. Now we populate the class with some instances:

```
instance : Inhabited Bool where
  default := true

instance : Inhabited Nat where
  default := 0

instance : Inhabited Unit where
  default := ()

instance : Inhabited Prop where
  default := True

#eval (Inhabited.default : Nat)
-- 0

#eval (Inhabited.default : Bool)
-- true
```

You can use the command `export` to create the alias `default` for `Inhabited.default`

```
export Inhabited (default)

#eval (default : Nat)
-- 0

#eval (default : Bool)
-- true
```

Chaining Instances

If that were the extent of type class inference, it would not be all that impressive; it would be simply a mechanism of storing a list of instances for the elaborator to find in a lookup table. What makes type class inference powerful is that one can *chain* instances. That is, an instance declaration can in turn depend on an implicit instance of a type class. This causes class inference to chain through instances recursively, backtracking when necessary, in a Prolog-like search.

For example, the following definition shows that if two types `a` and `b` are inhabited, then so is their product:

```
instance [Inhabited a] [Inhabited b] : Inhabited (a × b) where
  default := (default, default)
```

With this added to the earlier instance declarations, type class instance can infer, for example, a default element of `Nat × Bool`:

```
instance [Inhabited a] [Inhabited b] : Inhabited (a × b) where
  default := (default, default)

#eval (default : Nat × Bool)
-- (0, true)
```

Similarly, we can inhabit type function with suitable constant functions:

```
instance [Inhabited b] : Inhabited (a → b) where
  default := fun _ => default
```

As an exercise, try defining default instances for other types, such as `List` and `Sum` types.

The Lean standard library contains the definition `inferInstance`. It has type `{α : Sort u} → [i : α] → α`, and is useful for triggering the type class resolution procedure when the expected type is an instance.

```
#check (inferInstance : Inhabited Nat) -- Inhabited Nat

def foo : Inhabited (Nat × Nat) :=
  inferInstance

theorem ex : foo.default = (default, default) :=
  rfl
```

You can use the command `#print` to inspect how simple `inferInstance` is.

```
#print inferInstance
```

ToString

The polymorphic method `toString` has type `{α : Type u} → [ToString α] → α → String`. You implement the instance for your own types and use chaining to convert complex values into strings. Lean comes with `ToString` instances for most builtin types.

```
structure Person where
  name : String
  age  : Nat

instance : ToString Person where
  toString p := p.name ++ "@" ++ toString p.age

#eval toString { name := "Leo", age := 542 : Person }
#eval toString ({ name := "Daniel", age := 18 : Person }, "hello")
```

Numerals

Numerals are polymorphic in Lean. You can use a numeral (e.g., `2`) to denote an element of any type that implements the type class `OfNat`.

```
structure Rational where
  num : Int
  den : Nat
  inv : den ≠ 0

instance : OfNat Rational n where
  ofNat := { num := n, den := 1, inv := by decide }

instance : ToString Rational where
  toString r := s!"{r.num}/{r.den}"

#eval (2 : Rational) -- 2/1

#check (2 : Rational) -- Rational
#check (2 : Nat)      -- Nat
```

Lean elaborates the terms `(2 : Nat)` and `(2 : Rational)` as `OfNat.ofNat Nat 2 (instOfNatNat 2)` and `OfNat.ofNat Rational 2 (instOfNatRational 2)` respectively. We say the numerals `2` occurring in the elaborated terms are *raw* natural numbers. You can input the raw natural number `2` using the macro `nat_lit 2`.

```
#check nat_lit 2  -- Nat
```

Raw natural numbers are *not* polymorphic.

The `OfNat` instance is parametric on the numeral. So, you can define instances for particular numerals. The second argument is often a variable as in the example above, or a *raw* natural number.

```
class Monoid (α : Type u) where
  unit : α
  op   : α → α → α

instance [s : Monoid α] : OfNat α (nat_lit 1) where
  ofNat := s.unit

def getUnit [Monoid α] : α :=
  1
```

Output Parameters

By default, Lean only tries to synthesize an instance `Inhabited T` when the term `T` is known and does not contain missing parts. The following command produces the error

"typeclass instance problem is stuck, it is often due to metavariables `?m.7`" because the type has a missing part (i.e., the `_`).

```
#check_failure (inferInstance : Inhabited (Nat × _))
```

You can view the parameter of the type class `Inhabited` as an *input* value for the type class synthesizer. When a type class has multiple parameters, you can mark some of them as output parameters. Lean will start type class synthesizer even when these parameters have missing parts. In the following example, we use output parameters to define a *heterogeneous* polymorphic multiplication.

```
class HMul (α : Type u) (β : Type v) (γ : outParam (Type w)) where
  hMul : α → β → γ

export HMul (hMul)

instance : HMul Nat Nat Nat where
  hMul := Nat.mul

instance : HMul Nat (Array Nat) (Array Nat) where
  hMul a bs := bs.map (fun b => hMul a b)

#eval hMul 4 3           -- 12
#eval hMul 4 #[2, 3, 4]  -- #[8, 12, 16]
```

The parameters `α` and `β` are considered input parameters and `γ` an output one. Given an application `hMul a b`, after the types of `a` and `b` are known, the type class synthesizer is invoked, and the resulting type is obtained from the output parameter `γ`. In the example above, we defined two instances. The first one is the homogeneous multiplication for natural numbers. The second is the scalar multiplication for arrays. Note that you chain instances and generalize the second instance.

```
class HMul (α : Type u) (β : Type v) (γ : outParam (Type w)) where
  hMul : α → β → γ

export HMul (hMul)

instance : HMul Nat Nat Nat where
  hMul := Nat.mul

instance : HMul Int Int Int where
  hMul := Int.mul

instance [HMul α β γ] : HMul α (Array β) (Array γ) where
  hMul a bs := bs.map (fun b => hMul a b)

#eval hMul 4 3                    -- 12
#eval hMul 4 #[2, 3, 4]           -- #[8, 12, 16]
#eval hMul (-2) #[3, -1, 4]       -- #[-6, 2, -8]
#eval hMul 2 #[#[2, 3], #[0, 4]]  -- #[#[4, 6], #[0, 8]]
```

You can use our new scalar array multiplication instance on arrays of type `Array β` with a scalar of type `α` whenever you have an instance `HMul α β γ`. In the last `#eval`, note that the instance was used twice on an array of arrays.

Default Instances

In the class `HMul`, the parameters `α` and `β` are treated as input values. Thus, type class synthesis only starts after these two types are known. This may often be too restrictive.

```
class HMul (α : Type u) (β : Type v) (γ : outParam (Type w)) where
  hMul : α → β → γ

export HMul (hMul)

instance : HMul Int Int Int where
  hMul := Int.mul

def xs : List Int := [1, 2, 3]

-- Error "typeclass instance problem is stuck, it is often due to metavariables
HMul ?m.89 ?m.90 ?m.91"
#check_failure fun y => xs.map (fun x => hMul x y)
```

The instance `HMul` is not synthesized by Lean because the type of `y` has not been provided. However, it is natural to assume that the type of `y` and `x` should be the same in this kind of situation. We can achieve exactly that using *default instances*.

```
class HMul (α : Type u) (β : Type v) (γ : outParam (Type w)) where
  hMul : α → β → γ

export HMul (hMul)

@[default_instance]
instance : HMul Int Int Int where
  hMul := Int.mul

def xs : List Int := [1, 2, 3]

#check fun y => xs.map (fun x => hMul x y)  -- Int → List Int
```

By tagging the instance above with the attribute `default_instance`, we are instructing Lean to use this instance on pending type class synthesis problems. The actual Lean implementation defines homogeneous and heterogeneous classes for arithmetical operators. Moreover, `a+b`, `a*b`, `a-b`, `a/b`, and `a%b` are notations for the heterogeneous versions. The instance `OfNat Nat n` is the default instance (with priority 100) for the `OfNat` class. This is why the numeral `2` has type `Nat` when the expected type is not known. You can define default instances with higher priority to override the builtin ones.

```
structure Rational where
  num : Int
  den : Nat
  inv : den ≠ 0

@[default_instance 200]
instance : OfNat Rational n where
  ofNat := { num := n, den := 1, inv := by decide }

instance : ToString Rational where
  toString r := s!"{r.num}/{r.den}"

#check 2 -- Rational
```

Priorities are also useful to control the interaction between different default instances. For example, suppose `xs` has type `List α`. When elaborating `xs.map (fun x => 2 * x)`, we want the homogeneous instance for multiplication to have higher priority than the default instance for `OfNat`. This is particularly important when we have implemented only the instance `HMul α α α`, and did not implement `HMul Nat α α`. Now, we reveal how the notation `a*b` is defined in Lean.

```
class OfNat (α : Type u) (n : Nat) where
  ofNat : α

@[default_instance]
instance (n : Nat) : OfNat Nat n where
  ofNat := n

class HMul (α : Type u) (β : Type v) (γ : outParam (Type w)) where
  hMul : α → β → γ

class Mul (α : Type u) where
  mul : α → α → α

@[default_instance 10]
instance [Mul α] : HMul α α α where
  hMul a b := Mul.mul a b

infixl:70 " * " => HMul.hMul
```

The `Mul` class is convenient for types that only implement the homogeneous multiplication.

Local Instances

Type classes are implemented using attributes in Lean. Thus, you can use the `local` modifier to indicate that they only have effect until the current `section` or `namespace` is closed, or until the end of the current file.

```
structure Point where
  x : Nat
  y : Nat

section

local instance : Add Point where
  add a b := { x := a.x + b.x, y := a.y + b.y }

def double (p : Point) :=
  p + p

end -- instance `Add Point` is not active anymore

-- def triple (p : Point) :=
--  p + p + p  -- Error: failed to synthesize instance
```

You can also temporarily disable an instance using the `attribute` command until the current `section` or `namespace` is closed, or until the end of the current file.

```
structure Point where
  x : Nat
  y : Nat

instance addPoint : Add Point where
  add a b := { x := a.x + b.x, y := a.y + b.y }

def double (p : Point) :=
  p + p

attribute [-instance] addPoint

-- def triple (p : Point) :=
--  p + p + p  -- Error: failed to synthesize instance
```

We recommend you only use this command to diagnose problems.

Scoped Instances

You can also declare scoped instances in namespaces. This kind of instance is only active when you are inside of the namespace or open the namespace.

```
structure Point where
  x : Nat
  y : Nat

namespace Point

scoped instance : Add Point where
  add a b := { x := a.x + b.x, y := a.y + b.y }

def double (p : Point) :=
  p + p

end Point
-- instance `Add Point` is not active anymore

-- #check fun (p : Point) => p + p + p  -- Error

namespace Point
-- instance `Add Point` is active again
#check fun (p : Point) => p + p + p

end Point

open Point -- activates instance `Add Point`
#check fun (p : Point) => p + p + p
```

You can use the command `open scoped <namespace>` to activate scoped attributes but will not "open" the names from the namespace.

```
structure Point where
  x : Nat
  y : Nat

namespace Point

scoped instance : Add Point where
  add a b := { x := a.x + b.x, y := a.y + b.y }

def double (p : Point) :=
  p + p

end Point

open scoped Point -- activates instance `Add Point`
#check fun (p : Point) => p + p + p

-- #check fun (p : Point) => double p -- Error: unknown identifier 'double'
```

Decidable Propositions

Let us consider another example of a type class defined in the standard library, namely the type class of `Decidable` propositions. Roughly speaking, an element of `Prop` is said to be

decidable if we can decide whether it is true or false. The distinction is only useful in constructive mathematics; classically, every proposition is decidable. But if we use the classical principle, say, to define a function by cases, that function will not be computable. Algorithmically speaking, the `Decidable` type class can be used to infer a procedure that effectively determines whether or not the proposition is true. As a result, the type class supports such computational definitions when they are possible while at the same time allowing a smooth transition to the use of classical definitions and classical reasoning.

In the standard library, `Decidable` is defined formally as follows:

```
class inductive Decidable (p : Prop) where
  | isFalse (h : ¬p) : Decidable p
  | isTrue  (h : p)  : Decidable p
```

Logically speaking, having an element `t : Decidable p` is stronger than having an element `t : p ∨ ¬p`; it enables us to define values of an arbitrary type depending on the truth value of `p`. For example, for the expression `if p then a else b` to make sense, we need to know that `p` is decidable. That expression is syntactic sugar for `ite p a b`, where `ite` is defined as follows:

```
def ite {α : Sort u} (c : Prop) [h : Decidable c] (t e : α) : α :=
  Decidable.casesOn (motive := fun _ => α) h (fun _ => e) (fun _ => t)
```

The standard library also contains a variant of `ite` called `dite`, the dependent if-then-else expression. It is defined as follows:

```
def dite {α : Sort u} (c : Prop) [h : Decidable c] (t : c → α) (e : Not c → α) :
α :=
  Decidable.casesOn (motive := fun _ => α) h e t
```

That is, in `dite c t e`, we can assume `hc : c` in the "then" branch, and `hnc : ¬ c` in the "else" branch. To make `dite` more convenient to use, Lean allows us to write `if h : c then t else e` instead of `dite c (λ h : c => t) (λ h : ¬ c => e)`.

Without classical logic, we cannot prove that every proposition is decidable. But we can prove that *certain* propositions are decidable. For example, we can prove the decidability of basic operations like equality and comparisons on the natural numbers and the integers. Moreover, decidability is preserved under propositional connectives:

```
#check @instDecidableAnd
  -- {p q : Prop} → [Decidable p] → [Decidable q] → Decidable (And p q)

#check @instDecidableOr
#check @instDecidableNot
```

Thus we can carry out definitions by cases on decidable predicates on the natural numbers:

```
def step (a b x : Nat) : Nat :=
  if x < a ∨ x > b then 0 else 1

set_option pp.explicit true
#print step
```

Turning on implicit arguments shows that the elaborator has inferred the decidability of the proposition `x < a ∨ x > b`, simply by applying appropriate instances.

With the classical axioms, we can prove that every proposition is decidable. You can import the classical axioms and make the generic instance of decidability available by opening the `Classical` namespace.

```
open Classical
```

Thereafter `Decidable p` has an instance for every `p`. Thus all theorems in the library that rely on decidability assumptions are freely available when you want to reason classically. In Chapter Axioms and Computation, we will see that using the law of the excluded middle to define functions can prevent them from being used computationally. Thus, the standard library assigns a low priority to the `propDecidable` instance.

```
open Classical
noncomputable scoped
instance (priority := low) propDecidable (a : Prop) : Decidable a :=
  choice <| match em a with
    | Or.inl h => ⟨isTrue h⟩
    | Or.inr h => ⟨isFalse h⟩
```

This guarantees that Lean will favor other instances and fall back on `propDecidable` only after other attempts to infer decidability have failed.

The `Decidable` type class also provides a bit of small-scale automation for proving theorems. The standard library introduces the tactic `decide` that uses the `Decidable` instance to solve simple goals.

```
example : 10 < 5 ∨ 1 > 0 := by
  decide

example : ¬ (True ∧ False) := by
  decide

example : 10 * 20 = 200 := by
  decide

theorem ex : True ∧ 2 = 1+1 := by
  decide

#print ex
-- theorem ex : True ∧ 2 = 1 + 1 :=
-- of_decide_eq_true (Eq.refl true)

#check @of_decide_eq_true
-- ∀ {p : Prop} [Decidable p], decide p = true → p

#check @decide
-- (p : Prop) → [Decidable p] → Bool
```

They work as follows. The expression `decide p` tries to infer a decision procedure for `p`, and, if it is successful, evaluates to either `true` or `false`. In particular, if `p` is a true closed expression, `decide p` will reduce definitionally to the Boolean `true`. On the assumption that `decide p = true` holds, `of_decide_eq_true` produces a proof of `p`. The tactic `decide` puts it all together to prove a target `p`. By the previous observations, `decide` will succeed any time the inferred decision procedure for `c` has enough information to evaluate, definitionally, to the `isTrue` case.

Managing Type Class Inference

If you are ever in a situation where you need to supply an expression that Lean can infer by type class inference, you can ask Lean to carry out the inference using `inferInstance`:

```
def foo : Add Nat := inferInstance
def bar : Inhabited (Nat → Nat) := inferInstance

#check @inferInstance
-- {α : Sort u} → [α] → α
```

In fact, you can use Lean's `(t : T)` notation to specify the class whose instance you are looking for, in a concise manner:

```
#check (inferInstance : Add Nat)
```

You can also use the auxiliary definition `inferInstanceAs`:

```
#check inferInstanceAs (Add Nat)

#check @inferInstanceAs
-- (α : Sort u) → [α] → α
```

Sometimes Lean can't find an instance because the class is buried under a definition. For example, Lean cannot find an instance of `Inhabited (Set α)`. We can declare one explicitly:

```
def Set (α : Type u) := α → Prop

-- fails
-- example : Inhabited (Set α) :=
--  inferInstance

instance : Inhabited (Set α) :=
  inferInstanceAs (Inhabited (α → Prop))
```

At times, you may find that the type class inference fails to find an expected instance, or, worse, falls into an infinite loop and times out. To help debug in these situations, Lean enables you to request a trace of the search:

```
set_option trace.Meta.synthInstance true
```

If you are using VS Code, you can read the results by hovering over the relevant theorem or definition, or opening the messages window with `Ctrl-Shift-Enter`. In Emacs, you can use `C-c C-x` to run an independent Lean process on your file, and the output buffer will show a trace every time the type class resolution procedure is subsequently triggered.

You can also limit the search using the following options:

```
set_option synthInstance.maxHeartbeats 10000
set_option synthInstance.maxSize 400
```

Option `synthInstance.maxHeartbeats` specifies the maximum amount of heartbeats per typeclass resolution problem. A heartbeat is the number of (small) memory allocations (in thousands), 0 means there is no limit. Option `synthInstance.maxSize` is the maximum number of instances used to construct a solution in the type class instance synthesis procedure.

Remember also that in both the VS Code and Emacs editor modes, tab completion works in `set_option`, to help you find suitable options.

As noted above, the type class instances in a given context represent a Prolog-like program, which gives rise to a backtracking search. Both the efficiency of the program and the solutions that are found can depend on the order in which the system tries the instance. Instances which are declared last are tried first. Moreover, if instances are declared in other modules, the order in which they are tried depends on the order in which namespaces are opened. Instances declared in namespaces which are opened later are tried earlier.

You can change the order that type class instances are tried by assigning them a *priority*. When an instance is declared, it is assigned a default priority value. You can assign other priorities when defining an instance. The following example illustrates how this is done:

```
class Foo where
  a : Nat
  b : Nat

instance (priority := default+1) i1 : Foo where
  a := 1
  b := 1

instance i2 : Foo where
  a := 2
  b := 2

example : Foo.a = 1 :=
  rfl

instance (priority := default+2) i3 : Foo where
  a := 3
  b := 3

example : Foo.a = 3 :=
  rfl
```

Coercions using Type Classes

The most basic type of coercion maps elements of one type to another. For example, a coercion from `Nat` to `Int` allows us to view any element `n : Nat` as an element of `Int`. But some coercions depend on parameters; for example, for any type `α`, we can view any element `as : List α` as an element of `Set α`, namely, the set of elements occurring in the list. The corresponding coercion is defined on the "family" of types `List α`, parameterized by `α`.

Lean allows us to declare three kinds of coercions:

- from a family of types to another family of types
- from a family of types to the class of sorts
- from a family of types to the class of function types

The first kind of coercion allows us to view any element of a member of the source family as an element of a corresponding member of the target family. The second kind of coercion allows us to view any element of a member of the source family as a type. The third kind of coercion allows us to view any element of the source family as a function. Let us consider each of these in turn.

In Lean, coercions are implemented on top of the type class resolution framework. We define a coercion from `α` to `β` by declaring an instance of `Coe α β`. For example, we can define a coercion from `Bool` to `Prop` as follows:

```
instance : Coe Bool Prop where
  coe b := b = true
```

This enables us to use boolean terms in if-then-else expressions:

```
#eval if true then 5 else 3
#eval if false then 5 else 3
```

We can define a coercion from `List α` to `Set α` as follows:

```
def List.toSet : List α → Set α
  | []    => Set.empty
  | a::as => {a} ∪ as.toSet

instance : Coe (List α) (Set α) where
  coe a := a.toSet

def s : Set Nat := {1}
#check s ∪ [2, 3]
-- s ∪ List.toSet [2, 3] : Set Nat
```

We can use the notation `↑` to force a coercion to be introduced in a particular place. It is also helpful to make our intent clear, and work around limitations of the coercion resolution system.

```
def s : Set Nat := {1}

#check let x := ↑[2, 3]; s ∪ x
-- let x := List.toSet [2, 3]; s ∪ x : Set Nat
#check let x := [2, 3]; s ∪ x
-- let x := [2, 3]; s ∪ List.toSet x : Set Nat
```

Lean also supports dependent coercions using the type class `CoeDep`. For example, we cannot coerce arbitrary propositions to `Bool`, only the ones that implement the `Decidable` typeclass.

```
instance (p : Prop) [Decidable p] : CoeDep Prop p Bool where
  coe := decide p
```

Lean will also chain (non-dependent) coercions as necessary. Actually, the type class `CoeT` is the transitive closure of `Coe`.

Let us now consider the second kind of coercion. By the *class of sorts*, we mean the collection of universes `Type u`. A coercion of the second kind is of the form:

```
    c : (x1 : A1) → ... → (xn : An) → F x1 ... xn → Type u
```

where `F` is a family of types as above. This allows us to write `s : t` whenever `t` is of type `F a1 ... an`. In other words, the coercion allows us to view the elements of `F a1 ... an` as types. This is very useful when defining algebraic structures in which one component, the carrier of the structure, is a `Type`. For example, we can define a semigroup as follows:

```
structure Semigroup where
  carrier : Type u
  mul : carrier → carrier → carrier
  mul_assoc (a b c : carrier) : mul (mul a b) c = mul a (mul b c)

instance (S : Semigroup) : Mul S.carrier where
  mul a b := S.mul a b
```

In other words, a semigroup consists of a type, `carrier`, and a multiplication, `mul`, with the property that the multiplication is associative. The `instance` command allows us to write `a * b` instead of `Semigroup.mul S a b` whenever we have `a b : S.carrier`; notice that Lean can infer the argument `S` from the types of `a` and `b`. The function `Semigroup.carrier` maps the class `Semigroup` to the sort `Type u`:

```
#check Semigroup.carrier
```

If we declare this function to be a coercion, then whenever we have a semigroup `S : Semigroup`, we can write `a : S` instead of `a : S.carrier`:

```
instance : CoeSort Semigroup (Type u) where
  coe s := s.carrier

example (S : Semigroup) (a b c : S) : (a * b) * c = a * (b * c) :=
  Semigroup.mul_assoc _ a b c
```

It is the coercion that makes it possible to write `(a b c : S)`. Note that, we define an instance of `CoeSort Semigroup (Type u)` instead of `Coe Semigroup (Type u)`.

By the *class of function types*, we mean the collection of Pi types `(z : B) → C`. The third kind of coercion has the form:

```
c : (x1 : A1) → ... → (xn : An) → (y : F x1 ... xn) → (z : B) → C
```

where `F` is again a family of types and `B` and `C` can depend on `x1, ..., xn, y`. This makes it possible to write `t s` whenever `t` is an element of `F a1 ... an`. In other words, the coercion enables us to view elements of `F a1 ... an` as functions. Continuing the example above, we can define the notion of a morphism between semigroups `S1` and `S2`. That is, a function from the carrier of `S1` to the carrier of `S2` (note the implicit coercion) that respects the multiplication. The projection `morphism.mor` takes a morphism to the underlying function:

```
structure Morphism (S1 S2 : Semigroup) where
  mor : S1 → S2
  resp_mul : ∀ a b : S1, mor (a * b) = (mor a) * (mor b)

#check @Morphism.mor
```

As a result, it is a prime candidate for the third type of coercion.

```
instance (S1 S2 : Semigroup) : CoeFun (Morphism S1 S2) (fun _ => S1 → S2) where
  coe m := m.mor

theorem resp_mul {S1 S2 : Semigroup} (f : Morphism S1 S2) (a b : S1)
        : f (a * b) = f a * f b :=
  f.resp_mul a b

example (S1 S2 : Semigroup) (f : Morphism S1 S2) (a : S1) :
      f (a * a * a) = f a * f a * f a :=
  calc f (a * a * a)
    _ = f (a * a) * f a := by rw [resp_mul f]
    _ = f a * f a * f a := by rw [resp_mul f]
```

With the coercion in place, we can write `f (a * a * a)` instead of `f.mor (a * a * a)`. When the `Morphism`, `f`, is used where a function is expected, Lean inserts the coercion. Similar to `CoeSort`, we have yet another class `CoeFun` for this class of coercions. The field `F` is used to specify the function type we are coercing to. This type may depend on the type we are coercing from.

The Conversion Tactic Mode

Inside a tactic block, one can use the keyword `conv` to enter *conversion mode*. This mode allows to travel inside assumptions and goals, even inside function abstractions and dependent arrows, to apply rewriting or simplifying steps.

Basic navigation and rewriting

As a first example, let us prove example `(a b c : Nat) : a * (b * c) = a * (c * b)` (examples in this file are somewhat artificial since other tactics could finish them immediately). The naive first attempt is to enter tactic mode and try `rw [Nat.mul_comm]`. But this transforms the goal into `b * c * a = a * (c * b)`, after commuting the very first multiplication appearing in the term. There are several ways to fix this issue, and one way is to use a more precise tool: the conversion mode. The following code block shows the current target after each line.

```
example (a b c : Nat) : a * (b * c) = a * (c * b) := by
  conv =>
    -- ⊢ a * (b * c) = a * (c * b)
    lhs
    -- ⊢ a * (b * c)
    congr
    -- 2 goals: ⊢ a, ⊢ b * c
    rfl
    -- ⊢ b * c
    rw [Nat.mul_comm]
```

The above snippet shows three navigation commands:

- `lhs` navigates to the left-hand side of a relation (equality, in this case). There is also a `rhs` to navigate to the right-hand side.
- `congr` creates as many targets as there are (nondependent and explicit) arguments to the current head function (here the head function is multiplication).
- `rfl` closes target using reflexivity.

Once arrived at the relevant target, we can use `rw` as in normal tactic mode.

The second main reason to use conversion mode is to rewrite under binders. Suppose we want to prove example `(fun x : Nat => 0 + x) = (fun x => x)`. The naive first attempt is to enter tactic mode and try `rw [Nat.zero_add]`. But this fails with a frustrating

```
error: tactic 'rewrite' failed, did not find instance of the pattern
       in the target expression
  0 + ?n
⊢ (fun x => 0 + x) = fun x => x
```

The solution is:

```
example : (fun x : Nat => 0 + x) = (fun x => x) := by
  conv =>
    lhs
    intro x
    rw [Nat.zero_add]
```

where `intro x` is the navigation command entering inside the `fun` binder. Note that this example is somewhat artificial, one could also do:

```
example : (fun x : Nat => 0 + x) = (fun x => x) := by
  funext x; rw [Nat.zero_add]
```

or just

```
example : (fun x : Nat => 0 + x) = (fun x => x) := by
  simp
```

`conv` can also rewrite a hypothesis `h` from the local context, using `conv at h`.

Pattern matching

Navigation using the above commands can be tedious. One can shortcut it using pattern matching as follows:

```
example (a b c : Nat) : a * (b * c) = a * (c * b) := by
  conv in b * c => rw [Nat.mul_comm]
```

which is just syntax sugar for

```
example (a b c : Nat) : a * (b * c) = a * (c * b) := by
  conv =>
    pattern b * c
    rw [Nat.mul_comm]
```

Of course, wildcards are allowed:

```
example (a b c : Nat) : a * (b * c) = a * (c * b) := by
  conv in _ * c => rw [Nat.mul_comm]
```

Structuring conversion tactics

Curly brackets and `.` can also be used in `conv` mode to structure tactics:

```
example (a b c : Nat) : (0 + a) * (b * c) = a * (c * b) := by
  conv =>
    lhs
    congr
    . rw [Nat.zero_add]
    . rw [Nat.mul_comm]
```

Other tactics inside conversion mode

- `arg i` enter the `i`-th nondependent explicit argument of an application.

```
example (a b c : Nat) : a * (b * c) = a * (c * b) := by
  conv =>
    -- ⊢ a * (b * c) = a * (c * b)
    lhs
    -- ⊢ a * (b * c)
    arg 2
    -- ⊢ b * c
    rw [Nat.mul_comm]
```

- `args` alternative name for `congr`.

- `simp` applies the simplifier to the current goal. It supports the same options available in regular tactic mode.

```
def f (x : Nat) :=
  if x > 0 then x + 1 else x + 2

example (g : Nat → Nat) (h₁ : g x = x + 1) (h₂ : x > 0) : g x = f x := by
  conv =>
    rhs
    simp [f, h₂]
  exact h₁
```

- `enter [1, x, 2, y]` iterate `arg` and `intro` with the given arguments. It is just the macro:

```
syntax enterArg := ident <|> group("@"? num)
syntax "enter " "[" (colGt enterArg),+ "]": conv
macro_rules
  | `(conv| enter [$i:num]) => `(conv| arg $i)
  | `(conv| enter [@$i:num]) => `(conv| arg @$i)
  | `(conv| enter [$id:ident]) => `(conv| ext $id)
  | `(conv| enter [$arg:enterArg, $args,*]) => `(conv| (enter [$arg]; enter
[$args,*]))
```

- `done` fail if there are unsolved goals.
- `trace_state` display the current tactic state.
- `whnf` put term in weak head normal form.
- `tactic => <tactic sequence>` go back to regular tactic mode. This is useful for discharging goals not supported by `conv` mode, and applying custom congruence and extensionality lemmas.

```
example (g : Nat → Nat → Nat)
        (h₁ : ∀ x, x ≠ 0 → g x x = 1)
        (h₂ : x ≠ 0)
        : g x x + x = 1 + x := by
  conv =>
    lhs
    -- ⊢ g x x + x
    arg 1
    -- ⊢ g x x
    rw [h₁]
    -- 2 goals: ⊢ 1, ⊢ x ≠ 0
    . skip
    . tactic => exact h₂
```

- `apply <term>` is syntax sugar for `tactic => apply <term>`.

```
example (g : Nat → Nat → Nat)
        (h₁ : ∀ x, x ≠ 0 → g x x = 1)
        (h₂ : x ≠ 0)
        : g x x + x = 1 + x := by
  conv =>
    lhs
    arg 1
    rw [h₁]
    . skip
    . apply h₂
```

Axioms and Computation

We have seen that the version of the Calculus of Constructions that has been implemented in Lean includes dependent function types, inductive types, and a hierarchy of universes that starts with an impredicative, proof-irrelevant `Prop` at the bottom. In this chapter, we consider ways of extending the CIC with additional axioms and rules. Extending a foundational system in such a way is often convenient; it can make it possible to prove more theorems, as well as make it easier to prove theorems that could have been proved otherwise. But there can be negative consequences of adding additional axioms, consequences which may go beyond concerns about their correctness. In particular, the use of axioms bears on the computational content of definitions and theorems, in ways we will explore here.

Lean is designed to support both computational and classical reasoning. Users that are so inclined can stick to a "computationally pure" fragment, which guarantees that closed expressions in the system evaluate to canonical normal forms. In particular, any closed computationally pure expression of type `Nat`, for example, will reduce to a numeral.

Lean's standard library defines an additional axiom, propositional extensionality, and a quotient construction which in turn implies the principle of function extensionality. These extensions are used, for example, to develop theories of sets and finite sets. We will see below that using these theorems can block evaluation in Lean's kernel, so that closed terms of type `Nat` no longer evaluate to numerals. But Lean erases types and propositional information when compiling definitions to bytecode for its virtual machine evaluator, and since these axioms only add new propositions, they are compatible with that computational interpretation. Even computationally inclined users may wish to use the classical law of the excluded middle to reason about computation. This also blocks evaluation in the kernel, but it is compatible with compilation to bytecode.

The standard library also defines a choice principle that is entirely antithetical to a computational interpretation, since it magically produces "data" from a proposition asserting its existence. Its use is essential to some classical constructions, and users can import it when needed. But expressions that use this construction to produce data do not have computational content, and in Lean we are required to mark such definitions as `noncomputable` to flag that fact.

Using a clever trick (known as Diaconescu's theorem), one can use propositional extensionality, function extensionality, and choice to derive the law of the excluded middle. As noted above, however, use of the law of the excluded middle is still compatible with bytecode compilation and code extraction, as are other classical principles, as long as they are not used to manufacture data.

To summarize, then, on top of the underlying framework of universes, dependent function types, and inductive types, the standard library adds three additional components:

- the axiom of propositional extensionality
- a quotient construction, which implies function extensionality
- a choice principle, which produces data from an existential proposition.

The first two of these block normalization within Lean, but are compatible with bytecode evaluation, whereas the third is not amenable to computational interpretation. We will spell out the details more precisely below.

Historical and Philosophical Context

For most of its history, mathematics was essentially computational: geometry dealt with constructions of geometric objects, algebra was concerned with algorithmic solutions to systems of equations, and analysis provided means to compute the future behavior of systems evolving over time. From the proof of a theorem to the effect that "for every `x`, there is a `y` such that ...", it was generally straightforward to extract an algorithm to compute such a `y` given `x`.

In the nineteenth century, however, increases in the complexity of mathematical arguments pushed mathematicians to develop new styles of reasoning that suppress algorithmic information and invoke descriptions of mathematical objects that abstract away the details of how those objects are represented. The goal was to obtain a powerful "conceptual" understanding without getting bogged down in computational details, but this had the effect of admitting mathematical theorems that are simply *false* on a direct computational reading.

There is still fairly uniform agreement today that computation is important to mathematics. But there are different views as to how best to address computational concerns. From a *constructive* point of view, it is a mistake to separate mathematics from its computational roots; every meaningful mathematical theorem should have a direct computational interpretation. From a *classical* point of view, it is more fruitful to maintain a separation of concerns: we can use one language and body of methods to write computer programs, while maintaining the freedom to use nonconstructive theories and methods to reason about them. Lean is designed to support both of these approaches. Core parts of the library are developed constructively, but the system also provides support for carrying out classical mathematical reasoning.

Computationally, the purest part of dependent type theory avoids the use of `Prop` entirely. Inductive types and dependent function types can be viewed as data types, and terms of these types can be "evaluated" by applying reduction rules until no more rules can be applied. In principle, any closed term (that is, term with no free variables) of type `Nat` should evaluate to a numeral, `succ (... (succ zero)...)`.

Introducing a proof-irrelevant `Prop` and marking theorems irreducible represents a first step towards separation of concerns. The intention is that elements of a type `p : Prop` should play no role in computation, and so the particular construction of a term `t : p` is "irrelevant" in that sense. One can still define computational objects that incorporate elements of type `Prop`; the point is that these elements can help us reason about the effects of the computation, but can be ignored when we extract "code" from the term. Elements of type `Prop` are not entirely innocuous, however. They include equations `s = t : α` for any type `α`, and such equations can be used as casts, to type check terms. Below, we will see examples of how such casts can block computation in the system. However, computation is still possible under an evaluation scheme that erases propositional content, ignores intermediate typing constraints, and reduces terms until they reach a normal form. This is precisely what Lean's virtual machine does.

Having adopted a proof-irrelevant `Prop`, one might consider it legitimate to use, for example, the law of the excluded middle, `p ∨ ¬p`, where `p` is any proposition. Of course, this, too, can block computation according to the rules of CIC, but it does not block bytecode evaluation, as described above. It is only the choice principles discussed in :numref: `choice` that completely erase the distinction between the proof-irrelevant and data-relevant parts of the theory.

Propositional Extensionality

Propositional extensionality is the following axiom:

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

It asserts that when two propositions imply one another, they are actually equal. This is consistent with set-theoretic interpretations in which any element `a : Prop` is either empty or the singleton set `{*}`, for some distinguished element `*`. The axiom has the effect that equivalent propositions can be substituted for one another in any context:

```
theorem thm₁ (a b c d e : Prop) (h : a ↔ b) : (c ∧ a ∧ d → e) ↔ (c ∧ b ∧ d → e)
:=
  propext h ▸ Iff.refl _

theorem thm₂ (a b : Prop) (p : Prop → Prop) (h : a ↔ b) (h₁ : p a) : p b :=
  propext h ▸ h₁
```

Function Extensionality

Similar to propositional extensionality, function extensionality asserts that any two functions of type `(x : α) → β x` that agree on all their inputs are equal:

```
universe u v
#check (@funext :
           {α : Type u}
         → {β : α → Type u}
         → {f g : (x : α) → β x}
         → (∀ (x : α), f x = g x)
         → f = g)

#print funext
```

From a classical, set-theoretic perspective, this is exactly what it means for two functions to be equal. This is known as an "extensional" view of functions. From a constructive perspective, however, it is sometimes more natural to think of functions as algorithms, or computer programs, that are presented in some explicit way. It is certainly the case that two computer programs can compute the same answer for every input despite the fact that they are syntactically quite different. In much the same way, you might want to maintain a view of functions that does not force you to identify two functions that have the same input / output behavior. This is known as an "intensional" view of functions.

In fact, function extensionality follows from the existence of quotients, which we describe in the next section. In the Lean standard library, therefore, `funext` is thus proved from the quotient construction.

Suppose that for `α : Type` we define the `Set α := α → Prop` to denote the type of subsets of `α`, essentially identifying subsets with predicates. By combining `funext` and `propext`, we obtain an extensional theory of such sets:

```
def Set (α : Type u) := α → Prop

namespace Set

def mem (x : α) (a : Set α) := a x

infix:50 (priority := high) "∈" => mem

theorem setext {a b : Set α} (h : ∀ x, x ∈ a ↔ x ∈ b) : a = b :=
  funext (fun x => propext (h x))

end Set
```

We can then proceed to define the empty set and set intersection, for example, and prove set identities:

```
def empty : Set α := fun x => False

notation (priority := high) "∅" => empty

def inter (a b : Set α) : Set α :=
  fun x => x ∈ a ∧ x ∈ b

infix:70 " ∩ " => inter

theorem inter_self (a : Set α) : a ∩ a = a :=
  setext fun x => Iff.intro
    (fun ⟨h, _⟩ => h)
    (fun h => ⟨h, h⟩)

theorem inter_empty (a : Set α) : a ∩ ∅ = ∅ :=
  setext fun x => Iff.intro
    (fun ⟨_, h⟩ => h)
    (fun h => False.elim h)

theorem empty_inter (a : Set α) : ∅ ∩ a = ∅ :=
  setext fun x => Iff.intro
    (fun ⟨h, _⟩ => h)
    (fun h => False.elim h)

theorem inter.comm (a b : Set α) : a ∩ b = b ∩ a :=
  setext fun x => Iff.intro
    (fun ⟨h₁, h₂⟩ => ⟨h₂, h₁⟩)
    (fun ⟨h₁, h₂⟩ => ⟨h₂, h₁⟩)
```

The following is an example of how function extensionality blocks computation inside the Lean kernel:

```
def f (x : Nat) := x
def g (x : Nat) := 0 + x

theorem f_eq_g : f = g :=
  funext fun x => (Nat.zero_add x).symm

def val : Nat :=
  Eq.recOn (motive := fun _ _ => Nat) f_eq_g 0

-- does not reduce to 0
#reduce val

-- evaluates to 0
#eval val
```

First, we show that the two functions `f` and `g` are equal using function extensionality, and then we cast `0` of type `Nat` by replacing `f` by `g` in the type. Of course, the cast is vacuous, because `Nat` does not depend on `f`. But that is enough to do the damage: under the computational rules of the system, we now have a closed term of `Nat` that does not reduce to a numeral. In this case, we may be tempted to reduce the expression to `0`. But in nontrivial examples, eliminating cast changes the type of the term, which might make an ambient expression type incorrect. The virtual machine, however, has no trouble evaluating

the expression to `0`. Here is a similarly contrived example that shows how `propext` can get in the way:

```
theorem tteq : (True ∧ True) = True :=
  propext (Iff.intro (fun ⟨h, _⟩ => h) (fun h => ⟨h, h⟩))

def val : Nat :=
  Eq.recOn (motive := fun _ _ => Nat) tteq 0

-- does not reduce to 0
#reduce val

-- evaluates to 0
#eval val
```

Current research programs, including work on *observational type theory* and *cubical type theory*, aim to extend type theory in ways that permit reductions for casts involving function extensionality, quotients, and more. But the solutions are not so clear-cut, and the rules of Lean's underlying calculus do not sanction such reductions.

In a sense, however, a cast does not change the meaning of an expression. Rather, it is a mechanism to reason about the expression's type. Given an appropriate semantics, it then makes sense to reduce terms in ways that preserve their meaning, ignoring the intermediate bookkeeping needed to make the reductions type-correct. In that case, adding new axioms in `Prop` does not matter; by proof irrelevance, an expression in `Prop` carries no information, and can be safely ignored by the reduction procedures.

Quotients

Let `α` be any type, and let `r` be an equivalence relation on `α`. It is mathematically common to form the "quotient" `α / r`, that is, the type of elements of `α` "modulo" `r`. Set theoretically, one can view `α / r` as the set of equivalence classes of `α` modulo `r`. If `f : α → β` is any function that respects the equivalence relation in the sense that for every `x y : α`, `r x y` implies `f x = f y`, then `f` "lifts" to a function `f' : α / r → β` defined on each equivalence class `⟦x⟧` by `f' ⟦x⟧ = f x`. Lean's standard library extends the Calculus of Constructions with additional constants that perform exactly these constructions, and installs this last equation as a definitional reduction rule.

In its most basic form, the quotient construction does not even require `r` to be an equivalence relation. The following constants are built into Lean:

```
universe u v

axiom Quot : {α : Sort u} → (α → α → Prop) → Sort u

axiom Quot.mk : {α : Sort u} → (r : α → α → Prop) → α → Quot r

axiom Quot.ind :
    ∀ {α : Sort u} {r : α → α → Prop} {β : Quot r → Prop},
      (∀ a, β (Quot.mk r a)) → (q : Quot r) → β q

axiom Quot.lift :
    {α : Sort u} → {r : α → α → Prop} → {β : Sort u} → (f : α → β)
    → (∀ a b, r a b → f a = f b) → Quot r → β
```

The first one forms a type `Quot r` given a type `α` by any binary relation `r` on `α`. The second maps `α` to `Quot α`, so that if `r : α → α → Prop` and `a : α`, then `Quot.mk r a` is an element of `Quot r`. The third principle, `Quot.ind`, says that every element of `Quot.mk r a` is of this form. As for `Quot.lift`, given a function `f : α → β`, if `h` is a proof that `f` respects the relation `r`, then `Quot.lift f h` is the corresponding function on `Quot r`. The idea is that for each element `a` in `α`, the function `Quot.lift f h` maps `Quot.mk r a` (the `r`-class containing `a`) to `f a`, wherein `h` shows that this function is well defined. In fact, the computation principle is declared as a reduction rule, as the proof below makes clear.

```
def mod7Rel (x y : Nat) : Prop :=
  x % 7 = y % 7

-- the quotient type
#check (Quot mod7Rel : Type)

-- the class of a
#check (Quot.mk mod7Rel 4 : Quot mod7Rel)

def f (x : Nat) : Bool :=
  x % 7 = 0

theorem f_respects (a b : Nat) (h : mod7Rel a b) : f a = f b := by
  simp [mod7Rel, f] at *
  rw [h]

#check (Quot.lift f f_respects : Quot mod7Rel → Bool)

-- the computation principle
example (a : Nat) : Quot.lift f f_respects (Quot.mk mod7Rel a) = f a :=
  rfl
```

The four constants, `Quot`, `Quot.mk`, `Quot.ind`, and `Quot.lift` in and of themselves are not very strong. You can check that the `Quot.ind` is satisfied if we take `Quot r` to be simply `α`, and take `Quot.lift` to be the identity function (ignoring `h`). For that reason, these four constants are not viewed as additional axioms.

They are, like inductively defined types and the associated constructors and recursors, viewed as part of the logical framework.

What makes the `Quot` construction into a bona fide quotient is the following additional axiom:

```
axiom Quot.sound :
      ∀ {α : Type u} {r : α → α → Prop} {a b : α},
        r a b → Quot.mk r a = Quot.mk r b
```

This is the axiom that asserts that any two elements of `α` that are related by `r` become identified in the quotient. If a theorem or definition makes use of `Quot.sound`, it will show up in the `#print axioms` command.

Of course, the quotient construction is most commonly used in situations when `r` is an equivalence relation. Given `r` as above, if we define `r'` according to the rule `r' a b` iff `Quot.mk r a = Quot.mk r b`, then it's clear that `r'` is an equivalence relation. Indeed, `r'` is the *kernel* of the function `a ↦ quot.mk r a`. The axiom `Quot.sound` says that `r a b` implies `r' a b`. Using `Quot.lift` and `Quot.ind`, we can show that `r'` is the smallest equivalence relation containing `r`, in the sense that if `r''` is any equivalence relation containing `r`, then `r' a b` implies `r'' a b`. In particular, if `r` was an equivalence relation to start with, then for all `a` and `b` we have `r a b` iff `r' a b`.

To support this common use case, the standard library defines the notion of a *setoid*, which is simply a type with an associated equivalence relation:

```
class Setoid (α : Sort u) where
  r : α → α → Prop
  iseqv : Equivalence r

instance {α : Sort u} [Setoid α] : HasEquiv α :=
  ⟨Setoid.r⟩

namespace Setoid

variable {α : Sort u} [Setoid α]

theorem refl (a : α) : a ≈ a :=
  iseqv.refl a

theorem symm {a b : α} (hab : a ≈ b) : b ≈ a :=
  iseqv.symm hab

theorem trans {a b c : α} (hab : a ≈ b) (hbc : b ≈ c) : a ≈ c :=
  iseqv.trans hab hbc

end Setoid
```

Given a type `α`, a relation `r` on `α`, and a proof `p` that `r` is an equivalence relation, we can define `Setoid.mk r p` as an instance of the setoid class.

```
def Quotient {α : Sort u} (s : Setoid α) :=
  @Quot α Setoid.r
```

The constants `Quotient.mk`, `Quotient.ind`, `Quotient.lift`, and `Quotient.sound` are nothing more than the specializations of the corresponding elements of `Quot`. The fact that type class inference can find the setoid associated to a type `α` brings a number of benefits. First, we can use the notation `a ≈ b` (entered with `\approx`) for `Setoid.r a b`, where the instance of `Setoid` is implicit in the notation `Setoid.r`. We can use the generic theorems `Setoid.refl`, `Setoid.symm`, `Setoid.trans` to reason about the relation. Specifically with quotients we can use the generic notation `⟦a⟧` for `Quot.mk Setoid.r` where the instance of `Setoid` is implicit in the notation `Setoid.r`, as well as the theorem `Quotient.exact`:

```
#check (@Quotient.exact :
         ∀ {α : Sort u} {s : Setoid α} {a b : α},
           Quotient.mk s a = Quotient.mk s b → a ≈ b)
```

Together with `Quotient.sound`, this implies that the elements of the quotient correspond exactly to the equivalence classes of elements in `α`.

Recall that in the standard library, `α × β` represents the Cartesian product of the types `α` and `β`. To illustrate the use of quotients, let us define the type of *unordered* pairs of elements of a type `α` as a quotient of the type `α × α`. First, we define the relevant equivalence relation:

```
private def eqv (p₁ p₂ : α × α) : Prop :=
  (p₁.1 = p₂.1 ∧ p₁.2 = p₂.2) ∨ (p₁.1 = p₂.2 ∧ p₁.2 = p₂.1)

infix:50 " ~ " => eqv
```

The next step is to prove that `eqv` is in fact an equivalence relation, which is to say, it is reflexive, symmetric and transitive. We can prove these three facts in a convenient and readable way by using dependent pattern matching to perform case-analysis and break the hypotheses into pieces that are then reassembled to produce the conclusion.

```
private theorem eqv.refl (p : α × α) : p ~ p :=
  Or.inl ⟨rfl, rfl⟩

private theorem eqv.symm : ∀ {p₁ p₂ : α × α}, p₁ ~ p₂ → p₂ ~ p₁
  | (a₁, a₂), (b₁, b₂), (Or.inl ⟨a₁b₁, a₂b₂⟩) =>
    Or.inl (by simp_all)
  | (a₁, a₂), (b₁, b₂), (Or.inr ⟨a₁b₂, a₂b₁⟩) =>
    Or.inr (by simp_all)

private theorem eqv.trans : ∀ {p₁ p₂ p₃ : α × α}, p₁ ~ p₂ → p₂ ~ p₃ → p₁ ~ p₃
  | (a₁, a₂), (b₁, b₂), (c₁, c₂), Or.inl ⟨a₁b₁, a₂b₂⟩, Or.inl ⟨b₁c₁, b₂c₂⟩ =>
    Or.inl (by simp_all)
  | (a₁, a₂), (b₁, b₂), (c₁, c₂), Or.inl ⟨a₁b₁, a₂b₂⟩, Or.inr ⟨b₁c₂, b₂c₁⟩ =>
    Or.inr (by simp_all)
  | (a₁, a₂), (b₁, b₂), (c₁, c₂), Or.inr ⟨a₁b₂, a₂b₁⟩, Or.inl ⟨b₁c₁, b₂c₂⟩ =>
    Or.inr (by simp_all)
  | (a₁, a₂), (b₁, b₂), (c₁, c₂), Or.inr ⟨a₁b₂, a₂b₁⟩, Or.inr ⟨b₁c₂, b₂c₁⟩ =>
    Or.inl (by simp_all)

private theorem is_equivalence : Equivalence (@eqv α) :=
  { refl := eqv.refl, symm := eqv.symm, trans := eqv.trans }
```

Now that we have proved that `eqv` is an equivalence relation, we can construct a `Setoid (α × α)`, and use it to define the type `UProd α` of unordered pairs.

```
instance uprodSetoid (α : Type u) : Setoid (α × α) where
  r     := eqv
  iseqv := is_equivalence

def UProd (α : Type u) : Type u :=
  Quotient (uprodSetoid α)

namespace UProd

def mk {α : Type} (a₁ a₂ : α) : UProd α :=
  Quotient.mk' (a₁, a₂)

notation "{ " a₁ ", " a₂ " }" => mk a₁ a₂

end UProd
```

Notice that we locally define the notation `{a₁, a₂}` for unordered pairs as `Quotient.mk (a₁, a₂)`. This is useful for illustrative purposes, but it is not a good idea in general, since the notation will shadow other uses of curly brackets, such as for records and sets.

We can easily prove that `{a₁, a₂} = {a₂, a₁}` using `Quot.sound`, since we have `(a₁, a₂) ~ (a₂, a₁)`.

```
theorem mk_eq_mk (a₁ a₂ : α) : {a₁, a₂} = {a₂, a₁} :=
  Quot.sound (Or.inr ⟨rfl, rfl⟩)
```

To complete the example, given `a : α` and `u : uprod α`, we define the proposition `a ∈ u` which should hold if `a` is one of the elements of the unordered pair `u`. First, we define a

similar proposition `mem_fn a u` on (ordered) pairs; then we show that `mem_fn` respects the equivalence relation `eqv` with the lemma `mem_respects`. This is an idiom that is used extensively in the Lean standard library.

```
private def mem_fn (a : α) : α × α → Prop
  | (a₁, a₂) => a = a₁ ∨ a = a₂

-- auxiliary lemma for proving mem_respects
private theorem mem_swap {a : α} :
      ∀ {p : α × α}, mem_fn a p = mem_fn a (⟨p.2, p.1⟩)
  | (a₁, a₂) => by
    apply propext
    apply Iff.intro
    . intro
      | Or.inl h => exact Or.inr h
      | Or.inr h => exact Or.inl h
    . intro
      | Or.inl h => exact Or.inr h
      | Or.inr h => exact Or.inl h


private theorem mem_respects
      : {p₁ p₂ : α × α} → (a : α) → p₁ ~ p₂ → mem_fn a p₁ = mem_fn a p₂
  | (a₁, a₂), (b₁, b₂), a, Or.inl ⟨a₁b₁, a₂b₂⟩ => by simp_all
  | (a₁, a₂), (b₁, b₂), a, Or.inr ⟨a₁b₂, a₂b₁⟩ => by simp_all; apply mem_swap

def mem (a : α) (u : UProd α) : Prop :=
  Quot.liftOn u (fun p => mem_fn a p) (fun p₁ p₂ e => mem_respects a e)

infix:50 (priority := high) " ∈ " => mem

theorem mem_mk_left (a b : α) : a ∈ {a, b} :=
  Or.inl rfl

theorem mem_mk_right (a b : α) : b ∈ {a, b} :=
  Or.inr rfl

theorem mem_or_mem_of_mem_mk {a b c : α} : c ∈ {a, b} → c = a ∨ c = b :=
  fun h => h
```

For convenience, the standard library also defines `Quotient.lift₂` for lifting binary functions, and `Quotient.ind₂` for induction on two variables.

We close this section with some hints as to why the quotient construction implies function extensionality. It is not hard to show that extensional equality on the `(x : α) → β x` is an equivalence relation, and so we can consider the type `extfun α β` of functions "up to equivalence." Of course, application respects that equivalence in the sense that if `f₁` is equivalent to `f₂`, then `f₁ a` is equal to `f₂ a`. Thus application gives rise to a function `extfun_app : extfun α β → (x : α) → β x`. But for every `f`, `extfun_app ⟦f⟧` is definitionally equal to `fun x => f x`, which is in turn definitionally equal to `f`. So, when `f₁` and `f₂` are extensionally equal, we have the following chain of equalities:

```
    f₁ = extfun_app ⟦f₁⟧ = extfun_app ⟦f₂⟧ = f₂
```

As a result, `f₁` is equal to `f₂`.

Choice

To state the final axiom defined in the standard library, we need the `Nonempty` type, which is defined as follows:

```
class inductive Nonempty (α : Sort u) : Prop where
  | intro (val : α) : Nonempty α
```

Because `Nonempty α` has type `Prop` and its constructor contains data, it can only eliminate to `Prop`. In fact, `Nonempty α` is equivalent to `∃ x : α, True`:

```
example (α : Type u) : Nonempty α ↔ ∃ x : α, True :=
  Iff.intro (fun ⟨a⟩ => ⟨a, trivial⟩) (fun ⟨a, h⟩ => ⟨a⟩)
```

Our axiom of choice is now expressed simply as follows:

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

Given only the assertion `h` that `α` is nonempty, `choice h` magically produces an element of `α`. Of course, this blocks any meaningful computation: by the interpretation of `Prop`, `h` contains no information at all as to how to find such an element.

This is found in the `Classical` namespace, so the full name of the theorem is `Classical.choice`. The choice principle is equivalent to the principle of *indefinite description*, which can be expressed with subtypes as follows:

```
noncomputable def indefiniteDescription {α : Sort u} (p : α → Prop)
                                        (h : ∃ x, p x) : {x // p x} :=
  choice <| let ⟨x, px⟩ := h; ⟨⟨x, px⟩⟩
```

Because it depends on `choice`, Lean cannot generate bytecode for `indefiniteDescription`, and so requires us to mark the definition as `noncomputable`. Also in the `Classical` namespace, the function `choose` and the property `choose_spec` decompose the two parts of the output of `indefiniteDescription`:

```
noncomputable def choose {α : Sort u} {p : α → Prop} (h : ∃ x, p x) : α :=
  (indefiniteDescription p h).val

theorem choose_spec {α : Sort u} {p : α → Prop} (h : ∃ x, p x) : p (choose h) :=
  (indefiniteDescription p h).property
```

The `choice` principle also erases the distinction between the property of being `Nonempty` and the more constructive property of being `Inhabited`:

```
theorem inhabited_of_nonempty : Nonempty α → Inhabited α :=
  fun h => choice (let ⟨a⟩ := h; ⟨⟨a⟩⟩)
```

In the next section, we will see that `propext`, `funext`, and `choice`, taken together, imply the law of the excluded middle and the decidability of all propositions. Using those, one can strengthen the principle of indefinite description as follows:

```
#check (@strongIndefiniteDescription :
         {α : Sort u} → (p : α → Prop)
         → Nonempty α → {x // (∃ (y : α), p y) → p x})
```

Assuming the ambient type `α` is nonempty, `strongIndefiniteDescription p` produces an element of `α` satisfying `p` if there is one. The data component of this definition is conventionally known as *Hilbert's epsilon function*:

```
#check (@epsilon :
         {α : Sort u} → [Nonempty α]
         → (α → Prop) → α)

#check (@epsilon_spec :
         ∀ {α : Sort u} {p : α → Prop} (hex : ∃ (y : α), p y),
           p (@epsilon _ (nonempty_of_exists hex) p))
```

The Law of the Excluded Middle

The law of the excluded middle is the following

```
open Classical

#check (@em : ∀ (p : Prop), p ∨ ¬p)
```

Diaconescu's theorem states that the axiom of choice is sufficient to derive the law of excluded middle. More precisely, it shows that the law of the excluded middle follows from `Classical.choice`, `propext`, and `funext`. We sketch the proof that is found in the standard library.

First, we import the necessary axioms, and define two predicates `U` and `V`:

```
open Classical
theorem em (p : Prop) : p ∨ ¬p :=
  let U (x : Prop) : Prop := x = True ∨ p
  let V (x : Prop) : Prop := x = False ∨ p

  have exU : ∃ x, U x := ⟨True, Or.inl rfl⟩
  have exV : ∃ x, V x := ⟨False, Or.inl rfl⟩
```

If `p` is true, then every element of `Prop` is in both `U` and `V`. If `p` is false, then `U` is the singleton `true`, and `V` is the singleton `false`.

Next, we use `some` to choose an element from each of `U` and `V`:

```
  let u : Prop := choose exU
  let v : Prop := choose exV

  have u_def : U u := choose_spec exU
  have v_def : V v := choose_spec exV
```

Each of `U` and `V` is a disjunction, so `u_def` and `v_def` represent four cases. In one of these cases, `u = True` and `v = False`, and in all the other cases, `p` is true. Thus we have:

```
  have not_uv_or_p : u ≠ v ∨ p :=
    match u_def, v_def with
    | Or.inr h, _ => Or.inr h
    | _, Or.inr h => Or.inr h
    | Or.inl hut, Or.inl hvf =>
      have hne : u ≠ v := by simp [hvf, hut, true_ne_false]
      Or.inl hne
```

On the other hand, if `p` is true, then, by function extensionality and propositional extensionality, `U` and `V` are equal. By the definition of `u` and `v`, this implies that they are equal as well.

```
  have p_implies_uv : p → u = v :=
    fun hp =>
    have hpred : U = V :=
      funext fun x =>
        have hl : (x = True ∨ p) → (x = False ∨ p) :=
          fun _ => Or.inr hp
        have hr : (x = False ∨ p) → (x = True ∨ p) :=
          fun _ => Or.inr hp
        show (x = True ∨ p) = (x = False ∨ p) from
          propext (Iff.intro hl hr)
    have h₀ : ∀ exU exV, @choose _ U exU = @choose _ V exV := by
      rw [hpred]; intros; rfl
    show u = v from h₀ _ _
```

Putting these last two facts together yields the desired conclusion:

```
  match not_uv_or_p with
  | Or.inl hne => Or.inr (mt p_implies_uv hne)
  | Or.inr h   => Or.inl h
```

Consequences of excluded middle include double-negation elimination, proof by cases, and proof by contradiction, all of which are described in the Section Classical Logic. The law of the excluded middle and propositional extensionality imply propositional completeness:

```
open Classical
theorem propComplete (a : Prop) : a = True ∨ a = False :=
  match em a with
  | Or.inl ha => Or.inl (propext (Iff.intro (fun _ => ⟨⟩) (fun _ => ha)))
  | Or.inr hn => Or.inr (propext (Iff.intro (fun h => hn h) (fun h => False.elim
h)))
```

Together with choice, we also get the stronger principle that every proposition is decidable. Recall that the class of `Decidable` propositions is defined as follows:

```
class inductive Decidable (p : Prop) where
  | isFalse (h : ¬p) : Decidable p
  | isTrue  (h : p)  : Decidable p
```

In contrast to `p ∨ ¬ p`, which can only eliminate to `Prop`, the type `Decidable p` is equivalent to the sum type `Sum p (¬ p)`, which can eliminate to any type. It is this data that is needed to write an if-then-else expression.

As an example of classical reasoning, we use `choose` to show that if `f : α → β` is injective and `α` is inhabited, then `f` has a left inverse. To define the left inverse `linv`, we use a dependent if-then-else expression. Recall that `if h : c then t else e` is notation for `dite c (fun h : c => t) (fun h : ¬ c => e)`. In the definition of `linv`, choice is used twice: first, to show that `(∃ a : A, f a = b)` is "decidable," and then to choose an `a` such that `f a = b`. Notice that `propDecidable` is a scoped instance and is activated by the `open Classical` command. We use this instance to justify the if-then-else expression. (See also the discussion in Section Decidable Propositions).

```
open Classical

noncomputable def linv [Inhabited α] (f : α → β) : β → α :=
  fun b : β => if ex : (∃ a : α, f a = b) then choose ex else default

theorem linv_comp_self {f : α → β} [Inhabited α]
                       (inj : ∀ {a b}, f a = f b → a = b)
                       : linv f ∘ f = id :=
  funext fun a =>
    have ex  : ∃ a₁ : α, f a₁ = f a := ⟨a, rfl⟩
    have feq : f (choose ex) = f a  := choose_spec ex
    calc linv f (f a)
      _ = choose ex := dif_pos ex
      _ = a         := inj feq
```

From a classical point of view, `linv` is a function. From a constructive point of view, it is unacceptable; because there is no way to implement such a function in general, the construction is not informative.