Theorem Proving in Lean

Jeremy Avigad
Leonardo de Moura
Soonho Kong

Version 71877da, updated at 2016-10-12 22:05:01 -0400

Contents

1

Introduction

1.1 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 us 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.

1.2 About Lean

The *Lean* project was launched by Leonardo de Moura at Microsoft Research Redmond in 2012. 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.

There are currently two ways to use Lean. The first is to run it from the web: a Javascript version of Lean, a standard library of definitions and theorems, and an editor are actually downloaded to your browser and run there. This provides a quick and convenient way to begin experimenting with the system.

The second way to use Lean is to install and run it natively on your computer. The native version is much faster than the web version, and is more flexible in other ways, too. It comes with an Emacs mode that offers powerful support for writing and debugging proofs, and is much better suited for serious use.

1.3 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, we will explain 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 variant of a system known as the *Calculus of Inductive Constructions*[1, 3], or *CIC*. We will explain not only how to define mathematical objects and express mathematical assertions in dependent type theory, but also how to use it 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. We will describe various methods to support this in dependent type theory. For example, we will discuss term rewriting, and Lean's automated methods for simplifying terms and expressions automatically. Similarly, we will discuss methods of *elaboration* and *type inference*, which can be used to support flexible forms of algebraic reasoning.

Finally, of course, we will discuss features that are specific to Lean, including the language with which you can communicate with the system, and the mechanisms Lean offers for managing complex theories and data.

If you are reading this book within Lean's online tutorial system, you will see a copy of the Lean editor at right, with an output buffer beneath it. At any point, you can type things into the editor, press the "play" button, and see Lean's response. Notice that you can resize the various windows if you would like.

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

```
theorem and_commutative (p q : Prop) : p ∧ q → q ∧ p :=
assume hpq : p ∧ q,
have hp : p, from and.elim_left Hpq,
have hq : q, from and.elim_right Hpq,
show q ∧ p, from and.intro Hq Hp
```

Once again, if you are reading the book online, you will see a button that reads "try it yourself." Pressing the button copies the example into the Lean editor with enough surrounding context to make the example compile correctly, and then runs Lean. We recommend running the examples and experimenting with the code on your own as you work through the chapters that follow.

1.4 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, Nathan Carter, Amine Chaieb, Floris van Doorn, Anthony Hart, Sean Leather, Christopher John Mazey, Sebastian Ullrich, Daniel Velleman, and Théo Zimmerman for their contributions, and we apologize to those whose names we have inadvertently omitted.

2

Dependent Type Theory

Dependent type theory is a powerful and expressive language, allowing us 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 Inductive 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.

2.1 Simple Type Theory

As a foundation for mathematics, set theory has a simple ontology that is rather appealing. Everything is a set, including numbers, functions, triangles, stochastic processes, and Riemannian manifolds. It is a remarkable fact that one can construct a rich mathematical universe from a small number of axioms that describe a few basic set-theoretic constructions.

But for many purposes, including formal theorem proving, it is better to have an infrastructure that helps us manage and keep track of the various kinds of mathematical objects we are working with. "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.

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

```
/- declare some constants -/

constant m : nat        -- m is a natural number
constant n : nat
```

```
constants b1 b2 : bool  -- declare two constants at once

/- 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 boolean and
check b1 || b2     -- boolean or
check tt           -- boolean "true"

-- Try some examples of your own.
```

The `/-` and `-/` annotations indicate that the next line is 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 `constant` and `constants` commands introduce new constant symbols into the working environment, and the `check` command asks Lean to report their types. You should test this, and try typing some examples of your own. Declaring new objects in this way is a good way to experiment with the system, but it is ultimately undesirable: Lean is a foundational system, which is to say, it provides us with powerful mechanisms to *define* all the mathematical objects we need, rather than simply postulating them to the system. We will explore these mechanisms in the chapters to come.

What makes simple type theory powerful is that one 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 cartesian product, that is, the type of ordered pairs consisting of an element of `A` paired with an element of `B`.

```
constants m n : nat

constant f : nat → nat           -- type the arrow as "\to" or "\r"
constant f' : nat -> nat         -- alternative ASCII notation
constant f'' : ℕ → ℕ             -- \nat is alternative notation for nat
constant p : nat × nat           -- type the product as "\times"
constant q : prod nat nat        -- alternative notation
constant g : nat → nat → nat
constant g' : nat → (nat → nat)  -- has the same type as g!
constant h : nat × nat → nat

constant F : (nat → nat) → nat   -- a "functional"

check f              -- ℕ → ℕ
check f n            -- ℕ
check g m n          -- ℕ
check g m            -- ℕ → ℕ
check (m, n)         -- ℕ × ℕ
check p.1            -- ℕ
```

```
check p.2             -- ℕ
check (m, n).1        -- ℕ
check (p.1, n)        -- ℕ × ℕ
check F f             -- ℕ

-- Try it on your own: write down some types, declare some constants,
-- and check some expressions.
```

Let us dispense with some basic syntax. You can enter the unicode arrow → by typing `\to` or "`\r`. You can also use the ASCII alternative `->`, so that the expression `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 symbol ℕ is alternative unicode notation for `nat`; you can enter it by typing `\nat`. The unicode symbol × for the cartesian product is entered `\prod`.

There are a few more things to notice here. First, the application of a function `f` to a value `x` is denoted `f x`. Second, when writing type expressions, arrows associate to the *right*; for example, the type of `g` is `nat → (nat → nat)`. Thus we can view `g` as a function that takes natural numbers and returns another function that takes a natural number and returns a natural number. In type theory, this is generally more convenient than writing `g` as a function that takes a pair of natural numbers as input, and returns a natural number as output. For example, it allows us to "partially apply" the function `g`. The example above shows that `g m` has type `nat → nat`, that is, the function that "waits" for a second argument, `n`, and then returns `g m n`. Taking a function `h` of type `nat × nat → nat` and "redefining" it to look like `g` is a process known as *currying*, something we will come back to below.

By now you may also have guessed that, in Lean, `(m, n)` denotes the ordered pair of `m` and `n`, and if `p` is a pair, `fst p` and `snd p` denote the two projections.

2.2 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 of study. 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
```

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

```
constants A B : Type
constant F : Type → Type
constant G : Type → Type → Type

check A         -- Type
check F A       -- Type
check F nat     -- Type
check G A       -- Type → Type
check G A B     -- Type
check G A nat   -- Type
```

Indeed, we have already seen an example of a function of type `Type → Type → Type`, namely, the Cartesian product.

```
constants A B : Type

check prod A B       -- Type
check prod nat nat   -- Type
```

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

```
constant A : Type

check list A    -- Type
check list nat  -- Type
```

For those more comfortable with set-theoretic foundations, it may be helpful to think of a type as nothing more than a set, in which case, the elements of the type are just the elements of the set. Given that every expression in Lean has a type, it is natural to ask: what type does `Type` itself have?

```
check Type      -- Type₂
```

We 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 1   -- Type₂
check Type 2   -- Type₃
check Type 3   -- Type 4
check Type 4   -- Type 5
```

Think of `Type 1` as a universe of "small" or "ordinary" types. `Type 2` is then a larger universe of types, which contains `Type 1` as an element, and `Type 3` is an even larger universe of types, which contains `Type 2` as an element. The list is indefinite, so that there is a `Type n` for every natural number `n`. Lean introduces abbreviations for the first three levels:

```
check Type    -- same as Type 1
check Type₂   -- same as Type 2
check Type₃   -- same as Type 3
```

It is rare to have to use more than those. There is also a `Type 0`, which is also denoted `Prop`. This type has special properties, and will be discussed in the next chapter.

We want some operations, however, to be *polymorphic* over type universes. For example, `list A` should make sense for any type `A`, no matter which type universe `A` lives in. This explains the type annotation of the function `list`:

```
check list    -- Type u_1 → Type (max 1 u_1)
```

Here `u_1` is a variable ranging over type levels. The output of the `check` command means that whenever `A` has type `Type n`, `list A` also has type `Type n` if `n` is at least 1, and has `Type 1` if `A` has type `0`. The function `prod` is similarly polymorphic:

```
check prod    -- Type u_1 → Type u_2 → Type (max 1 u_1 u_2)
```

To define polymorphic constants and variables, Lean allows us to declare universe variables explicitly:

```
universe variable u
constant A : Type u
check A
```

Throughout this book, you will see us do this in examples when we want type constructions to have as much generality as possible. We will see that the ability to treat type constructors as instances of ordinary mathematical functions is a powerful feature of dependent type theory.

2.3 Function Abstraction and Evaluation

We have seen that if we have `m n : nat`, then we have `(m, n) : nat × nat`. This gives us a way of creating pairs of natural numbers. Conversely, if we have `p : nat × nat`, then we have `fst p : nat` and `snd p : nat`. This gives us a way of "using" a pair, by extracting its two components.

We already know how to "use" a function `f : A → B`, namely, we can apply it to an element `a : A` to obtain `f a : B`. But how do we create a function from another expression?

The companion to application is a process known as "abstraction," or "lambda abstraction." Suppose that by temporarily postulating a variable `x : A` we can construct an expression `t : B`. Then the expression `fun x : A, t`, or, equivalently, `λ x : A, t`, is an object of type `A → B`. Think of this as the function from `A` to `B` which maps any value `x` to the value `t`, which depends on `x`. For example, in mathematics it is common to say "let `f` be the function which maps any natural number `x` to `x + 5`." The expression `λ x : nat, x + 5` is just a symbolic representation of the right-hand side of this assignment.

```
check fun x : nat, x + 5
check λ x : nat, x + 5
```

Here are some more abstract examples:

```
constants A B  : Type
constants a1 a2 : A
constants b1 b2 : B

constant f : A → A
constant g : A → B
constant h : A → B → A
constant p : A → A → bool

check fun x : A, f x                      -- A → A
check λ x : A, f x                        -- A → A
check λ x : A, f (f x)                    -- A → A
check λ x : A, h x b1                     -- A → A
check λ y : B, h a1 y                     -- B → A
check λ x : A, p (f (f x)) (h (f a1) b2)  -- A → bool
check λ x : A, λ y : B, h (f x) y         -- A → B → A
check λ (x : A) (y : B), h (f x) y        -- A → B → A
check λ x y, h (f x) y                    -- A → B → A
```

Lean interprets the final three examples as the same expression; in the last expression, Lean infers the type of `x` and `y` from the types of `f` and `h`.

Be sure to try writing some expressions of your own. Some mathematically common examples of operations of functions can be described in terms of lambda abstraction:

```
constants A B C : Type
constant f : A → B
constant g : B → C
constant b : B

check λ x : A, x        -- A → A
check λ x : A, b        -- A → B
check λ x : A, g (f x)  -- A → C
check λ x, g (f x)
```

```
-- we can abstract any of the constants in the previous definitions

check λ b : B, λ x : A, x      -- B → A → A
check λ (b : B) (x : A), x     -- equivalent to the previous line
check λ (g : B → C) (f : A → B) (x : A), g (f x)
                               -- (B → C) → (A → B) → A → C
-- we can even abstract over the type

check λ (A B : Type) (b : B) (x : A), x
check λ (A B C : Type) (g : B → C) (f : A → B) (x : A), g (f x)
```

Think about what these expressions mean. The expression `λ x : A, x` denotes the identity function on `A`, the expression `λ x : A, b` denotes the constant function that always returns `b`, and `λ x : A, g (f x)`, denotes the composition of `f` and `g`. We can, in general, leave off the type annotations on the variable and let Lean infer it for us. So, for example, we can write `λ x, g (f x)` instead of `λ x : A, g (f x)`.

We can abstract over any of the constants in the previous definitions:

```
check λ b : B, λ x : A, x      -- B → A → A
check λ (b : B) (x : A), x     -- B → A → A
check λ (g : B → C) (f : A → B) (x : A), g (f x)
                               -- (B → C) → (A → B) → A → C
```

Lean lets us combine lambdas, so the second example is equivalent to the first. We can even abstract over the type:

```
check λ (A B : Type) (b : B) (x : A), x
check λ (A B C : Type) (g : B → C) (f : A → B) (x : A), g (f x)
```

The last expression, for example, denotes the function that takes three types, `A`, `B`, and `C`, and two functions, `g : B → C` and `f : A → B`, and returns the composition of `g` and `f`. (Making sense of the type of this function requires an understanding of dependent products, which we will explain below.) Within a lambda expression `λ x : A, t`, the variable `x` is a "bound variable": it is really a placeholder, whose "scope" does not extend beyond `t`. For example, the variable `b` in the expression `λ (b : B) (x : A), x` has nothing to do with the constant `b` declared earlier. In fact, the expression denotes the same function as `λ (u : B) (z : A), z`. Formally, the 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 : A → B` to a term `s : A` yields an expression `t s : B`. Returning to the previous example and renaming bound variables for clarity, notice the types of the following expressions:

```
constants A B C : Type
constant f : A → B
```

```
constant g : B → C
constant h : A → A
constants (a : A) (b : B)

check (λ x : A, x) a                -- A
check (λ x : A, b) a                -- B
check (λ x : A, b) (h a)            -- B
check (λ x : A, g (f x)) (h (h a))  -- C

check (λ (v : B → C) (u : A → B) x, v (u x)) g f a   -- C

check (λ (Q R S : Type) (v : R → S) (u : Q → R) (x : Q),
       v (u x)) A B C g f a         -- C
```

As expected, the expression `(λ x : A, x) a` has type `A`. In fact, more should be true: applying the expression `(λ x : A, x)` to `a` should "return" the value `a`. And, indeed, it does:

```
constants A B C : Type
constant f : A → B
constant g : B → C
constant h : A → A
constants (a : A) (b : B)

eval (λ x : A, x) a                 -- a
eval (λ x : A, b) a                 -- b
eval (λ x : A, b) (h a)             -- b
eval (λ x : A, g (f x)) a           -- g (f a)

eval (λ (v : B → C) (u : A → B) x, v (u x)) g f a   -- g (f a)

eval (λ (Q R S : Type) (v : R → S) (u : Q → R) (x : Q),
      v (u x)) A B C g f a          -- g (f a)
```

The command `eval` tells Lean to *evaluate* an expression. The process of simplifying an expression `(λ x, t)s` to `t[s/x]` – that is, `t` with `s` substituted for the variable `x` – is known as *beta reduction*, and two terms that beta reduce to a common term are called *beta equivalent*. But the `eval` command carries out other forms of reduction as well:

```
constants m n : nat
constant b : bool

print "reducing pairs"
eval (m, n).1         -- m
eval (m, n).2         -- n

print "reducing boolean expressions"
eval tt && ff         -- ff
eval b && ff          -- ff

print "reducing arithmetic expressions"
eval n + 0            -- n
```

```
eval n + 2          -- succ (succ n)
eval 2 + 3          -- 5
```

In a later chapter, we will explain how these terms are evaluated. For now, we only wish to emphasize that this is an important feature of dependent type theory: every term has a computational behavior, and supports a notion of reduction, or *normalization*. In principle, two terms that reduce to the same value are called *definitionally equal*. They are considered "the same" by the underlying logical framework, and Lean does its best to recognize and support these identifications.

2.4 Introducing Definitions

As we have noted above, declaring constants in the Lean environment is a good way to postulate new objects to experiment with, but most of the time what we really want to do is *define* objects in Lean and prove things about them. The `definition` command provides one important way of defining new objects.

```
definition foo : (ℕ → ℕ) → ℕ := λ f, f 0

check foo    -- ℕ
print foo    -- λ (f : ℕ → ℕ), f 0
```

We can omit the type when Lean has enough information to infer it:

```
definition foo' := λ f : ℕ → ℕ, f 0
```

The general form of a definition is `definition foo : T := bar`. Lean can usually infer the type `T`, 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 the right type.

Because function definitions are so common, Lean provides an alternative notation, which puts the abstracted variables before the colon and omits the lambda:

```
definition double (x : ℕ) : ℕ := x + x
print double
check double 3
eval double 3    -- 6

definition square (x : ℕ) := x * x
print square
check square 3
eval square 3    -- 9

definition do_twice (f : ℕ → ℕ) (x : ℕ) : ℕ := f (f x)

eval do_twice double 2    -- 8
```

These definitions are equivalent to the following:

```
definition double : ℕ → ℕ := λ x, x + x
definition square : ℕ → ℕ := λ x, x * x
definition do_twice : (ℕ → ℕ) → ℕ → ℕ := λ f x, f (f x)
```

We can even use this approach to specify arguments that are types:

```
definition compose (A B C : Type) (g : B → C) (f : A → B) (x : A) :
  C :=
g (f x)
```

As an exercise, we encourage you to use `do_twice` and `double` to define functions that quadruple their input, and multiply the input by 8. As a further exercise, we encourage you to try defining a function `Do_Twice : ((ℕ → ℕ) → (ℕ → ℕ)) → (ℕ → ℕ) → (ℕ → ℕ)` which iterates *its* argument twice, so that `Do_Twice do_twice` a function which iterates *its* input four times, and evaluate `Do_Twice do_twice double 2`.

Above, we discussed the process of "currying" a function, that is, taking a function `f (a, b)` that takes an ordered pair as an argument, and recasting it as a function `f' a b` that takes two arguments successively. As another exercise, we encourage you to complete the following definitions, which "curry" and "uncurry" a function.

```
definition curry (A B C : Type) (f : A × B → C) : A → B → C := sorry

definition uncurry (A B C : Type) (f : A → B → C) : A × B → C := sorry
```

2.5 Local Definitions

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

```
check let y := 2 + 2 in y * y   -- ℕ
eval  let y := 2 + 2 in y * y   -- 16

definition t (x : ℕ) : ℕ :=
let y := x + x in y * y

eval t 2   -- 16
```

Here, `t` is definitionally equal to the term `(x + x) * (x + x)`. You can combine multiple assignments in a single `let` statement:

```
check let y := 2 + 2, z := y + y in z * z   -- 16
eval  let y := 2 + 2, z := y + y in z * z   -- 64
```

Notice that the meaning of the expression `let a := t1 in t2` is very similar to the meaning of `(λ 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 `λ 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 in t2` that cannot be expressed as `(λ a, t2) t1`. As an exercise, try to understand why the definition of `foo` below type checks, but the definition of `bar` does not.

```
definition foo := let a := nat  in λ x : a, x + 2

/-
definition bar := (λ a, λ x : a, x + 2) nat
-/
```

2.6 Variables and Sections

This is a good place to introduce some organizational features of Lean that are not a part of the axiomatic framework *per se*, but make it possible to work in the framework more efficiently.

We have seen that the `constant` command allows us to declare new objects, which then become part of the global context. Declaring new objects in this way is somewhat crass. Lean enables us to *define* all of the mathematical objects we need, and *declaring* new objects willy-nilly is therefore somewhat lazy. In the words of Bertand Russell, it has all the advantages of theft over honest toil. We will see in the next chapter that it is also somewhat dangerous: declaring a new constant is tantamount to declaring an axiomatic extension of our foundational system, and may result in inconsistency.

So far, in this tutorial, we have used the `constant` command to create "arbitrary" objects to work with in our examples. For example, we have declared types `A`, `B`, and `C` to populate our context. This can be avoided, using implicit or explicit lambda abstraction in our definitions to declare such objects "locally":

```
definition compose (A B C : Type) (g : B → C) (f : A → B) (x : A) :
  C := g (f x)

definition do_twice (A : Type) (h : A → A) (x : A) : A := h (h x)

definition do_thrice (A : Type) (h : A → A) (x : A) : A := h (h (h x))
```

Repeating declarations in this way can be tedious, however. Lean provides us with the `variable` and `variables` commands to make such declarations look global:

```
variables (A B C : Type)

definition compose (g : B → C) (f : A → B) (x : A) : C := g (f x)
definition do_twice (h : A → A) (x : A) : A := h (h x)
definition do_thrice (h : A → A) (x : A) : A := h (h (h x))
```

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

```
variables (A B C : Type)
variables (g : B → C) (f : A → B) (h : A → A)
variable x : A

definition compose := g (f x)
definition do_twice := h (h x)
definition do_thrice := h (h (h x))

print compose
print do_twice
print do_thrice
```

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

The `variable` and `variables` commands look like the `constant` and `constants` commands we have used above, but there is an important difference: rather than creating permanent entities, the declarations simply tell Lean to insert the variables as bound variables in definitions that refer to them. Lean is smart enough to figure out which variables are used explicitly or implicitly in a definition. We can therefore proceed as though `A`, `B`, `C`, `g`, `f`, `h`, and `x` are fixed objects when we write our definitions, and let Lean abstract the definitions for us automatically.

When declared in this way, a variable stays in scope until the end of the file we are working on, and we cannot declare another variable with the same name. Sometimes, however, it is useful to limit the scope of a variable. For that purpose, Lean provides the notion of a `section`:

```
section useful
  variables (A B C : Type)
  variables (g : B → C) (f : A → B) (h : A → A)
  variable x : A

  definition compose := g (f x)
  definition do_twice := h (h x)
  definition do_thrice := h (h (h x))
end useful
```

When the section is closed, the variables go out of scope, and become nothing more than a distant memory.

You do not have to indent the lines within a section, since Lean treats any blocks of returns, spaces, and tabs equivalently as whitespace. 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.

2.7 Namespaces

Lean provides us with the ability to group definitions, notation, and other information into nested, hierarchical *namespaces*:

```
namespace foo
  definition a : ℕ := 5
  definition f (x : ℕ) : ℕ := x + 7

  definition fa : ℕ := f a
  definition ffa : ℕ := f (f a)

  print "inside foo"

  check a
  check f
  check fa
  check ffa
  check foo.fa
end foo

print "outside the namespace"

-- check a  -- error
-- check f  -- error
check foo.a
check foo.f
check foo.fa
check foo.ffa

open foo

print "opened foo"

check a
check f
check fa
check foo.fa
```

When we declare that we are working in the namespace `foo`, every identifier we declare has a full name with prefix "`foo.`" Within the namespace, we can refer to identifiers by their shorter names, but once we end the namespace, we have to use the longer names.

The `open` command brings the shorter names into the current context. Often, when we import a theory file, we will want to open one or more of the namespaces it contains, to have

access to the short identifiers, notations, and so on. But sometimes we will want to leave this information hidden, for example, when they conflict with identifiers and notations in another namespace we want to use. Thus namespaces give us a way to manage our working environment.

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

```
check list.nil
check list.cons
check list.append
```

We will discuss their types, below. The command `open list` allows us to use the shorter names:

```
open list

check nil
check cons
check append
```

Like sections, namespaces can be nested:

```
namespace foo
  definition a : ℕ := 5
  definition f (x : ℕ) : ℕ := x + 7

  definition fa : ℕ := f a

  namespace bar
    definition ffa : ℕ := 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
  definition a : ℕ := 5
  definition f (x : ℕ) : ℕ := x + 7

  definition fa : ℕ := f a
end foo

check foo.a
check foo.f

namespace foo
  definition ffa : ℕ := f (f a)
end foo
```

Like sections, nested namespaces have to be closed in the order they are opened. Also, a namespace cannot be opened within a section; namespaces have to live on the outer levels.

Namespaces and sections serve different purposes: namespaces organize data and sections declare variables for insertion in theorems. A namespace can be viewed as a special kind of section, however. 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.

As scoping mechanisms, namespaces and sections govern more than just variables and identifier names. We will later see that notations defined in a namespace are operant only when the namespace is open, and notation defined in a section has scope limited to the section. Similarly, if we use the `open` command inside a section or namespace, it only remains in effect until that section or namespace is closed. As a result, namespaces and sections provide useful ways of managing the background context while we work with Lean.

2.8 Dependent Types

You now have rudimentary ways of defining functions and objects in Lean, and we will gradually introduce you to many more. Our ultimate goal in Lean is to *prove* things about the objects we define, and the next chapter will introduce you to Lean's mechanisms for stating theorems and constructing proofs. Meanwhile, let us remain on the topic of defining objects in dependent type theory for just a moment longer, in order to explain what makes dependent type theory *dependent*, and why that is useful.

The short explanation is that what makes dependent type theory dependent is that types can depend on parameters. You have already seen a nice example of this: the type `list A` depends on the argument `A`, and this dependence is what distinguishes `list ℕ` and `list bool`. For another example, consider the type `vec A n`, the type of vectors of elements of `A` of length `n`. This type depends on *two* parameters: the type `A : Type` of the elements in the vector and the length `n : ℕ`.

Suppose we 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*: we expect the `cons`

function for ℕ, `bool`, or an arbitrary type `A` 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, `A`, `cons A` is the insertion function for lists of type `A`. In other words, for every `A`, `cons A` is the function that takes an element `a : A` and a list `l : list A`, and returns a new list, so we have `cons A a l : list A`.

It is clear that `cons A` should have type `A → list A → list A`. But what type should `cons` have? A first guess might be `Type → A → list A → list A`, but, on reflection, this does not make sense: the `A` in this expression does not refer to anything, whereas it should refer to the argument of type `Type`. In other words, *assuming* `A : Type` is the first argument to the function, the type of the next two elements are `A` and `list A`. These types vary depending on the first argument, `A`.

This is an instance of a *Pi type* in dependent type theory. Given `A : Type` and `B : A → Type`, think of `B` as a family of types over `A`, that is, a type `B a` for each `a : A`. In that case, the type `Π x : A, B x` denotes the type of functions `f` with the property that, for each `a : A`, `f a` is an element of `B a`. In other words, the type of the value returned by `f` depends on its input.

Notice that `Π x : A, B` makes sense for any expression `B : Type`. When the value of `B` depends on `x` (as does, for example, the expression `B x` in the previous paragraph), `Π x : A, B` denotes a dependent function type. When `B` doesn't depend on `x`, `Π x : A, B` is no different from the type `A → B`. Indeed, in dependent type theory (and in Lean), the Pi construction is fundamental, and `A → B` is nothing more than notation for `Π x : A, B` when `B` does not depend on `A`.

Returning to the example of lists, we can model some basic list operations as follows. We use `namespace hide` to avoid a naming conflict with the `list` type defined in the standard library.

```
namespace hide

universe variable u

constant list : Type u → Type u

constant cons : Π A : Type u, A → list A → list A
constant nil : Π A : Type u, list A
constant head : Π A : Type u, list A → A
constant tail : Π A : Type u, list A → list A
constant append : Π A : Type u, list A → list A → list A

end hide
```

You can enter the symbol Π by typing `\Pi`. Here, `nil` is intended to denote the empty list, `head` and `tail` return the first element of a list and the remainder, respectively. The constant `append` is intended to denote the function that concatenates two lists.

We emphasize that these constant declarations are only for the purposes of illustration. The `list` type and all these operations are, in fact, *defined* in Lean's standard library, and are proved to have the expected properties. In fact, as the next example shows, the types indicated above are essentially the types of the objects that are defined in the library. (We will explain the `@` symbol and the difference between the round and curly brackets momentarily.)

```
open list

check list     -- Type u_1 → Type u_1

check @cons    -- Π {A : Type u_1}, A → list A → list A
check @nil     -- Π {A : Type u_1}, list A
check @head    -- Π {A : Type u_1} [_inst_1 : inhabited A], list A → A
check @tail    -- Π {A : Type u_1}, list A → list A
check @append  -- Π {A : Type u_1}, list A → list A → list A
```

There is a subtlety in the definition of `head`: the type `A` is required to have at least one element, and when passed the empty list, the function must determine a default element of the relevant type. We will explain how this is done in a later chapter.

Vector operations are handled similarly:

```
universe variable u
constant vec : Type u → ℕ → Type u

namespace vec
  constant empty : Π A : Type u, vec A 0
  constant cons :
    Π (A : Type u) (n : ℕ), A → vec A n → vec A (n + 1)
  constant append :
    Π (A : Type u) (n m : ℕ),  vec A m → vec A n → vec A (n + m)
end vec
```

In the coming chapters, you will come across many instances of dependent types. Here we will mention just one more important and illustrative example, the *Sigma types*, Σ `x : A, B x`, sometimes also known as *dependent pairs*. These are, in a sense, companions to the Pi types. The type Σ `x : A, B x` denotes the type of pairs `sigma.mk a b` where `a : A` and `b : B a`.

Just as Pi types Π `x : A, B x` generalize the notion of a function type `A → B` by allowing `B` to depend on `A`, Sigma types Σ `x : A, B x` generalize the cartesian product `A × B` in the same way: in the expression `sigma.mk a b`, the type of the second element of the pair, `b : B a`, depends on the first element of the pair, `a : A`. Lean also defines the alternative syntax `dpair a b` for `sigma.mk a b`, denoting a "dependent pair."

```
variable A : Type
variable B : A → Type
variable a : A
```

```
variable b : B a

check sigma.mk a b   -- Σ (a : A), B a
check dpair a b      -- Σ (a : A), B a
check (dpair a b).1  -- A
check (dpair a b).2  -- B (sigma.fst (dpair a b))

eval  (dpair a b).1  -- a
eval  (dpair a b).2  -- b
```

Notice that when `p` is a dependent pair the expressions `(dpair a b).1` and `(dpair a b).2` are short for `sigma.fst (dpair a b)` and `sigma.snd (dpair a b)`, respectively, and that these reduce to `a` and `b`, respectively.

2.9 Implicit Arguments

Suppose we have an implementation of lists as described above.

```
namespace hide
universe variable u
constant list : Type u → Type u

namespace list
  constant cons : Π A : Type u, A → list A → list A
  constant nil : Π A : Type u, list A
  constant append : Π A : Type u, list A → list A → list A
end list
end hide
```

Then, given a type `A`, some elements of `A`, and some lists of elements of `A`, we can construct new lists using the constructors.

```
open hide.list

variable  A : Type
variable  a : A
variables l1 l2 : list A

check cons A a (nil A)
check append A (cons A a (nil A)) l1
check append A (append A (cons A a (nil A)) l1) l2
```

Because the constructors are polymorphic over types, we have to insert the type `A` as an argument repeatedly. But this information is redundant: one can infer the argument `A` in `cons A a (nil A)` from the fact that the second argument, `a`, has type `A`. One can similarly infer the argument in `nil A`, not from anything else in that expression, but from the fact that it is sent as an argument to the function `cons`, which expects an element of type `list A` 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 cons _ a (nil _)
check append _ (cons _ a (nil _)) l1
check append _ (append _ (cons _ a (nil _)) l1) l2
```

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 us to specify that this argument should, by default, be left implicit. This is done by putting the arguments in curly braces, as follows:

```
namespace list
  constant cons : Π {A : Type u}, A → list A → list A
  constant nil : Π {A : Type u}, list A
  constant append : Π {A : Type u}, list A → list A → list A
end list

open hide.list

variable  A : Type
variable  a : A
variables l1 l2 : list A

check cons a nil
check append (cons a nil) l1
check append (append (cons a nil) l1) l2
```

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

```
universe variable u
definition ident {A : Type u} (x : A) := x

variables A B : Type u
variables (a : A) (b : B)

check ident      -- ?M_1 → ?M_1
check ident a    -- A
check ident b    -- B
```

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 declared implicit when they are declared with the `variables` command:

```
universe variable u

section
  variable {A : Type u}
  variable x : A
  definition ident := x
end

variables A B : Type u
variables (a : A) (b : B)

check ident
check ident a
check ident b
```

This definition of `ident` 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. The second pair of examples below use this mechanism to specify the desired types of the expressions `id` and `list.nil`:

```
check list.nil             -- list ?M1
check id                   -- ?M1 → ?M1

check (list.nil : list ℕ)  -- list ℕ
check (id : ℕ → ℕ)          -- ℕ → ℕ
```

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 are elaborated in the same way, whereas the third `check` command interprets `2` as a raw numeral.

```
check 2            -- ℕ
check (2 : ℕ)       -- ℕ
check (2 : num)    -- num
```

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         -- Π {A : Type u_1}, A → A
check @id A       -- A → A
check @id B       -- B → B
check @id A a     -- A
check @id B b     -- B
```

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.

3

Propositions and Proofs

By now, you have seen how to define some elementary notions in dependent type theory.

In this chapter, we will explain how mathematical propositions and proofs are expressed in the language of dependent type theory, so that you can start proving assertions about the objects and notations that have been defined. The encoding we use here is specific to the standard library; we will discuss proofs in *homotopy type theory* in a later chapter.

3.1 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 constructors to build new propositions from others.

```
constant and : Prop → Prop → Prop
constant or : Prop → Prop → Prop
constant not : Prop → Prop
constant implies : Prop → Prop → Prop

variables 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 constant of such a type.

```
constant Proof : Prop → Type

constant and_comm : Π p q : Prop, Proof (implies (and p q) (and q p))

variables 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:

```
constant 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:

```
constant 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 Inductive 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 `Type 0`, the very bottom of the type hierarchy described in the last chapter. `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, which 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 `(λ 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 that the bottom line is clear. 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.

3.2 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:

```
constants p q : Prop

theorem t1 : p → q → p := λ hp : p, λ hq : q, hp
```

This looks exactly like the definition of the constant function in the last chapter, the only difference being that the arguments 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 `definition` 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 it when processing a file. Moreover, for efficiency purposes, Lean treats theorems as axiomatic constants within the file in which they are defined. This makes it possible to process and check theorems in parallel, since theorems later in a file do not make use of the contents of earlier proofs.

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

```
theorem t1 : p → q → p := λ hp : p, λ hq : q, hp

print t1
```

(To save space, the online version of Lean does not store proofs of theorems in the library, so you cannot print them in the browser interface.)

Notice that the lambda abstractions `hp : p` and `hq : q` can be viewed as temporary assumptions in the proof of `t1`. Lean provides the alternative syntax `assume` for such a lambda abstraction:

```
theorem t1 : p → q → p :=
assume hp : p,
assume hq : q,
hp
```

Lean also allows us to specify the type of the final term `hp`, explicitly, with a `show` statement.

```
theorem t1 : p → q → p :=
assume hp : p,
assume 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. Lean also allows you to use the alternative syntax `proposition`, `lemma`, or `corollary` instead of theorem:

```
lemma t1 : p → q → p :=
assume hp : p,
assume hq : q,
show p, from hp
```

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

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

check t1 -- p → q → p
```

Now we can apply the theorem `t1` just as a function application.

```
axiom hp : p

theorem t2 : q → p := t1 hp
```

Here, the `axiom` command is alternative syntax for `constant`. Declaring a "constant" `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 `t2 : q → p`.

Notice, by the way, that the original theorem `t1` is true for *any* propositions `p` and `q`, not just the particular constants declared. So it would be more natural to define the theorem so that it quantifies over those, too:

```
theorem t1 (p q : Prop) (hp : p) (hq : q) : p := hp
check 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`". The symbol ∀ is alternate syntax for Π, and later we will see how Pi types let us model universal quantifiers more generally. For the moment, however, we will focus on theorems in propositional logic, generalized over the propositions. We will tend to work in sections with variables over the propositions, so that they are generalized for us automatically.

When we generalize `t1` in that 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

variables 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
```

Remember that under the propositions-as-types correspondence, a variable `h` of type `r → s` can be viewed as the hypothesis, or premise, that `r → s` holds. For that reason, Lean offers the alternative syntax, `premise`, for `variable`.

```
premise h : r → s
check t1 (r → s) (s → r) h
```

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

```
variables p q r s : Prop

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

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

Lean allows the alternative syntax `premise` and `premises` for `variable` and `variables`. This makes sense, of course, for variables whose type is an element of `Prop`. It is also often

useful to use numeric unicode subscripts, entered as `\0`, `\1`, `\2`, ..., for hypotheses. The following definition of `t2` has the same net effect as the preceding one.

```
variables p q r s : Prop
premises (h₁ : q → r) (h₂ : p → q)

theorem t2 : p → r :=
assume h₃ : p,
show r, from h₁ (h₂ h₃)
```

3.3 Propositional Logic

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

Ascii	Unicode	Emacs shortcut for unicode	Definition
true			true
false			false
not	¬	`\not`, `\neg`	not
/\	∧	`\and`	and
\/	∨	`\or`	or
->	→	`\to`, `\r`, `\implies`	
<->	↔	`\iff`, `\lr`	iff

They all take values in `Prop`.

```
variables p q : Prop

check p → q → p ∧ q
check ¬p → p ↔ false
check p ∨ q → q ∨ p
```

The order of operations is fairly standard: unary negation ¬ binds most strongly, then ∧ and ∨, and finally → and ↔. 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 the standard library in the file `init.datatypes`, and each comes with its canonical introduction and elimination rules.

Conjunction

The expression `and.intro h1 h2` creates a proof for `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`.

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

check assume (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.elim_left H` creates a proof of `p` from a proof `h : p ∧ q`. Similarly, `and.elim_right H` is a proof of `q`. They are commonly known as the right and left *and-elimination* rules.

```
example (h : p ∧ q) : p := and.elim_left h
example (h : p ∧ q) : q := and.elim_right h
```

Because they are so commonly used, the standard library provides the abbreviations `and.left` and `and.right` for `and.elim_left` and `and.elim_right`, respectively.

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

```
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 `pair 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 a later chapter 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 can be inferred from the context. In particular, we can often write ⟨`hp, hq`⟩ instead of `and.intro hp hq`:

```
variables p q : Prop
premises  (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 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 three expressions all mean the same thing:

```
variable l : list ℕ

check list.head l
check l^.head
check l.head
```

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:

```
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 as 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:

```
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.

```
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`.

```
example (h : p ∨ q) : q ∨ p :=
or.elim h
  (assume hp : p,
    show q ∨ p, from or.intro_right q hp)
  (assume 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` as shorthands for `or.intro_right _` and `or.intro_left _`. Thus the proof term above could be written more concisely:

```
example (h : p ∨ q) : q ∨ p := or.elim h (λ hp, or.inr hp) (λ 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`:

```
example (h : p ∨ q) : q ∨ p :=
h^.elim
  (assume hp : p, or.inr hp)
  (assume hq : q, or.inl hq)
```

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

Negation and Falsity

The expression `not.intro h` produces a proof of `¬p` from `h : p → false`. That is, we obtain `¬p` if we can derive 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`.

```
example (hpq : p → q) (hnq : ¬q) : ¬p :=
assume 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*.

```
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`.

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

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

```
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`, sometimes abbreviated `trivial : true`. In other words, `true` is simply true, and has a canonical proof, `trivial`.

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.elim_left H` produces a proof of `p → q` from `h : p ↔ q`. Similarly, `iff.elim_right H` produces a proof of `q → p` from `h : p ↔ q`. Here is a proof of `p ∧ q ↔ q ∧ p`:

```
theorem and_swap : p ∧ q ↔ q ∧ p :=
iff.intro
  (assume h : p ∧ q,
    show q ∧ p, from and.intro (and.right h) (and.left h))
  (assume h : q ∧ p,
    show p ∧ q, from and.intro (and.right h) (and.left h))

check and_swap p q    -- p ∧ q ↔ q ∧ p
```

Because they represent a form of *modus ponens*, `iff.elim_left` and `iff.elim_right` can be abbreviated `iff.mp` and `iff.mpr`, respectively. In the next example, we use that theorem to derive `q ∧ p` from `p ∧ q`:

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

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

```
theorem and_swap : p ∧ q ↔ q ∧ p :=
⟨ λ h, ⟨h^.right, h^.left⟩, λ h, ⟨h^.right, h^.left⟩ ⟩

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

3.4 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:

```
section
  variables p q : Prop

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

Internally, the expression `have h : p, from s, t` produces the term `(λ (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.

3.5 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:

```
theorem dne {p : Prop} (h : ¬¬p) : p :=
or.elim (em p)
  (assume hp : p, hp)
  (assume 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 gives 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:

```
example (h : ¬¬p) : p :=
by_cases
  (assume h1 : p, h1)
  (assume h1 : ¬p, absurd h1 h)
```

Or you can carry out a proof by contradiction:

```
example (h : ¬¬p) : p :=
by_contradiction
  (assume 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)
  (assume hp : p,
    or.inr
      (show ¬q, from
        assume hq : q,
        h ⟨hp, hq⟩))
```

```
  (assume 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.

There are additional classical axioms that are not included by default in the standard library. We will discuss these in detail in a later chapter.

3.6 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. In this section, we will review some common identities, and encourage you to try proving them on your own using the rules above.

The following is a long list of assertions in propositional logic. Prove as many as you can, using the rules introduced above to replace the `sorry` placeholders by actual proofs. The ones that require classical reasoning are grouped together at the end, while the rest are constructively valid.

```
open classical

variables p q r s : 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 ↔ ¬p) := sorry
example : (p → q) → (¬q → ¬p) := sorry
```

```
-- these require classical reasoning
example : (p → r ∨ s) → ((p → r) ∨ (p → s)) := 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
```

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 that 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." This is followed by the type of the term it is expecting, and all the objects and hypothesis 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

variables p q r : Prop

-- distributivity
example : p ∧ (q ∨ r) ↔ (p ∧ q) ∨ (p ∧ r) :=
iff.intro
  (assume h : p ∧ (q ∨ r),
    have hp : p, from h^.left,
    or.elim (h^.right)
      (assume hq : q,
        show (p ∧ q) ∨ (p ∧ r), from or.inl ⟨hp, hq⟩)
      (assume hr : r,
        show (p ∧ q) ∨ (p ∧ r), from or.inr ⟨hp, hr⟩))
  (assume h : (p ∧ q) ∨ (p ∧ r),
    or.elim h
      (assume hpq : p ∧ q,
        have hp : p, from hpq^.left,
        have hq : q, from hpq^.right,
        show p ∧ (q ∨ r), from ⟨hp, or.inl hq⟩)
      (assume hpr : p ∧ r,
        have hp : p, from hpr^.left,
        have hr : r, from hpr^.right,
        show p ∧ (q ∨ r), from ⟨hp, or.inr hr⟩))
```

```
-- an example that requires classical reasoning
example : ¬(p ∧ ¬q) → (p → q) :=
assume h : ¬(p ∧ ¬q),
assume hp : p,
show q, from
  or.elim (em q)
    (assume hq : q, hq)
    (assume hnq : ¬q, absurd (and.intro hp hnq) h)
```

4

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.

4.1 The Universal Quantifier

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

The universal quantifier, `∀ x : A, p x` is supposed to denote the assertion that "for every `x : A`, `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 : A` is arbitrary, we obtain a proof `∀ x : A, p x`.

The elimination rule states:

> Given a proof `∀ x : A, p x` and any term `t : A`, 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 Pi types:

> Given a term `t` of type `B x`, in a context where `x : A` is arbitrary, we have `(λ x : A, t) : Π x : A, B x`.

The elimination rule states:

> Given a term `s : Π x : A, B x` and any term `t : A`, we have `s t : B t`.

In the case where `p x` has type `Prop`, if we replace `Π x : A, B x` with `∀ x : A, p x`, we can read these as the correct rules for building proofs involving the universal quantifier.

The Calculus of Inductive Constructions therefore identifies Π and ∀ in this way. If `p` is any expression, `∀ x : A, p` is nothing more than alternative notation for `Π x : A, p`, with the idea that the former is more natural than the latter in cases where where `p` is a proposition. Typically, the expression `p` will depend on `x : A`. Recall that, in the case of ordinary function spaces, we could interpret `A → B` as the special case of `Π x : A, B` in which `B` 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.

```
variables (A : Type) (p q : A → Prop)

example : (∀ x : A, p x ∧ q x) → ∀ y : A, p y  :=
assume h : ∀ x : A, p x ∧ q x,
take y : A,
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 : A, p y` is to take an arbitrary `y`, and prove `p y`. This is the introduction rule. Now, given that `h` has type `∀ x : A, 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, or chosen the variable `z` instead of `y` in the proof:

```
example : (∀ x : A, p x ∧ q x) → ∀ y : A, p y  :=
assume h : ∀ x : A, p x ∧ q x,
take z : A,
show p z, from and.elim_left (h z)
```

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

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

variables (a b c : A)
variables (hab : r a b) (hbc : r b c)

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

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:

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

variables (a b c : A)
variables (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 `check` command contains expressions like `?z A r trans_r a b c hab hbc`. Such an expression indicates an arbitrary value, that may depend on any of the values listed (in this case, all the variables in the local context).

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

```
variables (A : Type) (r : A → A → 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 : A) (hab : r a b) (hcb : r c b) (hcd : r c d) : r a d :=
trans_r (trans_r hab (symm_r hcb)) hcd
```

You might want to try to prove some of these equivalences:

```
variables (A : Type) (p q : A → 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.

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

```
variables (A : Type) (p q : A → Prop)
variable r : Prop

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

As a final example, 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 implies a contradiction:

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

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

It is the typing rule for Pi types, and the universal quantifier in particular, that distinguishes `Prop` from other types. Suppose we have `A : Type i` and `B : Type j`, where the expression `B` may depend on a variable `x : A`. Then `Π x : A, B` is an element of `Type (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 `Π x : A, B` is an element of `Type (max i j)`. In other words, the type of dependent functions from `A` to `B` "lives" in the universe with smallest index greater-than or equal to the indices of the universes of `A` and `B`. Suppose, however, that `B` is of `Type 0`, that is, an element of `Prop`. In that case, `Π x : A, B` is an element of `Type 0` as well, no matter which type universe `A` lives in. In other words, if `B` is a proposition depending on `A`, then `∀ x : A, B` 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 `A` is any type, we

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

4.2 Equality

Let us now turn to one of the most fundamental relations defined in Lean's library, namely, the equality relation. In a later chapter,

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    -- ∀ (a : ?M_1), a = a
check eq.symm    -- ?M_2 = ?M_3 → ?M_3 = ?M_2
check eq.trans   -- ?M_2 = ?M_3 → ?M_3 = ?M_4 → ?M_2 = ?M_4
```

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

```
variables (A : Type) (a b c d : A)
premises (hab : a = b) (hcb : c = b) (hcd : c = d)

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

If we "open" the eq namespace, the names become shorter:

```
open eq

example : a = d := trans (trans hab (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 Inductive 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:

```
variables (A B : Type)

example (f : A → B) (a : A) : (λ x, f x) a = f a := eq.refl _
example (a : A) (b : A) : (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 : A → B) (a : A) : (λ x, f x) a = f a := rfl
example (a : A) (b : A) : (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 (A : Type) (a b : A) (P : A → Prop) (h1 : a = b) (h2 : P a) : P b :=
eq.subst h1 h2

example (A : Type) (a b : A) (P : A → Prop) (h1 : a = b) (h2 : P a) : P b :=
h1 ▸ h2
```

The triangle in the second presentation is nothing more than notation for `eq.subst`, and you can enter it by typing `\t`.

It is often important to be able to carry out substitutions like this by hand, but it is tedious to prove examples like the one above in this way. Fortunately, Lean provides an environment that provides better support for such calculations, which we will turn to now.

4.3 The Calculation Environment

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 calculation 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
```

Each `<proof>_i` is a proof for `<expr>_{i-1} op_i <expr>_i`. The `<proof>_i` may also be of the form `{ <pr> }`, where `<pr>` is a proof for some equality `a = b`. The form `{ <pr>`

`}` is just syntactic sugar for `eq.subst <pr> (eq.refl <expr>_{i-1})` In other words, we are claiming we can obtain `<expr>_i` by replacing `a` with `b` in `<expr>_{i-1}`.

Here is an example:

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

include h3
theorem T : a = e :=
calc
  a     = b       : h1
    ... = c + 1   : h2
    ... = d + 1   : by rewrite h3
    ... = 1 + d   : nat.add_comm d (1 : ℕ)
    ... =  e      : eq.symm h4
```

We will explain the expression `include h3` in the next chatper. The third line of the proof uses the `rewrite` tactic, which we will also discuss 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.

```
attribute [trans] nat.le_trans
attribute [trans] nat.lt_of_lt_of_le
attribute [trans] nat.lt_trans

theorem T2 (a b c d : ℕ)
  (h1 : a = b) (h2 : b ≤ c) (h3 : c + 1 < d) : a < d :=
calc
  a     = b     : h1
    ... < b + 1 : nat.self_lt_succ b
    ... ≤ c + 1 : nat.succ_le_succ h2
    ... < d     : h3
```

4.4 The Existential Quantifier

Finally, consider the existential quantifier, which can be written as either `exists x : A, p x` or `∃ x : A, p x`. Both versions are actually notationally convenient abbreviations for a more long-winded expression, `Exists (λ x : A, 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 : A, p x`, it suffices to provide a suitable term `t` and a proof of `p t`. here are some examples:

```
open nat

example : ∃ x : ℕ, x > 0 :=
```

```
have h : 1 > 0, from zero_lt_succ 0,
exists.intro 1 h

example (x : ℕ) (h : x > 0) : ∃ y, y < x :=
exists.intro 0 h

example (x y z : ℕ) (hxy : x < y) (hyz : y < z) : ∃ w, x < w ∧ w < z :=
exists.intro y (and.intro hxy hyz)

check @exists.intro
```

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

```
example : ∃ x : ℕ, x > 0 :=
⟨1, zero_lt_succ 0⟩

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

example (x y z : ℕ) (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 : A → Prop` in the conclusion `∃ x, p x`. This is not a trivial affair. For example, if we have 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.implicit` to true to ask Lean's pretty-printer to show the implicit arguments.

```
variable g : ℕ → ℕ → ℕ
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.implicit true  -- display implicit arguments
check gex1
check gex2
check gex3
check gex4
```

We can view `exists.intro` as an information-hiding operation: we are "hiding" 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 : A, 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 the `q` follows from the existence of any such `x`. Here is an example:

```
variables (A : Type) (p q : A → Prop)

example (h : ∃ x, p x ∧ q x) : ∃ x, q x ∧ p x :=
exists.elim h
  (take w,
    assume 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 : A, p x` can be thought of as a big disjunction of the propositions `p a`, as `a` ranges over all the elements of `A`.

Notice that an existential proposition is very similar to a sigma type, as described in are very similar to the sigma types described in Section 2.8. The difference is that given `a : A` and `h : p a`, `exists.intro a h` has type `(∃ x : A, p x) : Prop` and `sigma.mk a h` has type `(Σ x : A, 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` statement:

```
variables (A : Type) (p q : A → Prop)

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

The `match` statement is part of Lean's function definition system, which provides a convenient and expressive ways of defining complex functions in the system. It is, once again, 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 : A), (hw : p w ∧ q w)⟩ :=
  ⟨w, hw^.right, hw^.left⟩
end
```

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⟩
end
```

Lean will even allow us to use an implicit `match` in the `assume` statement:

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

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

variables (A : Type) (p : A → Prop)

example (h : ¬ ∀ x, ¬ p x) : ∃ x, p x :=
by_contradiction
  (assume h1 : ¬ ∃ x, p x,
    have h2 : ∀ x, ¬ p x, from
      take x,
      assume h3 : p x,
      have h4 : ∃ x, p x, from  ⟨x, h3⟩,
      show false, from h1 h4,
    show false, from h h2)
```

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

```
open classical

variables (A : Type) (p q : A → Prop)
variable a : A
variable r : Prop

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

Notice that the declaration `variable a : A` amounts to the assumption that there is at least one element of type `A`. This assumption is needed in the second example, as well as in the last two.

Here are solutions to two of the more difficult ones:

```
example : (∃ x, p x ∨ q x) ↔ (∃ x, p x) ∨ (∃ x, q x) :=
iff.intro
  (assume ⟨a, (h1 : p a ∨ q a)⟩,
    or.elim h1
      (assume hpa : p a, or.inl ⟨a, hpa⟩)
      (assume hqa : q a, or.inr ⟨a, hqa⟩))
  (assume h : (∃ x, p x) ∨ (∃ x, q x),
    or.elim h
      (assume ⟨a, hpa⟩, ⟨a, (or.inl hpa)⟩)
      (assume ⟨a, hqa⟩, ⟨a, (or.inr hqa)⟩))

example : (∃ x, p x → r) ↔ (∀ x, p x) → r :=
iff.intro
  (assume ⟨b, (hb : p b → r)⟩,
    assume h2 : ∀ x, p x,
    show r, from  hb (h2 b))
  (assume h1 : (∀ x, p x) → r,
    show ∃ x, p x → r, from
      by_cases
        (assume hap : ∀ x, p x, ⟨a, λ h', h1 hap⟩)
        (assume hnap : ¬ ∀ x, p x,
          by_contradiction
            (assume hnex : ¬ ∃ x, p x → r,
              have hap : ∀ x, p x, from
                take x,
                by_contradiction
                  (assume hnp : ¬ p x,
                    have hex : ∃ x, p x → r,
                      from ⟨x, (assume hp, absurd hp hnp)⟩,
                    show false, from hnex hex),
              show false, from hnap hap)))
```

4.5 More on the Proof Language

We have seen that keywords like `assume`, `take`, `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 : ℕ → ℕ
premise h : ∀ x : ℕ, f x ≤ f (x + 1)
```

```
example : f 0 ≤ f 3 :=
have f 0 ≤ f 1, from h 0,
have f 0 ≤ f 2, from 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`:

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

example : f 0 ≤ f 3 :=
have f 0 ≤ f 1, from h 0,
have f 0 ≤ f 2, from 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 abou 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.

```
example : f 0 ≥ f 1 → f 1 ≥ f 2 → f 0 = f 2 :=
suppose f 0 ≥ f 1,
suppose f 1 ≥ f 2,
have f 0 ≥ f 2, from nat.le_trans this ‹f 0 ≥ f 1›,
have f 0 ≤ f 2, from nat.le_trans (h 0) (h 1),
show f 0 = f 2, from nat.le_antisymm this ‹f 0 ≥ f 2›
```

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:

```
example : f 0 ≤ f 3 :=
have f 0 ≤ f 1, from h 0,
have f 1 ≤ f 2, from h 1,
have f 2 ≤ f 3, from h 2,
show f 0 ≤ f 3, from nat.le_trans ‹f 0 ≤ f 1›
                    (nat.le_trans ‹f 1 ≤ f 2› ‹f 2 ≤ f 3›)
```

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

```
example (n : ℕ) : ℕ := ‹ℕ›
```

The `suppose` keyword acts as an anonymous assume:

```
example : f 0 ≥ f 1 → f 0 = f 1 :=
suppose f 0 ≥ f 1,
show f 0 = f 1, from nat.le_antisymm (h 0) this
```

Notice that there is an asymmetry: you can use `have` with or without a label, but if you do not wish to name the assumption, you must use `suppose` rather than `assume`. The reason is that Lean allows us to write `assume h` to introduce a hypothesis without specifying it, leaving it to the system to infer to relevant assumption. An anonymous `assume` would thus lead to ambiguities when parsing expressions.

As with the anonymous `have`, when you use `suppose` to introduce an assumption, that assumption can also be invoked later in the proof by enclosing it in backticks.

```
example : f 0 ≥ f 1 → f 1 ≥ f 2 → f 0 = f 2 :=
suppose f 0 ≥ f 1,
suppose f 1 ≥ f 2,
have f 0 ≥ f 2, from nat.le_trans ‹f 2 ≤ f 1› ‹f 1 ≤ f 0›,
have f 0 ≤ f 2, from nat.le_trans (h 0) (h 1),
show f 0 = f 2, from nat.le_antisymm this ‹f 0 ≥ f 2›
```

Notice that `le_antisymm` is the assertion that if `a ≤ b` and `b ≤ a` then `a = b`, and `a ≥ b` is definitionally equal to `b ≤ a`.

5

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, we 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 users 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 exmample, 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.

5.1 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
```

We 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, we meet such a goal by writing an explicit term. But wherever a term is expected, Lean allows us to insert instead a `begin ... end` block, followed by a sequence of commands, separated by commas. We can prove the theorem above in that way:

```
theorem test (p q : Prop) (hp : p) (hq : q) : p ∧ q ∧ p :=
begin
  apply and.intro,
  exact hp,
  apply and.intro,
  exact hq,
  exact hp
end
```

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:

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

⊢ q ∧ p
```

For brevity, Lean only displays the context for the first goal, which is the one addressed by the next tactic command. 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; but otherwise `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. If you run Lean on an incomplete tactic proof bracketed by `begin` and `end`, the system reports all the unsolved goals that remain.

If you are running Lean with its Emacs interface, you can see this information by putting your cursor on the `end` symbol, which should be underlined. In the Emacs interface, there is another extremely useful trick: if you put your cursor on a line of a tactic proof and press "C-c C-g", Lean will show you the goal that remains at the end of the line.

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 :=
begin
  apply (and.intro hp),
  exact (and.intro hq hp)
end
```

Unsurprisingly, it produces exactly the same proof term.

```
print test
```

Whenever a proof term is expected, instead of using a `begin...end` block, you can write the `by` keyword followed by a single tactic:

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

In the Lean Emacs mode, if you put your cursor on the "b" in "by" and press "C-c C-g", Lean shows you the goal that the tactic is supposed to meet.

5.2 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 Section 3.5 but now prove using tactics. We adopt the following convention regarding indentation: whenever a tactic introduces one or more additional subgoals, we indent another two spaces, until the additional subgoals are deleted.

```
example (p q r : Prop) : p ∧ (q ∨ r) ↔ (p ∧ q) ∨ (p ∧ r) :=
begin
  apply iff.intro,
    intro h,
    apply (or.elim (and.elim_right h)),
      intro hq,
      apply or.intro_left,
      apply and.intro,
        exact (and.elim_left h),
      exact hq,
    intro hr,
```

```
    apply or.intro_right,
    apply and.intro,
    exact (and.elim_left h),
    exact hr,
  intro h,
  apply (or.elim h),
    intro hpq,
    apply and.intro,
      exact (and.elim_left hpq),
    apply or.intro_left,
    exact (and.elim_right hpq),
  intro hpr,
  apply and.intro,
    exact (and.elim_left hpr),
  apply or.intro_right,
  exact (and.elim_right hpr)
end
```

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

```
example (A : Type) : A → A :=
begin
  intro a,
  exact a
end

example (A : Type) : ∀ x : A, x = x :=
begin
  intro x,
  exact eq.refl x
end
```

It has a plural form, `intros`, which takes a list of names.

```
example : ∀ a b c : nat, a = b → a = c → c = b :=
begin
  intros a b c h₁ h₂,
  exact eq.trans (eq.symm h₂) h₁
end
```

The `intros` command can also be used without any arguments, in which case, it chooses names and introduces as many variables as it can. We 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 (h₁ : x = y) (h₂ : y = z) (h₃ : z = w) : x = w :=
begin
  apply (eq.trans h₁),
  apply (eq.trans h₂),
  assumption   -- applied h₃
end
```

It will unify metavariables in the conclusion if necessary:

```
example (h₁ : x = y) (h₂ : y = z) (h₃ : z = w) : x = w :=
begin
  apply eq.trans,
  assumption,     -- solves x = ?b with h₁
  apply eq.trans,
  assumption,     -- solves ?b = w with h₂
  assumption      -- solves z = w with h₃
end
```

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 :=
begin
  intros,
  apply eq.trans,
  apply eq.symm,
  assumption,
  assumption
end
```

Incidentally, there are tactics `reflexivity`, `symmetry`, and `transitivity`, which apply the corresponding operation. Using reflexivity, for example, is more general than writing `apply eq.refl`, because it works for any relation that has been tagged with the `refl` attribute.

With that tactic, the previous proof can be written more elegantly as follows:

```
example : ∀ a b c : nat, a = b → a = c → c = b :=
begin
  intros,
  transitivity,
  symmetry,
  assumption,
  assumption
end
```

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

```
variables x y z : ℕ

example : x = x :=
begin
  generalize x z, -- goal is x : ℕ ⊢ ∀ (z : ℕ), z = z
  intro y,      -- goal is x y : ℕ ⊢ y = y
  reflexivity
end
```

Te `generalize` tactic generalizes the conclusion over the variable `x`, using a universal quantifier over `z`. We generalize any term, not just a variable:

```
example : x + y + z = x + y + z :=
begin
  generalize (x + y + z) w, -- goal is x y z : ℕ ⊢ ∀ (w : ℕ), w = w
  intro u,                  -- goal is x y z u : ℕ ⊢ u = u
  reflexivity
end
```

Notice that once we generalize over `x + y + z`, the variables `x y z : ℕ` in the context become irrelevant. The `clear` tactic throws away elements of the context, when it is safe to do so:

```
example : x + y + z = x + y + z :=
begin
  generalize (x + y + z) w, -- goal is x y z : ℕ ⊢ ∀ (w : ℕ), w = w
  clear x y z,
  intro u,                  -- goal is u : ℕ ⊢ u = u
  reflexivity
end
```

Another useful tactic is the `revert` tactic, which moves an element of the context into the goal. When applied to a variable, it has the same effect as `generalize` and `clear`:

```
example (x : ℕ) : x = x :=
begin
  revert x,     -- goal is ⊢ ∀ (x : ℕ), x = x
  intro y,      -- goal is y : ℕ ⊢ y = y
  reflexivity
end
```

Moving a hypothesis into the goal yields an implication:

```
example (x y : ℕ) (h : x = y) : y = x :=
begin
  revert h,     -- goal is x y : ℕ ⊢ x = y → y = x
  intro h₁,     -- goal is x y : ℕ, h₁ : x = y ⊢ y = x
  symmetry,
  assumption
end
```

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 exmaple, reverting `x` in the example above brings `h` along with it:

```
example (x y : ℕ) (h : x = y) : y = x :=
begin
  revert x,     -- goal is y : ℕ ⊢ ∀ (x : ℕ), x = y → y = x
```

```
  intros,
  symmetry,
  assumption
end
```

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

```
example (x y : ℕ) (h : x = y) : y = x :=
begin
  revert x y,     -- goal is x y : ℕ ⊢ ∀ (x : ℕ), x = y → y = x
  intros,
  symmetry,
  assumption
end
```

5.3 The Rewrite Tactic

The `rewrite` tactic provide a basic mechanism for applying substitutions to goals and hypotheses, providing a convenient and efficient way of working with equality.

The `rewrite` tactic has many features. The most basic form of the tactic is `rewrite t`, where `t` is a term which conclusion is an equality. In the following example, we use this basic form to rewrite the goal using a hypothesis.

```
open nat
variables (f : nat → nat) (k : nat)

example (h₁ : f 0 = 0) (h₂ : k = 0) : f k = 0 :=
begin
  rewrite h₂, -- replace k with 0
  rewrite h₁  -- replace f 0 with 0
end
```

In the example above, the first `rewrite` tactic replaces `k` with `0` in the goal `f k = 0`. Then, the second `rewrite` replace `f 0` with `0`. The `rewrite` tactic automatically closes any goal of the form `t = t`.

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

```
open nat
variables (f : nat → nat) (k : nat)

example (h₁ : f 0 = 0) (h₂ : k = 0) : f k = 0 :=
begin
  rewrite [h₂, h₁]
end
```

By default, the `rewrite` tactic 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.

```
open nat
variables (f : nat → nat) (a b : nat)

example (h₁ : a = b) (h₂ : f a = 0) : f b = 0 :=
begin
  rewrite [-h₁, h₂]
end
```

In this example, the term `-h₁` instructs the `rewriter` to replace `b` with `a`.

6

Inductive Types

We have seen that Lean's formal foundation includes basic types, `Prop`, `Type.{1}`, `Type.{2}`, ..., and allows for the formation of dependent function types, Π `x : A. B`. 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 Pi 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, Pi 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 : Type
| constructor_1 : ... → foo
| constructor_2 : ... → foo
...
| constructor_n : ... → 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 to 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 Pi type constructed from `foo` and previously defined types, in which `foo` appears, if at all, only as the "target" of the Pi type. For more details, see [2].

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.

6.1 Enumerated Types

The simplest kind of inductive type is simply a type with a finite, enumerated list of elements.

```
inductive weekday : Type
| 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
```

Think of the `sunday`, `monday`, … as being distinct elements of `weekday`, with no other distinguishing properties. The elimination principle, `weekday.rec`, is defined at the same

time as 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 a slight (automatically generated) variant, `weekday.rec_on`, which takes its arguments in a more convenient order. Note that the shorter versions of names like `weekday.rec` and `weekday.rec_on` are not made available by default when we open the `weekday` namespace, to avoid clashes. If we import `nat`, we can use `rec_on` to define a function from weekday to the natural numbers:

```
definition number_of_day (d : weekday) : ℕ :=
weekday.rec_on d 1 2 3 4 5 6 7

eval number_of_day weekday.sunday
eval number_of_day weekday.monday
eval number_of_day weekday.tuesday
```

The first (explicit) argument to `rec_on` is the element being "analyzed." The next seven arguments are the values corresponding to the seven constructors. Note that `number_of_day weekday.sunday` evaluates to `1`: the computation rule for `rec_on` recognizes that `sunday` is a constructor, and returns the appropriate argument.

Below we will encounter a more restricted variant of `rec_on`, namely, `cases_on`. When it comes to enumerated types, `rec_on` and `cases_on` are the same. You may prefer to use the label `cases_on`, because it emphasizes that the definition is really a definition by cases.

```
definition number_of_day (d : weekday) : ℕ :=
weekday.cases_on d 1 2 3 4 5 6 7
```

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 `number_of_day` function in the `weekday` namespace. We are then allowed to use the shorter name when we open the namespace.

The names `rec_on`, `cases_on`, `induction_on`, and so on are generated automatically. As noted above, they are *protected* to avoid name clashes. In other words, they are not provided by default when the namespace is opened. However, you can explicitly declare abbreviations for them using the `renaming` option when you open a namespace.

```
namespace weekday
  @[reducible]
  private definition cases_on := @weekday.cases_on

  definition number_of_day (d : weekday) : nat :=
  cases_on d 1 2 3 4 5 6 7
end weekday
```

```
eval weekday.number_of_day weekday.sunday

open weekday (renaming cases_on → cases_on)

eval number_of_day sunday
check cases_on
```

We can define functions from `weekday` to `weekday`:

```
namespace weekday
  definition next (d : weekday) : weekday :=
  weekday.cases_on d monday tuesday wednesday thursday friday saturday sunday

  definition previous (d : weekday) : weekday :=
  weekday.cases_on d saturday sunday monday tuesday wednesday thursday friday

  eval next (next tuesday)
  eval next (previous 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`? The induction principle parallels the recursion principle: we simply have to provide a proof of the claim for each constructor:

```
  theorem next_previous (d: weekday) : next (previous d) = d :=
  weekday.induction_on d
    (show next (previous sunday) = sunday, from rfl)
    (show next (previous monday) = monday, from rfl)
    (show next (previous tuesday) = tuesday, from rfl)
    (show next (previous wednesday) = wednesday, from rfl)
    (show next (previous thursday) = thursday, from rfl)
    (show next (previous friday) = friday, from rfl)
    (show next (previous saturday) = saturday, from rfl)
```

In fact, `induction_on` is just a special case of `rec_on` where the target type is an element of `Prop`. In other words, under the propositions-as-types correspondence, the principle of induction is a type of definition by recursion, where what is being "defined" is a proof instead of a piece of data. We could equally well have used `cases_on`:

```
  theorem next_previous (d: weekday) : next (previous d) = d :=
  weekday.cases_on d
    (show next (previous sunday) = sunday, from rfl)
    (show next (previous monday) = monday, from rfl)
    (show next (previous tuesday) = tuesday, from rfl)
    (show next (previous wednesday) = wednesday, from rfl)
    (show next (previous thursday) = thursday, from rfl)
    (show next (previous friday) = friday, from rfl)
    (show next (previous saturday) = saturday, from rfl)
```

While the `show` commands make the proof clearer and more readable, they are not necessary:

```
theorem next_previous (d: weekday) : next (previous d) = d :=
weekday.cases_on d rfl rfl rfl rfl rfl rfl rfl
```

Some fundamental data types in the Lean library are instances of enumerated types.

```
inductive empty : Type

inductive unit : Type
| star : unit

inductive bool : Type
| ff : bool
| tt : bool
```

(To run these examples, we put them in a namespace called `hide`, 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 with the system is started.)

The type `empty` is an inductive datatype with no constructors. The type `unit` has a single element, `star`, and the type `bool` represents the familiar boolean values. 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 `band`, `bor`, `bnot` on the boolean, and verifying common identities. Note that defining a binary operation like `band` will require nested cases splits:

```
definition band (b1 b2 : bool) : bool :=
bool.cases_on b1
  ff
  (bool.cases_on b2 ff tt)
```

Similarly, most identities can be proved by introducing suitable case splits, and then using `rfl`.

6.2 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:

```
universe variables u v

inductive prod (A : Type u) (B : Type v)
| mk : A → B → prod

inductive sum (A : Type u) (B : Type v)
| inl {} : A → sum
| inr {} : B → sum
```

Notice that we do not include the types `A` and `B` in the target of the constructors. For the moment, ignore the annotation `{}` after the constructors `inl` and `inr`; we will explain that below. In the meanwhile, think about 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 A B`, 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 `A × B` for `prod A B` and `(a, b)` for `prod.mk a b`.

```
definition fst {A : Type u} {B : Type v} (p : A × B) : A :=
prod.rec_on p (λ a b, a)

definition snd {A : Type u} {B : Type v} (p : A × B) : B :=
prod.rec_on p (λ a b, b)
```

The function `fst` takes a pair, `p`. Applying the recursor `prod.rec_on p (fun a b, a)` interprets `p` as a pair, `prod.mk a b`, and then uses the second argument to determine what to do with `a` and `b`. Remember that you can enter the symbol for a product by typing `\times`. Recall also from Section 2.8 that to give these definitions the greatest generality possible, we allow the types `A` and `B` to belong to any universe.

Here is another example:

```
definition prod_example (p : bool × ℕ) : ℕ :=
prod.rec_on p (λ b n, cond b (2 * n) (2 * n + 1))

eval prod_example (tt, 3)
eval prod_example (ff, 3)
```

The `cond` function is a boolean conditional: `cond b t1 t2` return `t1` if `b` is true, and `t2` otherwise. (It has the same effect as `bool.rec_on b t2 t1`.) 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 A B`, 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`.

```
definition sum_example (s : ℕ ⊕ ℕ) : ℕ :=
sum.cases_on s (λ n, 2 * n) (λ 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`. You can enter the symbol for the sum by typing `\oplus`.

In the section after next 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 this is not the case: each constructor relies only on previously specified types.

Notice that a type with multiple constructors is disjunctive: an element of `sum A B` 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 A B` 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.

A type, like `prod`, with 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, these two words are synonymous, and provide alternative syntax for inductive types with a single constructor.

```
structure prod (A B : Type) :=
mk :: (fst : A) (snd : B)
```

The `structure` command simultaneously introduces the inductive type, `prod`, its constructor, `mk`, the usual eliminators (`rec`, `rec_on`), 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:

```
record color := (red : nat) (green : nat) (blue : nat)
definition 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. 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:

```
universe variable u

structure Semigroup :=
(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 a later chapter.

Notice that the product type depends on parameters `A B : 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, and makes them implicit in that case. Sometimes an argument can only be inferred from the return type, which means that it could not be inferred by parsing the expression from bottom up, but may be inferrable from context. In that case, Lean does not make the argument implicit by default, but will do so if we add the annotation `{}` after the constructor. We used that option, for example, in the definition of `sum`:

```
inductive sum (A : Type u) (B : Type v)
| inl {} : A → sum
| inr {} : B → sum
```

As a result, the argument `A` to `inl` and the argument `B` to `inr` are left implicit.

We have already discussed sigma types, also known as the dependent product:

```
inductive sigma {A : Type u} (B : A → Type v)
| dpair : Π a : A, B a → sigma
```

Two more examples of inductive types in the library are the following:

```
inductive option (A : Type u)
| none {} : option
| some    : A → option

inductive inhabited (A : Type u)
| mk : A → inhabited
```

In the semantics of dependent type theory, there is no built-in notion of a partial function. Every element of a function type `A → B` or a Pi type `Π x : A, B` is assumed to have a value at every input. The `option` type provides a way of representing partial functions. An element of `option B` is either `none` or of the form `some b`, for some value `b : B`. Thus we can think of an element `f` of the type `A → option B` as being a partial function from `A` to `B`: for every `a : A`, `f a` either returns `none`, indicating the `f a` is "undefined", or `some b`.

An element of `inhabited A` is simply a witness to the fact that there is an element of `A`. 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 `A` to `B` and `B` to `C`, 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.

6.3 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
| intro : true

inductive and (a b : Prop) : Prop
| intro : a → b → and

inductive or (a b : Prop) : Prop
| intro_left  : a → or
| intro_right : b → or
```

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 `p : P` carries no data. There is a small exception to this rule, however, which we will discuss below, in the section on inductive families.

Even the existential quantifier is inductively defined:

```
inductive Exists {A : Type u} (p : A → Prop) : Prop
| intro : ∀ (a : A), p a → Exists

definition exists.intro := @Exists.intro
```

Keep in mind that the notation `∃ x : A, p` is syntactic sugar for `Exists (λ x : A, 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 i` for `i` greater than 0. In a similar way, `∃ x : A, p` is a `Prop`-valued variant of `Σ x : A, p`.

This is a good place to mention another inductive type, denoted `{x : A | p}`, which is sort of a hybrid between `∃ x : A, P` and `Σ x : A, P`.

```
inductive subtype {A : Type u} (p : A → Prop)
| tag : Π x : A, p x → subtype
```

The notation `{x : A | p}` is syntactic sugar for `subtype (λ x : A, p)`. It is modeled after subset notation in set theory: the idea is that `{x : A | p}` denotes the collection of elements of `A` that have property `[`.

6.4 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 : Type
| 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 `Π n : nat, C n` for some `C : nat → Type`. 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_on
```

we find the following:

```
Π {C : nat → Type} (n : nat),
  C nat.zero → (Π (a : nat), C a → C (nat.succ a)) → C n
```

The implicit argument, `C`, is the codomain of the function being defined. In type theory it is common to say `C` is the `motive` for the elimination/recursion. The next argument, `n : nat`, is the input to the function. It is also known as the `major premise`. Finally, the two arguments after specify how to compute the zero and successor cases, as described above. They are also known as the `minor premises`.

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)`.

```
namespace nat

definition add (m n : nat) : nat :=
nat.rec_on n m (λ n add_m_n, succ add_m_n)

-- try it out
eval add (succ zero) (succ (succ zero))

end nat
```

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 hide

inductive nat : Type
| zero : nat
| succ : nat → nat

namespace nat

definition add (m n : nat) : nat :=
nat.rec_on n m (fun n add_m_n, succ add_m_n)
-- BEGIN]
instance : has_zero nat := has_zero.mk zero
instance : has_add nat := has_add.mk add

theorem add_zero (m : nat) : m + 0 = m := rfl
theorem add_succ (m n : nat) : m + succ n = succ (m + n) := rfl
-- END
end nat

end hide
```

We will explain how the `instance` command works in a later chapter. In the examples below, we will henceforth use Lean's version of the natural numbers.

Proving a fact like `0 + 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 `C n` is an element of `Prop`. It represents the familiar pattern of an inductive proof: to prove `∀ n, C n`, first prove `C 0`, and then, for arbitrary `n`, assume `ih : C n` and prove `C (succ n)`.

```
theorem zero_add (n : ℕ) : 0 + n = n :=
nat.induction_on n
  (show 0 + 0 = 0, from rfl)
  (take n,
    assume ih : 0 + n = n,
    show 0 + succ n = succ n, from
      calc
        0 + succ n = succ (0 + n) : rfl
          ... = succ n : by rewrite ih)
```

In the example above, we encourage you to replace `induction_on` with `rec_on` and observe that the theorem is still accepted by Lean. As we have seen above, `induction_on` is just a special case of `rec_on`.

For 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:

```
theorem add_assoc (m n k : ℕ) : m + n + k = m + (n + k) :=
nat.induction_on k
  (show m + n + 0 = m + (n + 0), from rfl)
  (take k,
    assume 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 rewrite ih
          ... = m + succ (n + k) : rfl
          ... = m + (n + succ k) : rfl)
```

For another example, suppose we try to prove the commutativity of addition. Choosing induction on the second argument, we might begin as follows:

```
theorem add_comm (m n : nat) : m + n = n + m :=
nat.induction_on n
  (show m + 0 = 0 + m, by rewrite nat.zero_add)
  (take n,
    assume ih : m + n = n + m,
    calc
      m + succ n = succ (m + n) : rfl
        ... = succ (n + m) : by rewrite ih
        ... = succ n + m : sorry)
```

At this point, we see that we need another supporting fact, namely, that `succ (n + m) = succ n + m`. We can prove this by induction on `m`:

```
theorem succ_add (m n : nat) : succ m + n = succ (m + n) :=
nat.induction_on n
  (show succ m + 0 = succ (m + 0), from rfl)
  (take n,
    assume ih : succ m + n = succ (m + n),
    show succ m + succ n = succ (m + succ n), from
      calc
        succ m + succ n = succ (succ m + n) : rfl
          ... = succ (succ (m + n)) : by rewrite ih
          ... = succ (m + succ n) : rfl)
```

We can then replace the `sorry` in the previous proof with `succ_add`.

As an exercise, 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.

6.5 Other Inductive Types

Let us consider some more examples of inductively defined types. For any type, `A`, the type `list A` of lists of elements of `A` is defined in the library.

```
inductive list (A : Type u)
| nil {} : list
| cons : A → list → list

namespace list

variable {A : Type}

notation h :: t  := cons h t

definition append (s t : list A) : list A :=
list.rec t (λ x l u, x::u) s

notation s ++ t := append s t

theorem nil_append (t : list A) : nil ++ t = t := rfl

theorem cons_append (x : A) (s t : list A) : x::s ++ t = x::(s ++ t) := rfl

end list
```

A list of elements of type `A` is either the empty list, `nil`, or an element `h : A` followed by a list `t : list A`. We define the notation `h :: t` to represent the latter. The first element, `h`, is commonly known as the "head" of the list, and the remainder, `t`, is known as

the "tail." Recall that the notation `{}` in the definition of the inductive type ensures that the argument to `nil` is implicit. In most cases, it can be inferred from context. When it cannot, we have to write `@nil A` to specify the type `A`.

Lean allows us to define iterative notation for lists:

```
inductive list (A : Type u)
| nil {} : list
| cons : A → list → list

namespace list

notation `[` l:(foldr `,` (h t, cons h t) nil) `]` := l

section
  open nat
  check [1, 2, 3, 4, 5]
  check ([1, 2, 3, 4, 5] : list num)
end

end list
```

In the first `check`, Lean assumes that `[1, 2, 3, 4, 5]` is a list of natural numbers. The `(t : list num)` expression forces Lean to interpret `t` as a list of numerals.

As an exercise, prove the following:

```
theorem append_nil (t : list A) : t ++ nil = t := sorry

theorem append_assoc (r s t : list A) : r ++ s ++ t = r ++ (s ++ t) := sorry
```

Try also defining the function `length : Π A : Type, list A → nat` that returns the length of a list, and prove that it behaves as expected (for example, `length (s ++ t) = length s + length t`).

For another example, we can define the type of binary trees:

```
inductive binary_tree
| leaf : binary_tree
| node : binary_tree → binary_tree → binary_tree
```

In fact, we can even define the type of countably branching trees:

```
inductive cbtree
| leaf : cbtree
| sup : (ℕ → cbtree) → cbtree

namespace cbtree

definition succ (t : cbtree) : cbtree :=
sup (λ n, t)
```

```
definition omega : cbtree :=
sup (λ n, nat.rec_on n leaf (λ n t, succ t))

end cbtree
```

7

Induction and Recursion

Bibliography

[1] Thierry Coquand and Gerard Huet. The calculus of constructions. *Inf. Comput.*, 76(2-3):95–120, February 1988.

[2] Peter Dybjer. Inductive families. *Formal Asp. Comput.*, 6(4):440–465, 1994.

[3] Frank Pfenning and Christine Paulin-Mohring. Inductively defined types in the calculus of constructions. In Michael G. Main, Austin Melton, Michael W. Mislove, and David A. Schmidt, editors, *Mathematical Foundations of Programming Semantics, 5th International Conference, Tulane University, New Orleans, Louisiana, USA, March 29 - April 1, 1989, Proceedings*, volume 442 of *Lecture Notes in Computer Science*, pages 209–228. Springer, 1989.