Theorem Proving in Lean

Release 3.23.0

Jeremy Avigad, Leonardo de Moura, and Soonho Kong

Jan 21, 2021

CONTENTS

CHAPTER

ONE

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.

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

1.2 About Lean

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

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. Special modes in Visual Studio Code (VS Code for short) and Emacs offer powerful support for writing and debugging proofs, and is much better suited for serious use. The source code, and instructions for building Lean, are available at https://github.com/leanprover/lean/.

This tutorial describes the current version of Lean, known as Lean 3. A prior version, Lean 2, had special support for homotopy type theory. You can find Lean 2 and the HoTT library at https://github.com/leanprover/lean2/. The tutorial for that version of Lean is at https://leanprover.github.io/tutorial/.

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 version of a system known as the *Calculus of Constructions* [CoHu88] with *inductive types* [Dybj94]. 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.

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.left hpq,
have hq : q, from and.right hpq,
show q ∧ p, from and.intro hq hp
```

If you are reading the book online, you will see a button that reads "try it!" Pressing the button opens up a tab with a Lean editor, and copies the example with enough surrounding context to make the example compile correctly. You can type things into the editor and modify the examples, and Lean will check the results and provide feedback continuously as you type. We recommend running the examples and experimenting with the code on your own as you work through the chapters that follow.

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, Kevin Buzzard, Mario Carneiro, Nathan Carter, Eduardo Cavazos, Amine Chaieb, Joe Corneli, William DeMeo, Marcus Klaas de Vries, Ben Dyer, Gabriel Ebner, Anthony Hart, Simon Hudon, Sean Leather, Assia Mahboubi, Gihan Marasingha, Patrick Massot, Christopher John Mazey, Sebastian Ullrich, Floris van Doorn, Daniel Velleman, and Théo Zimmerman for their contributions, and we apologize to those whose names we have inadvertently omitted.

CHAPTER

TWO

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 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.
```

Any text between the `/-` and `-/` constitutes a comment 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. The `#check` command asks Lean to report their types; in Lean, commands that query the system for information typically begin with the hash symbol. You should try declaring some constants and type checking some expressions on 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. 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 α and β are types, $\alpha \to \beta$ denotes the type of functions from α to β, and $\alpha \times \beta$ denotes the cartesian product, that is, the type of ordered pairs consisting of an element of α paired with an element of β.

```
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             -- 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
```

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

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 `\times`. We will generally use lower-case greek letters like α, β, and γ to range over types. You can enter these particular ones with `\a`, `\b`, and `\g`.

There are a few more things to notice here. First, the application of a function `f` to a value `x` is denoted `f x`. 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, `p.1` and `p.2` 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 α β : Type
constant F : Type → Type
constant G : Type → Type → Type

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

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

```
constants α β : Type

#check prod α β       -- Type
#check prod nat nat   -- Type
```

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

```
constant α : Type

#check list α     -- 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       -- Type 1
```

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

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

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

There is also another type, `Prop`, which has special properties.

```
#check Prop -- Type
```

We will discuss `Prop` in the next chapter.

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

```
#check list    -- Type u_1 → Type u_1
```

Here `u_1` is a variable ranging over type levels. The output of the `#check` command means that whenever α has type `Type n`, `list α` also has type `Type n`. The function `prod` is similarly polymorphic:

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

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

```
universe u
constant α : Type u
#check α
```

Equivalently, we can write `Type _` or `Type*` to avoid giving the arbitrary universe a name:

```
constant α : Type _
#check α

constant β : Type*
#check β
```

Throughout this book, we will generally use `Type*` in examples when we want type constructions to have as much generality as possible. We will come to learn 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 : α → β`, namely, we can apply it to an element `a : α` to obtain `f a : β`. 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 : α` we can construct an expression `t : β`. Then the expression `fun x : α, t`, or, equivalently, `λ x : α, t`, is an object of type `α → β`. Think of this as the function from α to β which maps any value `x` to the value `t`, 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 α β  : Type
constants a1 a2 : α
constants b1 b2 : β

constant f : α → α
constant g : α → β
constant h : α → β → α
constant p : α → α → bool

#check fun x : α, f x                      -- α → α
#check λ x : α, f x                        -- α → α
#check λ x : α, f (f x)                    -- α → α
#check λ x : α, h x b1                     -- α → α
#check λ y : β, h a1 y                     -- β → α
#check λ x : α, p (f (f x)) (h (f a1) b2)  -- α → bool
#check λ x : α, λ y : β, h (f x) y         -- α → β → α
#check λ (x : α) (y : β), h (f x) y        -- α → β → α
#check λ x y, h (f x) y                    -- α → β → α
```

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`.

Try writing some expressions on your own. Some mathematically common examples of operations of functions can be described in terms of lambda abstraction:

```
constants α β γ : Type
constant f : α → β
constant g : β → γ
constant b : β

#check λ x : α, x          -- α → α
#check λ x : α, b          -- α → β
#check λ x : α, g (f x)    -- α → γ
#check λ x, g (f x)
```

Think about what these expressions mean. The expression `λ x : α, x` denotes the identity function on α, the expression `λ x : α, b` denotes the constant function that always returns `b`, and `λ x : α, 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 : α, g (f x)`.

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

```
#check λ b : β, λ x : α, x      -- β → α → α
#check λ (b : β) (x : α), x     -- β → α → α
#check λ (g : β → γ) (f : α → β) (x : α), g (f x)
                              -- (β → γ) → (α → β) → α → γ
```

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

```
#check λ (α β : Type*) (b : β) (x : α), x
#check λ (α β γ : Type*) (g : β → γ) (f : α → β) (x : α), g (f x)
```

The last expression, for example, denotes the function that takes three types, α, β, and γ, and two functions, `g : β → γ` and `f : α → β`, and returns the composition of `g` and `f`. (Making sense of the type of this function requires an understanding of dependent products, which we will explain below.) Within a lambda expression `λ x : α, 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 : β) (x : α), x` has nothing to do with the constant `b` declared earlier. In fact, the expression denotes the same function as `λ (u : β) (z : α), 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 : α → β` to a term `s : α` yields an expression `t s : β`. Returning to the previous example and renaming bound variables for clarity, notice the types of the following expressions:

```
constants α β γ : Type
constant f : α → β
constant g : β → γ
constant h : α → α
constants (a : α) (b : β)

#check (λ x : α, x) a                -- α
#check (λ x : α, b) a                -- β
#check (λ x : α, b) (h a)            -- β
#check (λ x : α, g (f x)) (h (h a))  -- γ

#check (λ (v : β → γ) (u : α → β) x, v (u x)) g f a   -- γ

#check (λ (Q R S : Type*) (v : R → S) (u : Q → R) (x : Q),
        v (u x)) α β γ g f a          -- γ
```

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

```
constants α β γ : Type
constant f : α → β
constant g : β → γ
constant h : α → α
constants (a : α) (b : β)

#reduce (λ x : α, x) a                -- a
#reduce (λ x : α, b) a                -- b
#reduce (λ x : α, b) (h a)            -- b
#reduce (λ x : α, g (f x)) a          -- g (f a)

#reduce (λ (v : β → γ) (u : α → β) x, v (u x)) g f a   -- g (f a)
```

(continues on next page)

(continued from previous page)

```
#reduce (λ (Q R S : Type*) (v : R → S) (u : Q → R) (x : Q),
       v (u x)) α β γ g f a        -- g (f a)
```

The command `#reduce` tells Lean to evaluate an expression by *reducing* it to normal form, which is to say, carrying out all the computational reductions that are sanctioned by the underlying logic. 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 `#reduce` command carries out other forms of reduction as well:

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

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

#print "reducing boolean expressions"
#reduce tt && ff        -- ff
#reduce ff && b         -- ff
#reduce b && ff         -- bool.rec ff ff b

#print "reducing arithmetic expressions"
#reduce n + 0           -- n
#reduce n + 2           -- nat.succ (nat.succ n)
#reduce 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.

It is this computational behavior that makes it possible to use Lean as a programming language as well. Indeed, Lean extracts bytecode from terms in a computationally pure fragment of the logical framework, and can evaluate them quite efficiently:

```
#eval 12345 * 54321
```

In contrast, the `#reduce` command relies on Lean's trusted kernel, the part of Lean that is responsible for checking and verifying the correctness of expressions and proofs. As such, the `#reduce` command is more trustworthy, but far less efficient. We will have more to say about `#eval` in Chapter 11, and it will play a central role in Programming in Lean. In this tutorial, however, we will generally rely on `#reduce` instead.

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 `def` command provides one important way of defining new objects.

```
def foo : (ℕ → ℕ) → ℕ := λ f, f 0

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

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

```
def foo' := λ f : ℕ → ℕ, f 0
```

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

Lean also allows us to use an alternative format that puts the abstracted variables before the colon and omits the lambda:

```
def double (x : ℕ) : ℕ := x + x
#print double
#check double 3
#reduce double 3    -- 6

def square (x : ℕ) := x * x
#print square
#check square 3
#reduce square 3    -- 9

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

#reduce do_twice double 2    -- 8
```

These definitions are equivalent to the following:

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

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

```
def compose (α β γ : Type*) (g : β → γ) (f : α → β) (x : α) :
  γ :=
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 applies *its* argument twice, so that `Do_Twice do_twice` is a function that applies its input four times. Then 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.

```
def curry (α β γ : Type*) (f : α × β → γ) : α → β → γ := sorry

def uncurry (α β γ : Type*) (f : α → β → γ) : α × β → γ := 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
#reduce  let y := 2 + 2 in y * y   -- 16

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

#reduce 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   -- ℕ
#reduce  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.

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

/-
def 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 Bertrand 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 α, β, and γ to populate our context. This can be avoided, using implicit or explicit lambda abstraction in our definitions to declare such objects "locally":

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

def do_twice (α : Type*) (h : α → α) (x : α) : α := h (h x)

def do_thrice (α : Type*) (h : α → α) (x : α) : α := 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 (α β γ : Type*)

def compose (g : β → γ) (f : α → β) (x : α) : γ := g (f x)
def do_twice (h : α → α) (x : α) : α := h (h x)
def do_thrice (h : α → α) (x : α) : α := h (h (h x))
```

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

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

def compose := g (f x)
def do_twice := h (h x)
def 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 former commands simply instruct Lean to insert the declared 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 α, β, γ, `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 (α β γ : Type*)
variables (g : β → γ) (f : α → β) (h : α → α)
variable x : α

def compose := g (f x)
def do_twice := h (h x)
def 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 string 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.

We will see in Chapter 6 that, as a scoping mechanism, sections govern more than just variables; other commands have effects that are only operant in the current section. Similarly, if we use the `open` command inside a section, it only remains in effect until that section is closed.

2.7 Namespaces

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

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

def fa : ℕ := f a
def 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. But sometimes we will want to leave this information hidden, for example, when they conflict with identifiers 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
```

(continues on next page)

(continued from previous page)

```
#check nil
#check cons
#check append
```

Like sections, namespaces can be nested:

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

def fa : ℕ := f a

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

def fa : ℕ := f a
end foo

#check foo.a
#check foo.f

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

2.8 Dependent Types

You have now seen one way of defining functions and objects in Lean, and we will gradually introduce you to many more. But an important 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 this section, we will explain what makes dependent type theory *dependent*, and why dependent types are 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 α` depends on the argument α, and this dependence is what distinguishes `list ℕ` and `list bool`. For another example, consider the type `vec α n`, the type of vectors of elements of α of length `n`. This type depends on *two* parameters: the type `α : 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 α to behave the same way. So it makes sense to take the type to be the first argument to `cons`, so that for any type, α, `cons α` is the insertion function for lists of type α. In other words, for every α, `cons α` is the function that takes an element `a : α` and a list `l : list α`, and returns a new list, so we have `cons α a l : list α`.

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

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

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

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

```
namespace hidden

universe u

constant list   : Type u → Type u

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

end hidden
```

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. Moreover, 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 {α : Type u_1}, α → list α → list α
#check @nil      -- Π {α : Type u_1}, list α
#check @head     -- Π {α : Type u_1} [_inst_1 : inhabited α], list α → α
#check @tail     -- Π {α : Type u_1}, list α → list α
#check @append   -- Π {α : Type u_1}, list α → list α → list α
```

There is a subtlety in the definition of `head`: the type α 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 Chapter 10.

Vector operations are handled similarly:

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

namespace vec
constant empty : Π α : Type u, vec α 0
constant cons :
  Π (α : Type u) (n : ℕ), α → vec α n → vec α (n + 1)
constant append :
  Π (α : Type u) (n m : ℕ),  vec α m → vec α n → vec α (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 : α, β x`, sometimes also known as *dependent products*. These are, in a sense, companions to the Pi types. The type `Σ x : α, β x` denotes the type of pairs `sigma.mk a b` where `a : α` and `b : β a`.

Just as Pi types `Π x : α, β x` generalize the notion of a function type `α → β` by allowing β to depend on α, Sigma types `Σ x : α, β x` generalize the cartesian product `α × β` in the same way: in the expression `sigma.mk a b`, the type of the second element of the pair, `b : β a`, depends on the first element of the pair, `a : α`.

```
variable α : Type
variable β : α → Type
variable a : α
variable b : β a

#check sigma.mk a b      -- Σ (a : α), β a
#check (sigma.mk a b).1  -- α
#check (sigma.mk a b).2  -- β (sigma.fst (sigma.mk a b))

#reduce  (sigma.mk a b).1  -- a
#reduce  (sigma.mk a b).2  -- b
```

Notice that the expressions `(sigma.mk a b).1` and `(sigma.mk a b).2` are short for `sigma.fst (sigma.mk a b)` and `sigma.snd (sigma.mk 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 hidden
universe u
constant list : Type u → Type u

namespace list
constant cons   : Π α : Type u, α → list α → list α
constant nil    : Π α : Type u, list α
constant append : Π α : Type u, list α → list α → list α
end list
end hidden
```

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

```
open hidden.list

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

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

Because the constructors are polymorphic over types, we have to insert the type α as an argument repeatedly. But this information is redundant: one can infer the argument α in `cons α a (nil α)` from the fact that the second argument, `a`, has type α. One can similarly infer the argument in `nil α`, 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 α` in that position.

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

```
#check 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 {α : Type u}, α → list α → list α
constant nil    : Π {α : Type u}, list α
constant append : Π {α : Type u}, list α → list α → list α
end list

open hidden.list

variable  α : Type
variable  a : α
variables l1 l2 : list α
```

(continues on next page)

(continued from previous page)

```
#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 `α : Type u` in the declaration of the variables. We can also use this device in function definitions:

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

variables α β : Type u
variables (a : α) (b : β)

#check ident      -- ?M_1 → ?M_1
#check ident a    -- α
#check ident b    -- β
```

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

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

```
universe u

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

variables α β : Type u
variables (a : α) (b : β)

#check ident
#check ident a
#check ident b
```

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

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

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

```
#check list.nil              -- list ?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 below are elaborated in the same way, whereas the third `#check` command interprets `2` as an integer.

```
#check 2            -- ℕ
#check (2 : ℕ)      -- ℕ
#check (2 : ℤ)      -- ℤ
```

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 {α : Type u_1}, α → α
#check @id α        -- α → α
#check @id β        -- β → β
#check @id α a      -- α
#check @id β b      -- β
```

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.

2.10 Exercises

1. Define the function `Do_Twice`, as described in Section 2.4.
2. Define the functions `curry` and `uncurry`, as described in Section 2.4.
3. Above, we used the example `vec α n` for vectors of elements of type α of length `n`. Declare a constant `vec_add` that could represent a function that adds two vectors of natural numbers of the same length, and a constant `vec_reverse` that can represent a function that reverses its argument. Use implicit arguments for parameters that can be inferred. Declare some variables and check some expressions involving the constants that you have declared.
4. Similarly, declare a constant `matrix` so that `matrix α m n` could represent the type of `m` by `n` matrices. Declare some constants to represent functions on this type, such as matrix addition and multiplication, and (using `vec`) multiplication of a matrix by a vector. Once again, declare some variables and check some expressions involving the constants that you have declared.

CHAPTER

THREE

PROPOSITIONS AND PROOFS

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

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 introduce 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 a 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 :
  Π 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 :
  Π p q : Prop, (Proof p → Proof q) → Proof (implies p q).
```

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

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

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

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

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

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

Indeed, if `p : Prop` is any proposition, Lean's kernel treats any two elements `t1 t2 : p` as being definitionally equal, much the same way as it treats `(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 the bottom line. To formally express a mathematical assertion in the language of dependent type theory, we need to exhibit a term `p : Prop`. To *prove* that assertion, we need to exhibit a term `t : p`. Lean's task, as a proof assistant, is to help us to construct such a term, `t`, and to verify that it is well-formed and has the correct type.

3.2 Working with Propositions as Types

In the propositions-as-types paradigm, theorems involving only $\to$ 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. In fact, Lean is generally able to process and check proofs in parallel, since assessing the correctness of one proof does not require knowing the details of another.

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

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

#print t1
```

Notice that the lambda abstractions `hp : p` and `hq : q` can be viewed as temporary assumptions in the proof of `t1`. Lean 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 `lemma` instead of theorem:

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

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

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

#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 example, we can move all parameters to the right of the colon:

```
theorem t1 : ∀ (p q : Prop), p → q → p :=
λ (p q : Prop) (hp : p) (hq : q), hp
```

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

```
variables p q : Prop

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

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

```
variables p q : Prop
variable  hp : p

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

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

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

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

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
```

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

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

```
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?

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

3.3 Propositional Logic

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

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

They all take values in `Prop`.

```
variables p q : Prop

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

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

In the last chapter we observed that lambda abstraction can be viewed as an "introduction rule" for →. In the current setting, it shows how to "introduce" or establish an implication. Application can be viewed as an "elimination rule," showing how to "eliminate" or use an implication in a proof. The other propositional connectives are defined in Lean's library in the file `init.core` (see Section 6.1 for more information on the library hierarchy), and each connective comes with its canonical introduction and elimination rules.

3.3.1 Conjunction

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

```
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 Chapter 9 that certain types in Lean are *structures*, which is to say, the type is defined with a single canonical *constructor* which builds an element of the type from a sequence of suitable arguments. For every `p q : Prop`, `p ∧ q` is an example: the canonical way to construct an element is to apply `and.intro` to suitable arguments `hp : p` and `hq : q`. Lean allows us to use *anonymous constructor* notation `⟨arg1, arg2, ...⟩` in situations like these, when the relevant type is an inductive type and can be inferred from the context. In particular, we can often write `⟨hp, hq⟩` instead of `and.intro hp hq`:

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

#check (⟨hp, hq⟩ : p ∧ q)
```

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

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

```
variable l : list ℕ

#check list.head l
#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 are salient, the notation is clean and effective.

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

```
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.

3.3.2 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 shorthand 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.

3.3.3 Negation and Falsity

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

```
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`.

3.3.4 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))
```

(continues on next page)

(continued from previous page)

```
  (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`:

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

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

```
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:

```
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
```

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.

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

```
variables p q : Prop

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

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

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

```
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 without relying on excluded middle.

The full list of axioms that are used in Lean to support classical reasoning are discussed in Chapter 11.

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. The following list includes a number of common identities.

Commutativity:

1. `p ∧ q ↔ q ∧ p`
2. `p ∨ q ↔ q ∨ p`

Associativity:

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

Distributivity:

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

Other properties:

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

These require classical reasoning:

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

25. `(((p → q) → p) → p)`

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

Here is another useful trick. Instead of using `sorry`, you can use an underscore _ as a placeholder. Recall 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)
```

3.7 Exercises

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

```
variables p q r : Prop

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

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

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

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

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

```
open classical

variables p q r s : Prop

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
```

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

CHAPTER

FOUR

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

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

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

The elimination rule states:

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

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

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

The elimination rule states:

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

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

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

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

```
variables (α : Type*) (p q : α → Prop)

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

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

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

```
example : (∀ x : α, p x ∧ q x) → ∀ x : α, p x  :=
assume h : ∀ x : α, p x ∧ q x,
assume z : α,
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 (α : Type*) (r : α → α → Prop)
variable  trans_r : ∀ x y z, r x y → r y z → r x z

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

#check trans_r    -- ∀ (x y z : α), r x y → r y z → r x z
#check trans_r a b c
#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 (α : Type*) (r : α → α → Prop)
variable  trans_r : ∀ {x y z}, r x y → r y z → r x z

variables (a b c : α)
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 first `#check` command is `r ?M_1 ?M_2 → r ?M_2 ?M_3 → r ?M_1 ?M_3`, indicating that the implicit arguments are unspecified in this case.

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

```
variables (α : Type*) (r : α → α → Prop)

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

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

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

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

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

The term "predicative" stems from foundational developments around the turn of the twentieth century, when logicians such as Poincaré and Russell blamed set-theoretic paradoxes on the "vicious circles" that arise when we define a property by quantifying over a collection that includes the very property being defined. Notice that if α is any type, we can form the type `α → Prop` of all predicates on α (the "power type of α"). The impredicativity of Prop means that we can form propositions that quantify over `α → Prop`. In particular, we can define predicates on α by quantifying over all predicates on α, which is exactly the type of circularity that was once considered problematic.

4.2 Equality

Let us now turn to one of the most fundamental relations defined in Lean's library, namely, the equality relation. In Chapter 7, 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
```

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

```
universe u

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

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

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

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

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

We can also use the projection notation:

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

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

```
variables (α β : Type*)

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

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

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

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

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

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

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

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

```
variable α : Type
variables a b : α
variables f g : α → ℕ
variable h₁ : a = b
variable h₂ : f = g

example : f a = f b := congr_arg f h₁
example : f a = g a := congr_fun h₂ a
example : f a = g b := congr h₂ h₁
```

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

```
import data.int.basic

variables a b c d : ℤ

example : a + 0 = a := add_zero a
example : 0 + a = a := zero_add a
example : a * 1 = a := mul_one a
example : 1 * a = a := one_mul a
example : -a + a = 0 := neg_add_self a
example : a + -a = 0 := add_neg_self a
example : a - a = 0 := sub_self a
example : a + b = b + a := add_comm a b
example : a + b + c = a + (b + c) := add_assoc a b c
example : a * b = b * a := mul_comm a b
example : a * b * c = a * (b * c) := mul_assoc a b c
example : a * (b + c) = a * b + a * c := mul_add a b c
example : a * (b + c) = a * b + a * c := left_distrib a b c
example : (a + b) * c = a * c + b * c := add_mul a b c
example : (a + b) * c = a * c + b * c := right_distrib a b c
example : a * (b - c) = a * b - a * c := mul_sub a b c
example : (a - b) * c = a * c - b * c := sub_mul a b c
```

Note that `mul_add` and `add_mul` are alternative names for `left_distrib` and `right_distrib`, respectively. The properties above are stated for the integers; the type ℤ can be entered as `\int`, though we can also use the ascii equivalent `int`. Identities like these are designed to work in arbitrary instances of the relevant algebraic structures, using the type class mechanism that is described in Chapter 10. In particular, all these facts hold in any commutative ring, of which Lean recognizes the integers to be an instance. Chapter 6 provides some pointers as to how to find theorems like this in the library.

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

```
variables x y z : ℤ

example (x y z : ℕ) : x * (y + z) = x * y + x * z := mul_add x y z
example (x y z : ℕ) : (x + y) * z = x * z + y * z := add_mul x y z
example (x y z : ℕ) : x + y + z = x + (y + z) := add_assoc x y z

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

Notice that the second implicit parameter to `eq.subst`, which provides the context in which the substitution is to occur, has type $\alpha \to$ `Prop`. Inferring this predicate therefore requires an instance of *higher-order unification*. In full generality, the problem of determining whether a higher-order unifier exists is undecidable, and Lean can at best provide imperfect and approximate solutions to the problem. As a result, `eq.subst` doesn't always do what you want it to. This issue is discussed in greater detail in Section 6.10.

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

4.3 Calculational Proofs

A calculational proof is just a chain of intermediate results that are meant to be composed by basic principles such as the transitivity of equality. In Lean, a 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`.

Here is an example:

```
import data.nat.basic

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

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

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

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

In the next chapter, we will see that hypotheses can be introduced, renamed, and modified by tactics, so it is not always clear what the names in `rw h1` refer to (though, in this case, it is). For that reason, section variables and variables that only appear in a tactic command or block are not automatically added to the context. The `include` command takes care of that. Essentially, the `rewrite` tactic uses a given equality (which can be a hypothesis, a theorem name, or a complex term) to "rewrite" the goal. If doing so reduces the goal to an identity `t = t`, the tactic applies reflexivity to prove it.

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

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

Or even this:

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

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

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

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

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

```
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.lt_succ_self b
    ... ≤ c + 1 : nat.succ_le_succ h2
    ... < d     : h3
```

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

```
example (x y : ℕ) :
  (x + y) * (x + y) = x * x + y * x + x * y + y * y :=
calc
  (x + y) * (x + y) = (x + y) * x + (x + y) * y  : by rw mul_add
    ... = x * x + y * x + (x + y) * y            : by rw add_mul
    ... = x * x + y * x + (x * y + y * y)        : by rw add_mul
    ... = x * x + y * x + x * y + y * y          : by rw ←add_assoc
```

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

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

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

4.4 The Existential Quantifier

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

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

```
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 : α → 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
#print gex1
#print gex2
#print gex3
#print gex4
```

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

```
variables (α : Type*) (p q : α → Prop)
```

(continues on next page)

(continued from previous page)

```
example (h : ∃ x, p x ∧ q x) : ∃ x, q x ∧ p x :=
exists.elim h
  (assume 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 : α, p x` can be thought of as a big disjunction of the propositions `p a`, as `a` ranges over all the elements of α. Note that the anonymous constructor notation `⟨w, hw.right, hw.left⟩` abbreviates a nested constructor application; we could equally well have written `⟨w, ⟨hw.right, hw.left⟩⟩`.

Notice that an existential proposition is very similar to a sigma type, as described in Section 2.8. The difference is that given `a : α` and `h : p a`, the term `exists.intro a h` has type `(∃ x : α, p x) : Prop` and `sigma.mk a h` has type `(Σ x : α, p x) : Type`. The similarity between $\exists$ 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 (α : Type*) (p q : α → Prop)

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

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

```
example (h : ∃ x, p x ∧ q x) : ∃ x, q x ∧ p x :=
match h with ⟨(w : α), (hw : p w ∧ q w)⟩ :=
  ⟨w, hw.right, hw.left⟩
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 also provides a pattern-matching `let` expression:

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

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

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

We will see in Chapter 8 that all these variations are instances of a more general pattern-matching construct.

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

```
import data.nat.basic

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

theorem even_plus_even {a b : nat}
  (h1 : is_even a) (h2 : is_even b) : is_even (a + b) :=
exists.elim h1 (assume w1, assume hw1 : a = 2 * w1,
exists.elim h2 (assume w2, assume hw2 : b = 2 * w2,
  exists.intro (w1 + w2)
    (calc
      a + b = 2 * w1 + 2 * w2  : by rw [hw1, hw2]
        ... = 2*(w1 + w2)      : by rw mul_add)))
```

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

```
theorem even_plus_even {a b : nat}
  (h1 : is_even a) (h2 : is_even b) : is_even (a + b) :=
match h1, h2 with
  ⟨w1, hw1⟩, ⟨w2, hw2⟩ := ⟨w1 + w2, by rw [hw1, hw2, mul_add]⟩
end
```

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 (α : Type*) (p : α → Prop)

example (h : ¬ ∀ x, ¬ p x) : ∃ x, p x :=
by_contradiction
  (assume h1 : ¬ ∃ x, p x,
    have h2 : ∀ x, ¬ p x, from
      assume 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. In the exercises below, we encourage you to prove as many as you can. We also leave it to you to determine which are nonconstructive, and hence require some form of classical reasoning.

```
open classical

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

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

(continues on next page)

(continued from previous page)

```
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 : α` amounts to the assumption that there is at least one element of type α. 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
                assume 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`, `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
variable 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 le_trans this (h 1),
show f 0 ≤ f 3, from 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
variable 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 le_trans (by assumption) (h 1),
show f 0 ≤ f 3, from le_trans (by assumption) (h 2)
```

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

We can also ask Lean to fill in the proof by writing `‹p›`, where `p` is the proposition whose proof we want Lean to find in the context.

```
example : f 0 ≥ f 1 → f 1 ≥ f 2 → f 0 = f 2 :=
assume : f 0 ≥ f 1,
assume : f 1 ≥ f 2,
have f 0 ≥ f 2, from le_trans this ‹f 0 ≥ f 1›,
have f 0 ≤ f 2, from le_trans (h 0) (h 1),
show f 0 = f 2, from 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 le_trans ‹f 0 ≤ f 1›
                       (le_trans ‹f 1 ≤ f 2› ‹f 2 ≤ f 3›)
```

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

```
example (n : ℕ) : ℕ := ‹ℕ›
```

We can also `assume` a hypothesis without giving it a label:

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

In contrast to the usage with `have`, an anonymous `assume` needs an extra colon. The reason is that Lean allows us to write `assume h` to introduce a hypothesis without specifying it, and without the colon it would be ambiguous as to whether the `h` here is meant as the label or the assumption.

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

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

4.6 Exercises

1. Prove these equivalences:

```
variables (α : Type*) (p q : α → Prop)

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

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

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

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

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

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

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

```

(continues on next page)

(continued from previous page)

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

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

```
import data.nat.basic

#check even

def prime (n : ℕ) : Prop := sorry

def infinitely_many_primes : Prop := sorry

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

def infinitely_many_Fermat_primes : Prop := sorry

def goldbach_conjecture : Prop := sorry

def Goldbach's_weak_conjecture : Prop := sorry

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

5. Prove as many of the identities listed in Section 4.4 as you can.

6. Give a calculational proof of the theorem `log_mul` below.

```
import data.real.basic

variables log exp     : real → real
variable  log_exp_eq : ∀ x, log (exp x) = x
variable  exp_log_eq : ∀ {x}, x > 0 → exp (log x) = x
variable  exp_pos    : ∀ x, exp x > 0
variable  exp_add    : ∀ x y, exp (x + y) = exp x * exp y

-- this ensures the assumptions are available in tactic proofs
include log_exp_eq exp_log_eq exp_pos exp_add

example (x y z : real) :
  exp (x + y + z) = exp x * exp y * exp z :=
by rw [exp_add, exp_add]

example (y : real) (h : y > 0)  : exp (log y) = y :=
exp_log_eq h

theorem log_mul {x y : real} (hx : x > 0) (hy : y > 0) :
  log (x * y) = log x + log y :=
sorry
```

7. Prove the theorem below, using only the ring properties of ℤ enumerated in Section 4.2 and the theorem `sub_self`.

```
import data.int.basic

#check sub_self

example (x : ℤ) : x * 0 = 0 :=
sorry
```

CHAPTER

FIVE

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 example, tactic-style proofs can be harder to read, because they require the reader to predict or guess the results of each instruction. But they can also be shorter and easier to write. Moreover, tactics offer a gateway to using Lean's automation, since automated procedures are themselves tactics.

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. It is also more robust than apply, since the elaborator takes the expected type, given by the target of the goal, into account when processing the expression that is being applied. In this case, however, apply would work just as well.

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

```
#print test
```

You can write a tactic script incrementally. In VS Code, you can open a window to display messages by pressing Ctrl-Shift-Enter, and that window will then show you the current goal whenever the cursor is in a tactic block. In Emacs, you can see the goal at the end of any line by pressing C-c C-g, or see the remaining goal in an incomplete proof by putting the cursor on the end symbol.

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
```

Tactic applications can also be concatenated with a semicolon. Formally speaking, there is only one (compound) step in the following proof:

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

See Section 5.5 for a more precise description of the semantics of the semicolon. When a single tactic step can be used to dispell a goal, you can use the by keyword instead of using a begin...end block.

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

In VS Code, the tactic state will appear in the messages window whenever the cursor is within the contexts of the by. 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.

We will see below that hypotheses can be introduced, reverted, modified, and renamed over the course of a tactic block. As a result, it is impossible for the Lean parser to detect when an identifier that occurs in a tactic block refers to a section variable that should therefore be added to the context. As a result, you need to explicitly tell Lean to include the relevant entities:

```
variables {p q : Prop} (hp : p) (hq : q)

include hp hq

example : p ∧ q ∧ p :=
begin
  apply and.intro hp,
  exact and.intro hq hp
end
```

The `include` command tells Lean to include the indicated variables (as well as any variables they depend on) from that point on, until the end of the section or file. To limit the effect of an `include`, you can use the `omit` command afterwards:

```
include hp hq

example : p ∧ q ∧ p :=
begin
  apply and.intro hp,
  exact and.intro hq hp
end

omit hp hq
```

Thereafter, `hp` and `hq` are no longer included by default. Alternatively, you can use a section to delimit the scope.

```
section
include hp hq

example : p ∧ q ∧ p :=
begin
  apply and.intro hp,
  exact and.intro hq hp
end
end
```

Once again, thereafter, `hp` and `hq` are no longer included by default. Another workaround is to find a way to refer to the variable in question before entering a tactic block:

```
example : p ∧ q ∧ p :=
let hp := hp, hq := hq in
begin
  apply and.intro hp,
  exact and.intro hq hp
end
```

Any mention of `hp` or `hq` at all outside a tactic block will cause them to be added to the hypotheses.

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 3.6, now proved 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. That rationale behind this convention, and other structuring mechanisms, will be discussed in Section 5.4 below.

```
example (p q r : Prop) : p ∧ (q ∨ r) ↔ (p ∧ q) ∨ (p ∧ r) :=
begin
  apply iff.intro,
    intro h,
    apply or.elim (and.right h),
      intro hq,
      apply or.inl,
      apply and.intro,
        exact and.left h,
      exact hq,
    intro hr,
    apply or.inr,
    apply and.intro,
      exact and.left h,
    exact hr,
  intro h,
  apply or.elim h,
    intro hpq,
    apply and.intro,
      exact and.left hpq,
    apply or.inl,
    exact and.right hpq,
  intro hpr,
  apply and.intro,
    exact and.left hpr,
  apply or.inr,
  exact and.right hpr
end
```

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

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

example (α : Type*) : ∀ x : α, 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 : ℕ, a = b → a = c → c = b :=
begin
  intros a b c h₁ h₂,
```

(continues on next page)

(continued from previous page)

```
  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.

```
variables x y z w : ℕ

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 = ?m_1 with h₁
  apply eq.trans,
  assumption,     -- solves y = ?m_1 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 : ℕ, a = b → a = c → c = b :=
begin
  intros,
  apply eq.trans,
  apply eq.symm,
  assumption,
  assumption
end
```

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. (Attributes will be discussed in Section 6.4.) The `reflexivity` tactic can also be abbreviated as `refl`.

```
example (y : ℕ) : (λ x : ℕ, 0) y = 0 :=
begin
  refl
end

example (x : ℕ) : x ≤ x :=
begin
  refl
end
```

With these tactics, the transitivity proof above can be written more elegantly as follows:

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

In each case, the use of transitivity introduces a metavariable for the middle term, which is then determined by the later tactics. Alternatively, we can send this middle term as an optional argument to `transitivity`:

```
example : ∀ a b c : ℕ, a = b → a = c → c = b :=
begin
  intros a b c h₁ h₂,
  transitivity a,
  symmetry,
  assumption,
  assumption
end
```

The `repeat` combinator can be used to simplify the last two lines:

```
example : ∀ a b c : ℕ, a = b → a = c → c = b :=
begin
  intros,
  apply eq.trans,
  apply eq.symm,
  repeat { assumption }
end
```

The curly braces introduce a new tactic block; they are equivalent to using a nested `begin ... end` pair, as discussed in the next section.

If some of the goals that are needed to complete the result of an `apply` depend on others, the `apply` tactic places those subgoals last, in the hopes that they will be solved implicitly by the solutions to the previous subgoals. For example, consider the following proof:

```
example : ∃ a : ℕ, 5 = a :=
begin
  apply exists.intro,
  reflexivity
end
```

The first `apply` requires us to construct two values, namely, a value of `a` and a proof that `5 = a`. But the `apply` tactic takes the second goal to be the more important one, and places it first. Solving it with reflexivity forces `a` to be instantiated to `5`, at which point, the second goal is solved automatically.

Sometimes, however, we want to synthesize the necessary arguments in the order that they appear. For that purpose there is a variant of `apply` called `fapply`:

```
example : ∃ a : ℕ, a = a :=
begin
  fapply exists.intro,
  exact 0,
  reflexivity
end
```

Here, the command `fapply exists.intro` leaves two goals. The first requires us to provide a natural number, `a`, and the second requires us to prove that `a = a`. The second goal depends on the first, so solving the first goal instantiates a metavariable in the second goal, which we then prove with `reflexivity`.

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

```
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 example, 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 = y → y = x
  intros,
  symmetry,
  assumption
end
```

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

```
example : 3 = 3 :=
begin
  generalize : 3 = x,
  -- goal is x : ℕ ⊢ x = x,
  revert x,
```

(continues on next page)

(continued from previous page)

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

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

```
example : 2 + 3 = 5 :=
begin
  generalize : 3 = x,
  -- goal is x : ℕ ⊢ 2 + x = 5,
  sorry
end
```

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

```
example : 2 + 3 = 5 :=
begin
  generalize h : 3 = x,
  -- goal is x : ℕ, h : 3 = x ⊢ 2 + x = 5,
  rw ←h
end
```

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

5.3 More Tactics

Some additional tactics are useful for constructing and destructing propositions and data. For example, when applied to a goal of the form `p ∨ q`, the tactics `left` and `right` are equivalent to `apply or.inl` and `apply or.inr`, respectively. Conversely, the `cases` tactic can be used to decompose a disjunction.

```
example (p q : Prop) : p ∨ q → q ∨ p :=
begin
  intro h,
  cases h with hp hq,
  -- case hp : p
  right, exact hp,
  -- case hq : q
  left, exact hq
end
```

After `cases h` is applied, there are two goals. In the first, the hypothesis `h : p ∨ q` is replaced by `hp : p`, and in the second, it is replaced by `hq : q`. The `cases` tactic can also be used to decompose a conjunction.

```
example (p q : Prop) : p ∧ q → q ∧ p :=
begin
  intro h,
  cases h with hp hq,
  constructor, exact hq, exact hp
end
```

In this example, there is only one goal after the `cases` tactic is applied, with `h : p ∧ q` replaced by a pair of assumptions, `hp : p` and `hq : q`. The `constructor` tactic applies the unique constructor for conjunction, `and.intro`. With these tactics, an example from the previous section can be rewritten as follows:

```
example (p q r : Prop) : p ∧ (q ∨ r) ↔ (p ∧ q) ∨ (p ∧ r) :=
begin
  apply iff.intro,
  intro h,
   cases h with hp hqr,
   cases hqr with hq hr,
     left, constructor, repeat { assumption },
     right, constructor, repeat { assumption },
  intro h,
    cases h with hpq hpr,
      cases hpq with hp hq,
        constructor, exact hp, left, exact hq,
      cases hpr with hp hr,
        constructor, exact hp, right, exact hr
end
```

We will see in Chapter 7 that these tactics are quite general. The `cases` tactic can be used to decompose any element of an inductively defined type; `constructor` always applies the first constructor of an inductively defined type, and `left` and `right` can be used with inductively defined types with exactly `two` constructors. For example, we can use `cases` and `constructor` with an existential quantifier:

```
example (p q : ℕ → Prop) : (∃ x, p x) → ∃ x, p x ∨ q x :=
begin
  intro h,
  cases h with x px,
  constructor, left, exact px
end
```

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

```
example (p q : ℕ → Prop) : (∃ x, p x) → ∃ x, p x ∨ q x :=
begin
  intro h,
  cases h with x px,
  existsi x, left, exact px
end
```

Here is another example:

```
example (p q : ℕ → Prop) :
  (∃ x, p x ∧ q x) → ∃ x, q x ∧ p x :=
begin
  intro h,
  cases h with x hpq,
  cases hpq with hp hq,
  existsi x,
  split; assumption
end
```

Here the semicolon after `split` tells Lean to apply the `assumption` tactic to both of the goals that are introduced by splitting the conjunction; see Section 5.5 for more information.

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

```
universes u v

def swap_pair {α : Type u} {β : Type v} : α × β → β × α :=
begin
  intro p,
  cases p with ha hb,
  constructor, exact hb, exact ha
end

def swap_sum {α : Type u} {β : Type v} : α ⊕ β → β ⊕ α :=
begin
  intro p,
  cases p with ha hb,
    right, exact ha,
    left, exact hb
end
```

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

```
open nat

example (P : ℕ → Prop) (h₀ : P 0) (h₁ : ∀ n, P (succ n)) (m : ℕ) :
  P m :=
begin
  cases m with m', exact h₀, exact h₁ m'
end
```

The `cases` tactic, and its companion, the `induction` tactic, are discussed in greater detail in Section 7.6.

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

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

5.4 Structuring Tactic Proofs

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

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

```
example (p q r : Prop) : p ∧ (q ∨ r) → (p ∧ q) ∨ (p ∧ r) :=
begin
  intro h,
  exact
```

(continues on next page)

(continued from previous page)

```
    have hp : p, from h.left,
    have hqr : q ∨ r, from h.right,
    show (p ∧ q) ∨ (p ∧ r),
    begin
      cases hqr with hq hr,
        exact or.inl ⟨hp, hq⟩,
      exact or.inr ⟨hp, hr⟩
    end
end
```

The following is a more natural example:

```
example (p q r : Prop) : p ∧ (q ∨ r) ↔ (p ∧ q) ∨ (p ∧ r) :=
begin
  apply iff.intro,
    intro h,
    cases h.right with hq hr,
      exact or.inl ⟨h.left, hq⟩,
    exact or.inr ⟨h.left, hr⟩,
  intro h,
  cases h with hpq hpr,
    exact ⟨hpq.left, or.inl hpq.right⟩,
  exact ⟨hpr.left, or.inr hpr.right⟩
end
```

In fact, there is a `show` tactic, which is the analog of the `show` keyword in a proof term. It simply declares the type of the goal that is about to be solved, while remaining in tactic mode. Moreover, in tactic mode, `from` is an alternative name for `exact`. With the `show` and `from` tactics, the previous proof can be written more perspicuously as follows:

```
example (p q r : Prop) : p ∧ (q ∨ r) ↔ (p ∧ q) ∨ (p ∧ r) :=
begin
  apply iff.intro,
    intro h,
    cases h.right with hq hr,
      show (p ∧ q) ∨ (p ∧ r),
        from or.inl ⟨h.left, hq⟩,
      show (p ∧ q) ∨ (p ∧ r),
        from or.inr ⟨h.left, hr⟩,
  intro h,
  cases h with hpq hpr,
    show p ∧ (q ∨ r),
      from ⟨hpq.left, or.inl hpq.right⟩,
    show p ∧ (q ∨ r),
      from ⟨hpr.left, or.inr hpr.right⟩
end
```

Alternatively, you can leave off the `from` and remain in tactic mode:

```
example (p q r : Prop) : p ∧ (q ∨ r) ↔ (p ∧ q) ∨ (p ∧ r) :=
begin
  apply iff.intro,
    intro h,
    cases h.right with hq hr,
      show (p ∧ q) ∨ (p ∧ r),
        { left, split, exact h.left, assumption },
      show (p ∧ q) ∨ (p ∧ r),
```

(continues on next page)

(continued from previous page)

```
        { right, split, exact h.left, assumption },
  intro h,
  cases h with hpq hpr,
    show p ∧ (q ∨ r),
      { cases hpq, split, assumption, left, assumption },
    show p ∧ (q ∨ r),
      { cases hpr, split, assumption, right, assumption }
end
```

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

```
example (n : ℕ) : n + 1 = nat.succ n :=
begin
  show nat.succ n = nat.succ n,
  reflexivity
end
```

In fact, `show` does a little more work. When there are multiple goals, you can use `show` to select which goal you want to work on. Thus both proofs below work:

```
example (p q : Prop) : p ∧ q → q ∧ p :=
begin
  intro h,
  cases h with hp hq,
  split,
  show q, from hq,
  show p, from hp
end

example (p q : Prop) : p ∧ q → q ∧ p :=
begin
  intro h,
  cases h with hp hq,
  split,
  show p, from hp,
  show q, from hq
end
```

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

```
example (p q r : Prop) : p ∧ (q ∨ r) → (p ∧ q) ∨ (p ∧ r) :=
begin
  intro h,
  cases h with hp hqr,
  show (p ∧ q) ∨ (p ∧ r),
  cases hqr with hq hr,
    have hpq : p ∧ q,
      from and.intro hp hq,
    left, exact hpq,
  have hpr : p ∧ r,
    from and.intro hp hr,
  right, exact hpr
end
```

As with `show`, you can omit the `from` and stay in tactic mode:

```
example (p q r : Prop) : p ∧ (q ∨ r) → (p ∧ q) ∨ (p ∧ r) :=
begin
  intro h,
  cases h with hp hqr,
  show (p ∧ q) ∨ (p ∧ r),
  cases hqr with hq hr,
    have hpq : p ∧ q,
      split; assumption,
    left, exact hpq,
  have hpr : p ∧ r,
    split; assumption,
  right, exact hpr
end
```

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

```
example (p q r : Prop) : p ∧ (q ∨ r) → (p ∧ q) ∨ (p ∧ r) :=
begin
  intro h,
  cases h with hp hqr,
  show (p ∧ q) ∨ (p ∧ r),
  cases hqr with hq hr,
    have : p ∧ q,
      split; assumption,
    left, exact this,
  have : p ∧ r,
    split; assumption,
  right, exact this
end
```

You can also use the `have` tactic with the `:=` token, which has the same effect as `from`:

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

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

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

```
example : ∃ x, x + 2 = 8 :=
begin
  let a : ℕ := 3 * 2,
  existsi a,
  reflexivity
end
```

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

For even more structured proofs, you can nest `begin...end` blocks within other `begin...end` blocks. In a nested block, Lean focuses on the first goal, and generates an error if it has not been fully solved at the end of the block. This can be helpful in indicating the separate proofs of multiple subgoals introduced by a tactic.

```
example (p q r : Prop) : p ∧ (q ∨ r) ↔ (p ∧ q) ∨ (p ∧ r) :=
begin
  apply iff.intro,
  begin
    intro h,
    cases h.right with hq hr,
    begin
      show (p ∧ q) ∨ (p ∧ r),
        exact or.inl ⟨h.left, hq⟩
    end,
    show (p ∧ q) ∨ (p ∧ r),
      exact or.inr ⟨h.left, hr⟩
  end,
  intro h,
  cases h with hpq hpr,
  begin
    show p ∧ (q ∨ r),
      exact ⟨hpq.left, or.inl hpq.right⟩
  end,
  show p ∧ (q ∨ r),
    exact ⟨hpr.left, or.inr hpr.right⟩
end
```

Here, we have introduced a new `begin..end` block whenever a tactic leaves more than one subgoal. You can check that at every line in this proof, there is only one goal visible. Notice that you still need to use a comma after a `begin...end` block when there are remaining goals to be discharged.

Within a `begin...end` block, you can abbreviate nested occurrences of `begin` and `end` with curly braces:

```
example (p q r : Prop) : p ∧ (q ∨ r) ↔ (p ∧ q) ∨ (p ∧ r) :=
begin
  apply iff.intro,
  { intro h,
    cases h.right with hq hr,
    { show (p ∧ q) ∨ (p ∧ r),
        exact or.inl ⟨h.left, hq⟩ },
    show (p ∧ q) ∨ (p ∧ r),
      exact or.inr ⟨h.left, hr⟩ },
  intro h,
  cases h with hpq hpr,
  { show p ∧ (q ∨ r),
      exact ⟨hpq.left, or.inl hpq.right⟩ },
  show p ∧ (q ∨ r),
    exact ⟨hpr.left, or.inr hpr.right⟩
end
```

This helps explain the convention on indentation we have adopted here: every time a tactic leaves more than one subgoal, we separate the remaining subgoals by enclosing them in blocks and indenting, until we are back down to one subgoal. Thus if the application of theorem `foo` to a single goal produces four subgoals, one would expect the proof to look like this:

```
begin
  apply foo,
  { ... proof of first goal ... },
  { ... proof of second goal ... },
  { ... proof of third goal ... },
  proof of final goal
end
```

Another reasonable convention is to enclose *all* the remaining subgoals in indented blocks, including the last one:

```
example (p q r : Prop) : p ∧ (q ∨ r) ↔ (p ∧ q) ∨ (p ∧ r) :=
begin
  apply iff.intro,
  { intro h,
    cases h.right with hq hr,
    { show (p ∧ q) ∨ (p ∧ r),
        exact or.inl ⟨h.left, hq⟩ },
    { show (p ∧ q) ∨ (p ∧ r),
        exact or.inr ⟨h.left, hr⟩ }},
  { intro h,
    cases h with hpq hpr,
    { show p ∧ (q ∨ r),
        exact ⟨hpq.left, or.inl hpq.right⟩ },
    { show p ∧ (q ∨ r),
        exact ⟨hpr.left, or.inr hpr.right⟩ }}
end
```

With this convention, the proof using `foo` described above would look like this:

```
begin
  apply foo,
  { ... proof of first goal ... },
  { ... proof of second goal ... },
  { ... proof of third goal ... },
  { ... proof of final goal ....}
end
```

Both conventions are reasonable. The second convention has the effect that the text in a long proof gradually creeps to the right. Many theorems in mathematics have side conditions that can be dispelled quickly; using the first convention means that the proofs of these side conditions are indented until we return to the "linear" part of the proof.

Combining these various mechanisms makes for nicely structured tactic proofs:

```
example (p q : Prop) : p ∧ q ↔ q ∧ p :=
begin
  apply iff.intro,
  { intro h,
    have hp : p := h.left,
    have hq : q := h.right,
    show q ∧ p,
      exact ⟨hq, hp⟩ },
  intro h,
  have hp : p := h.right,
  have hq : q := h.left,
  show p ∧ q,
    exact ⟨hp, hq⟩
end
```

5.5 Tactic Combinators

Tactic combinators are operations that form new tactics from old ones. A sequencing combinator is already implicit in the commas that appear in a `begin...end` block:

```
example (p q : Prop) (hp : p) : p ∨ q :=
begin left, assumption end
```

This is essentially equivalent to the following:

```
example (p q : Prop) (hp : p) : p ∨ q :=
by { left, assumption }
```

Here, `{ left, assumption }` is functionally equivalent to a single tactic which first applies `left` and then applies `assumption`.

In an expression t_1; t_2, the semicolon provides a *parallel* version of the sequencing operation: t_1 is applied to the current goal, and then t_2 is applied to *all* the resulting subgoals:

```
example (p q : Prop) (hp : p) (hq : q) : p ∧ q :=
by split; assumption
```

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

The *orelse* combinator, denoted `<|>`, applies one tactic, and then backtracks and applies another one if the first one fails:

```
example (p q : Prop) (hp : p) : p ∨ q :=
by { left, assumption } <|> { right, assumption}

example (p q : Prop) (hq : q) : p ∨ q :=
by { left, assumption } <|> { right, assumption}
```

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

```
example (p q r : Prop) (hp : p) : p ∨ q ∨ r :=
by repeat { {left, assumption} <|> right <|> assumption }

example (p q r : Prop) (hq : q) : p ∨ q ∨ r :=
by repeat { {left, assumption} <|> right <|> assumption }

example (p q r : Prop) (hr : r) : p ∨ q ∨ r :=
by repeat { {left, assumption} <|> right <|> assumption }
```

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

Incidentally, a tactic expression is really a formal term in Lean, of type `tactic` α for some α. Tactics can be defined and then applied later on.

```
meta def my_tac : tactic unit :=
`[ repeat { {left, assumption} <|> right <|> assumption } ]

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

(continues on next page)

(continued from previous page)

```
example (p q r : Prop) (hq : q) : p ∨ q ∨ r :=
by my_tac

example (p q r : Prop) (hr : r) : p ∨ q ∨ r :=
by my_tac
```

With a `begin...end` block or after a `by`, Lean's parser uses special mechanisms to parse these expressions, but they are similar to ordinary expressions in Lean like `x + 2` and `list α`. (The annotation `[...]` in the definition of `my_tac` above invokes the special parsing mechanism here, too.) The book Programming in Lean provides a fuller introduction to writing tactics and installing them for interactive use. The tactic combinators we are discussing here serve as casual entry points to the tactic programming language.

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

```
example (p q r : Prop) (hp : p) (hq : q) (hr : r) :
  p ∧ q ∧ r :=
by split; try {split}; assumption
```

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

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

```
example (p q r : Prop) (hp : p) (hq : q) (hr : r) :
  p ∧ q ∧ r :=
begin
  split,
  all_goals { try {split} },
  all_goals { assumption }
end
```

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

```
example (p q r : Prop) (hp : p) (hq : q) (hr : r) :
  p ∧ q ∧ r :=
begin
  split,
  any_goals { split },
  any_goals { assumption }
end
```

The first tactic in the `begin...end` block below repeatedly splits conjunctions:

```
example (p q r : Prop) (hp : p) (hq : q) (hr : r) :
  p ∧ ((p ∧ q) ∧ r) ∧ (q ∧ r ∧ p) :=
begin
  repeat { any_goals { split }},
  all_goals { assumption }
end
```

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

```
example (p q r : Prop) (hp : p) (hq : q) (hr : r) :
  p ∧ ((p ∧ q) ∧ r) ∧ (q ∧ r ∧ p) :=
by repeat { any_goals { split <|> assumption} }
```

The combinators `focus` and `solve1` go in the other direction. Specifically, `focus t` ensures that `t` only effects the current goal, temporarily hiding the others from the scope. So, if `t` ordinarily only effects the current goal, `focus { all_goals {t} }` has the same effect as `t`. The tactic `solve1 t` is similar, except that it fails unless `t` succeeds in solving the goal entirely. The `done` tactic is also sometimes useful to direct the flow of control; it succeeds only if there are no goals left to be solved.

5.6 Rewriting

The `rewrite` tactic (abbreviated `rw`) and the `simp` tactic were introduced briefly in Section 4.3. In this section and the next, we discuss them in greater detail.

The `rewrite` tactic provides a basic mechanism for applying substitutions to goals and hypotheses, providing a convenient and efficient way of working with equality. The most basic form of the tactic is `rewrite t`, where `t` is a term whose type asserts an equality. For example, `t` can be a hypothesis `h : x = y` in the context; it can be a general lemma, like `add_comm : ∀ x y, x + y = y + x`, in which the rewrite tactic tries to find suitable instantiations of `x` and `y`; or it can be any compound term asserting a concrete or general equation. In the following example, we use this basic form to rewrite the goal using a hypothesis.

```
variables (f : ℕ → ℕ) (k : ℕ)

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

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

```
example (x y : ℕ) (p : ℕ → Prop) (q : Prop) (h : q → x = y)
  (h' : p y) (hq : q) : p x :=
by { rw (h hq), assumption }
```

Here, `h hq` establishes the equation `x = y`. The parentheses around `h hq` are not necessary, but we have added them for clarity.

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

```
variables (f : ℕ → ℕ) (k : ℕ)

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

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

```
variables (f : ℕ → ℕ) (a b : ℕ)
```

(continues on next page)

(continued from previous page)

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

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

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

```
import data.nat.basic

example (a b c : ℕ) : a + b + c = a + c + b :=
begin
  rw [add_assoc, add_comm b, ←add_assoc]
end

example (a b c : ℕ) : a + b + c = a + c + b :=
begin
  rw [add_assoc, add_assoc, add_comm b]
end

example (a b c : ℕ) : a + b + c = a + c + b :=
begin
  rw [add_assoc, add_assoc, add_comm _ b]
end
```

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

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

```
variables (f : ℕ → ℕ) (a : ℕ)

example (h : a + 0 = 0) : f a = f 0 :=
by { rw add_zero at h, rw h }
```

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

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

```
def tuple (α : Type*) (n : ℕ) :=
  { l : list α // list.length l = n }

variables {α : Type*} {n : ℕ}

example (h : n = 0) (t : tuple α n) : tuple α 0 :=
begin
  rw h at t,
```

(continues on next page)

(continued from previous page)

```
  exact t
end
```

Note that the rewrite tactic can carry out generic calculations in any algebraic structure. The following examples involve an arbitrary ring and an arbitrary group, respectively.

```
import algebra.ring

example {α : Type*} [ring α] (a b c : α) :
  a * 0 + 0 * b + c * 0 + 0 * a = 0 :=
begin
  rw [mul_zero, mul_zero, zero_mul, zero_mul],
  repeat { rw add_zero }
end

example {α : Type*} [group α] {a b : α} (h : a * b = 1) :
  a⁻¹ = b :=
by rw [←(mul_one a⁻¹), ←h, inv_mul_cancel_left]
```

Using the type class mechanism described in Chapter 10, Lean identifies both abstract and concrete instances of the relevant algebraic structures, and instantiates the relevant facts accordingly.

5.7 Using the Simplifier

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

```
import data.nat.basic

variables (x y z : ℕ) (p : ℕ → Prop)
variable  (h : p (x * y))

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

include h
example : p ((x + 0) * (0 + y * 1 + z * 0)) :=
by { simp, assumption }
```

In the first example, the left-hand side of the equality in the goal is simplified using the usual identities involving 0 and 1, reducing the goal to `x * y = x * y`. At that point, `simp` applies reflexivity to finish it off. In the second example, `simp` reduces the goal to `p (x * y)`, at which point the assumption `h` finishes it off. (Remember that we have to `include h` explicitly because it is not explicitly mentioned.) Here are some more examples with lists:

```
import data.list.basic

variable {α : Type*}

open list

example (xs : list ℕ) :
  reverse (xs ++ [1, 2, 3]) = [3, 2, 1] ++ reverse xs :=
by simp
```

(continues on next page)

(continued from previous page)

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

This example uses facts about lists that are found in Lean's mathematics library, which we need to explicitly *import*.

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

```
variables (x y z : ℕ) (p : ℕ → Prop)

example (h : p ((x + 0) * (0 + y * 1 + z * 0))) :
  p (x * y) :=
by { simp at h, assumption }
```

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

```
variables (w x y z : ℕ) (p : ℕ → Prop)

local attribute [simp] mul_comm mul_assoc mul_left_comm
local attribute [simp] add_assoc add_comm add_left_comm

example (h : p (x * y + z * w  * x)) : p (x * w * z + y * x) :=
by { simp at *, assumption }

example (h₁ : p (1 * x + y)) (h₂ : p  (x * z * 1)) :
  p (y + 0 + x) ∧ p (z * x) :=
by { simp at *, split; assumption }
```

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

```
example : x * y + z * w  * x = x * w * z + y * x :=
by simp

example (h : p (x * y + z * w  * x)) : p (x * w * z + y * x) :=
begin simp, simp at h, assumption end
```

As with the rewriter, the simplifier behaves appropriately in algebraic structures:

```
import algebra.ring

variables {α : Type*} [comm_ring α]

local attribute [simp] mul_comm mul_assoc mul_left_comm
local attribute [simp] add_assoc add_comm add_left_comm

example (x y z : α) : (x - x) * y + z = z :=
```

(continues on next page)

(continued from previous page)

```
begin simp end

example (x y z w : α) : x * y + z * w  * x = x * w * z + y * x :=
by simp
```

As with `rewrite`, you can send `simp` a list of facts to use, including general lemmas, local hypotheses, definitions to unfold, and compound expressions. The `simp` tactic does not recognize the `←t` syntax that `rewrite` does, so to use an identity in the other direction you need to use `eq.symm` explicitly. In any case, the additional rules are added to the collection of identities that are used to simplify a term.

```
def f (m n : ℕ) : ℕ := m + n + m

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

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

```
variables (f : ℕ → ℕ) (k : ℕ)

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

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

```
example (h₁ : f 0 = 0) (h₂ : k = 0) : f k = 0 :=
by simp *
```

Here is another example:

```
import data.nat.basic

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

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

```
variables (p q r : Prop)

example (hp : p) : p ∧ q ↔ q :=
by simp *

example (hp : p) : p ∨ q :=
by simp *

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

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

```
import data.nat.basic

variables (u w x x' y y' z : ℕ) (p : ℕ → Prop)

example (h₁ : x + 0 = x') (h₂ : y + 0 = y') :
```

(continues on next page)

(continued from previous page)

```
  x + y + 0 = x' + y' :=
by { simp at *, simp * }
```

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

```
import data.list.basic

open list

variables {α : Type*} (x y z : α) (xs ys zs : list α)

def mk_symm (xs : list α) := xs ++ reverse xs
```

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

```
theorem reverse_mk_symm (xs : list α) :
  reverse (mk_symm xs) = mk_symm xs :=
by { unfold mk_symm, simp }
```

Or even more simply,

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

We can now use this theorem to prove new results:

```
example (xs ys : list ℕ) :
  reverse (xs ++ mk_symm ys) = mk_symm ys ++ reverse xs :=
by simp [reverse_mk_symm]

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

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

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

example (xs ys : list ℕ) :
  reverse (xs ++ mk_symm ys) = mk_symm ys ++ reverse xs :=
by simp

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

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

```
attribute [simp]
theorem reverse_mk_symm (xs : list α) :
  reverse (mk_symm xs) = mk_symm xs :=
by simp [mk_symm]
```

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

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

attribute [simp] reverse_mk_symm

example (xs ys : list ℕ) :
  reverse (xs ++ mk_symm ys) = mk_symm ys ++ reverse xs :=
by simp

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

Once the attribute is applied, however, there is no way to remove it; it persists in any file that imports the one where the attribute is assigned. As we will discuss further in Section 6.4, one can limit the scope of an attribute to the current file or section using the `local attribute` command:

```
section
local attribute [simp] reverse_mk_symm

example (xs ys : list ℕ) :
  reverse (xs ++ mk_symm ys) = mk_symm ys ++ reverse xs :=
by simp

example (xs ys : list ℕ) (p : list ℕ → Prop)
    (h : p (reverse (xs ++ (mk_symm ys)))) :
  p (mk_symm ys ++ reverse xs) :=
by simp at h; assumption

end
```

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

You can even create your own sets of simplifier rules, to be applied in special situations.

```
run_cmd mk_simp_attr `my_simps

attribute [my_simps] reverse_mk_symm

example (xs ys : list ℕ) :
  reverse (xs ++ mk_symm ys) = mk_symm ys ++ reverse xs :=
by simp with my_simps

example (xs ys : list ℕ) (p : list ℕ → Prop)
  (h : p (reverse (xs ++ (mk_symm ys)))) :
    p (mk_symm ys ++ reverse xs) :=
by simp with my_simps at h; assumption
```

The command `run_cmd mk_simp_attr `my_simps` creates a new attribute `[my_simps]`. (The backtick is used to indicate that `my_simps` is a new name, something that is explained more fully in Programming in Lean.) The command `simp with my_simps` then adds all the theorems that have been marked with attribute `[my_simps]` to the default set of theorems marked with attribute `[simp]` before applying `[simp]`, and similarly with `simp with my_simps at h`.

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

There are two additional modifiers that are useful. By default, `simp` includes all theorems that have been marked with the attribute `[simp]`. Writing `simp only` excludes these defaults, allowing you to use a more explicitly crafted list of rules. Alternatively, writing `simp without t` filters `t` and removes it from the set of simplification rules. In the examples below, the minus sign and `only` are used to block the application of `reverse_mk_symm`.

```
attribute [simp] reverse_mk_symm

example (xs ys : list ℕ) (p : list ℕ → Prop)
    (h : p (reverse (xs ++ (mk_symm ys)))) :
  p (mk_symm ys ++ reverse xs) :=
by { simp at h, assumption }

example (xs ys : list ℕ) (p : list ℕ → Prop)
    (h : p (reverse (xs ++ (mk_symm ys)))) :
  p (reverse (mk_symm ys) ++ reverse xs) :=
by { simp [-reverse_mk_symm] at h, assumption }

example (xs ys : list ℕ) (p : list ℕ → Prop)
    (h : p (reverse (xs ++ (mk_symm ys)))) :
  p (reverse (mk_symm ys) ++ reverse xs) :=
by { simp only [reverse_append] at h, assumption }
```

5.8 Exercises

1. Go back to the exercises in Chapter 3 and Chapter 4 and redo as many as you can now with tactic proofs, using also `rw` and `simp` as appropriate.
2. Use tactic combinators to obtain a one line proof of the following:

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

CHAPTER

SIX

INTERACTING WITH LEAN

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

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

6.1 Importing Files

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

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

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

```
import foo bar.baz.blah
```

imports the files `foo.lean` and `bar/baz/blah.lean`, where the descriptions are interpreted relative to the Lean *search path*. Information as to how the search path is determined can be found on the documentation pages. By default, it includes the standard library directory, and (in some contexts) the root of the user's local project. One can also specify imports relative to the current directory; for example,

```
import .foo ..bar.baz
```

tells Lean to import `foo.lean` from the current directory and `bar/baz.lean` relative to the parent of the current directory.

Importing is transitive. In other words, if you import `foo` and `foo` imports `bar`, then you also have access to the contents of `bar`, and do not need to import it explicitly.

6.2 More on Sections

Lean provides various sectioning mechanisms to help structure a theory. We saw in Section 2.6 that the `section` command makes it possible not only to group together elements of a theory that go together, but also to declare variables that are inserted as arguments to theorems and definitions, as necessary. Remember that the point of the variable command is to declare variables for use in theorems, as in the following example:

```
import data.nat.basic

section
variables x y : ℕ

def double := x + x

#check double y
#check double (2 * x)

local attribute [simp] add_assoc add_comm add_left_comm

theorem t1 : double (x + y) = double x + double y :=
by simp [double]

#check t1 y
#check t1 (2 * x)

theorem t2 : double (x * y) = double x * y :=
by simp [double, add_mul]
end
```

The definition of `double` does not have to declare `x` as an argument; Lean detects the dependence and inserts it automatically. Similarly, Lean detects the occurrence of `x` in `t1` and `t2`, and inserts it automatically there, too.

Note that double does *not* have `y` as argument. Variables are only included in declarations where they are actually mentioned. More precisely, they must be mentioned outside of a tactic block; because variables can appear and can be renamed dynamically in a tactic proof, there is no reliable way of determining when a name used in a tactic proof refers to an element of the context in which the theorem is parsed, and Lean does not try to guess. You can manually ask Lean to include a variable in every definition in a section with the `include` command.

```
section
variables (x y z : ℕ)
variables (h₁ : x = y) (h₂ : y = z)

include h₁ h₂
theorem foo : x = z :=
begin
  rw [h₁, h₂]
end
omit h₁ h₂

theorem bar : x = z :=
eq.trans h₁ h₂

theorem baz : x = x := rfl

#check @foo
#check @bar
#check @baz
```

(continues on next page)

(continued from previous page)

```
end
```

The `omit` command simply undoes the effect of the `include`. It does not, however, prevent the arguments from being included automatically in subsequent theorems that mention them. The scope of the `include` statement can also be delimited by enclosing it in a section.

```
section include_hs
include h₁ h₂

theorem foo : x = z :=
begin
  rw [h₁, h₂]
end

end include_hs
```

The include command is often useful with structures that are not mentioned explicitly but meant to be inferred by type class inference, as described in Chapter 10.

It is often the case that we want to declare section variables as explicit variables but later make them implicit, or vice-versa. One can do this with a `variables` command that mentions these variables with the desired brackets, without repeating the type again. Once again, sections can be used to delimit scope. In the example below, the variables `x`, `y`, and `z` are marked implicit in `foo` but explicit in `bar`, while `x` is (somewhat perversely) marked as implicit in `baz`.

```
section
variables (x y z : ℕ)
variables (h₁ : x = y) (h₂ : y = z)

section
variables {x y z}
include h₁ h₂
theorem foo : x = z :=
begin
  rw [h₁, h₂]
end
end

theorem bar : x = z :=
eq.trans h₁ h₂

variable {x}
theorem baz : x = x := rfl

#check @foo
#check @bar
#check @baz
end
```

Using these subsequent `variables` commands does not change the order in which variables are inserted. It only changes the explicit / implicit annotations.

In fact, Lean has two ways of introducing local elements into the sections, namely, as `variables` or as `parameters`. In the initial example in this section, the variable `x` is generalized immediately, so that even within the section `double` is a function of `x`, and `t1` and `t2` depend explicitly on `x`. This is what makes it possible to apply `double` and `t1` to other expressions, like `y` and `2 * x`. It corresponds to the ordinary mathematical locution "in this section, let `x` and `y` range over the natural numbers." Whenever `x` and `y` occur, we assume they denote natural numbers, but we do not assume they refer to the same natural number from theorem to theorem.

Sometimes, however, we wish to *fix* a value in a section. For example, following ordinary mathematical vernacular, we might say "in this section, we fix a type, α, and a binary relation `r` on α." The notion of a `parameter` captures this usage:

```
section
parameters {α : Type*} (r : α → α → Prop)
parameter  transr : ∀ {x y z}, r x y → r y z → r x z

variables {a b c d e : α}

theorem t1 (h₁ : r a b) (h₂ : r b c) (h₃ : r c d) : r a d :=
transr (transr h₁ h₂) h₃

theorem t2 (h₁ : r a b) (h₂ : r b c) (h₃ : r c d)
    (h₄ : r d e) :
  r a e :=
transr h₁ (t1 h₂ h₃ h₄)

#check t1
#check t2
end

#check t1
#check t2
```

As with variables, the parameters α, `r`, and `transr` are inserted as arguments to definitions and theorems as needed. But there is a difference: within the section, `t1` is an abbreviation for `@t1 α r transr`, which is to say, these arguments are held fixed until the section is closed. On the plus side, this means that you do not have to specify the explicit arguments `r` and `transr` when you write `t1 h₂ h₃ h₄`, in contrast to the previous example. But it also means that you cannot specify other arguments in their place. In this example, making `r` a parameter is appropriate if `r` is the only binary relation you want to reason about in the section. In that case, it would make sense to introduce temporary infix notation like $\preceq$ for `r`, and we will see in Section 6.6 how to do that. On the other hand, if you want to apply your theorems to arbitrary binary relations within the section, you should make `r` a variable.

6.3 More on Namespaces

In Lean, identifiers are given by hierarchical *names* like `foo.bar.baz`. We saw in Section 2.7 that Lean provides mechanisms for working with hierarchical names. The command `namespace foo` causes `foo` to be prepended to the name of each definition and theorem until `end foo` is encountered. The command `open foo` then creates temporary *aliases* to definitions and theorems that begin with prefix `foo`.

```
namespace foo
def bar : ℕ := 1
end foo

open foo

#check bar
#check foo.bar
```

It is not important that the definition of `foo.bar` was the result of a `namespace` command:

```
def foo.bar : ℕ := 1

open foo
```

(continues on next page)

(continued from previous page)

```
#check bar
#check foo.bar
```

Although the names of theorems and definitions have to be unique, the aliases that identify them do not. For example, the standard library defines a theorem `add_sub_cancel`, which asserts `a + b - b = a` in any additive group. The corresponding theorem on the natural numbers is named `nat.add_sub_cancel`; it is not a special case of `add_sub_cancel`, because the natural numbers do not form a group. When we open the `nat` namespace, the expression `add_sub_cancel` is overloaded, and can refer to either one. Lean tries to use type information to disambiguate the meaning in context, but you can always disambiguate by giving the full name. To that end, the string `_root_` is an explicit description of the empty prefix.

```
import algebra.ordered_ring

#check add_sub_cancel
#check nat.add_sub_cancel
#check _root_.add_sub_cancel
```

We can prevent the shorter alias from being created by using the `protected` keyword:

```
namespace foo
protected def bar : ℕ := 1
end foo

open foo

-- #check bar -- error
#check foo.bar
```

This is often used for names like `nat.rec` and `nat.rec_on`, to prevent overloading of common names.

The `open` command admits variations. The command

```
open nat (succ add sub)
```

creates aliases for only the identifiers listed. The command

```
open nat (hiding succ add sub)
```

creates aliases for everything in the `nat` namespace *except* the identifiers listed. The command

```
open nat (renaming mul → times) (renaming add → plus)
  (hiding succ sub)
```

creates aliases for everything in the `nat` namespace except `succ` and `sub`, renaming `nat.mul` to `times` and `nat.add` to `plus`.

It is sometimes useful to `export` aliases from one namespace to another, or to the top level. The command

```
export nat (succ add sub)
```

creates aliases for `succ`, `add`, and `sub` in the current namespace, so that whenever the namespace is open, these aliases are available. If this command is used outside a namespace, the aliases are exported to the top level. The `export` command admits all the variations described above.

6.4 Attributes

The main function of Lean is to translate user input to formal expressions that are checked by the kernel for correctness and then stored in the environment for later use. But some commands have other effects on the environment, either assigning attributes to objects in the environment, defining notation, or declaring instances of type classes, as described in Chapter 10. Most of these commands have global effects, which is to say, that they remain in effect not only in the current file, but also in any file that imports it. However, such commands can often be prefixed with the `local` modifier, which indicates that they only have effect until the current `section` or `namespace` is closed, or until the end of the current file.

In Section 5.7, we saw that theorems can be annotated with the `[simp]` attribute, which makes them available for use by the simplifier. The following example defines the prefix relation on lists, proves that this relation is reflexive, and assigns the `[simp]` attribute to that theorem.

```
variable {α : Type*}

def is_prefix (l₁ : list α) (l₂ : list α) : Prop :=
∃ t, l₁ ++ t = l₂

infix ` <+: `:50 := is_prefix

attribute [simp]
theorem list.is_prefix_refl (l : list α) : l <+: l :=
⟨[], by simp⟩

example : [1, 2, 3] <+: [1, 2, 3] := by simp
```

The simplifier then proves `[1, 2, 3] <+: [1, 2, 3]` by rewriting it to `true`. Lean allows the alternative annotation `@[simp]` before a theorem to assign the attribute:

```
@[simp]
theorem list.is_prefix_refl (l : list α) : l <+: l :=
⟨[], by simp⟩
```

One can also assign the attribute any time after the definition takes place:

```
theorem list.is_prefix_refl (l : list α) : l <+: l :=
⟨[], by simp⟩

attribute [simp] list.is_prefix_refl
```

In all these cases, the attribute remains in effect in any file that imports the one in which the declaration occurs. Adding the `local` modifier restricts the scope:

```
section
local attribute [simp]
theorem list.is_prefix_refl (l : list α) : l <+: l :=
⟨[], by simp⟩

example : [1, 2, 3] <+: [1, 2, 3] := by simp
end

-- error:
-- example : [1, 2, 3] <+: [1, 2, 3] := by simp
```

For another example, we can use the `instance` command to assign the notation $\leq$ to the *is_prefix* relation. That command, which will be explained in Chapter 10, works by assigning an `[instance]` attribute to the associated definition.

```
instance list_has_le : has_le (list α) := ⟨is_prefix⟩

theorem list.is_prefix_refl (l : list α) : l ≤ l :=
⟨[], by simp⟩
```

That assignment can also be made local:

```
def list_has_le : has_le (list α) := ⟨is_prefix⟩

section
local attribute [instance] list_has_le

theorem foo (l : list α) : l ≤ l := ⟨[], by simp⟩
end

-- error:
-- theorem bar (l : list α) : l ≤ l := ⟨[], by simp⟩
```

For yet another example, the `reflexivity` tactic makes use of objects in the environment that have been tagged with the `[refl]` attribute:

```
@[simp, refl]
theorem list.is_prefix_refl (l : list α) : l <+: l :=
⟨[], by simp⟩

example : [1, 2, 3] <+: [1, 2, 3] := by reflexivity
```

The scope of the `[refl]` attribute can similarly be restricted using the `local` modifier, as above.

In Section 6.6 below, we will discuss Lean's mechanisms for defining notation, and see that they also support the `local` modifier. However, in Section 6.9, we will discuss Lean's mechanisms for setting options, which does *not* follow this pattern: options can *only* be set locally, which is to say, their scope is always restricted to the current section or current file.

6.5 More on Implicit Arguments

In Section 2.9, we saw that if Lean displays the type of a term `t` as `Π {x : α}, β x`, then the curly brackets indicate that `x` has been marked as an *implicit argument* to `t`. This means that whenever you write `t`, a placeholder, or "hole," is inserted, so that `t` is replaced by `@t _`. If you don't want that to happen, you have to write `@t` instead.

Notice that implicit arguments are inserted eagerly. Suppose we define a function `f (x : ℕ) {y : ℕ} (z : ℕ)` with the arguments shown. Then, when we write the expression `f 7` without further arguments, it is parsed as `f 7 _`. Lean offers a weaker annotation, `{{y : ℕ}}`, which specifies that a placeholder should only be added *before* a subsequent explicit argument. This annotation can also be written using as `⦃y : ℕ⦄`, where the unicode brackets are entered as `\{{` and `\}}`, respectively. With this annotation, the expression `f 7` would be parsed as is, whereas `f 7 3` would be parsed as `f 7 _ 3`, just as it would be with the strong annotation.

To illustrate the difference, consider the following example, which shows that a reflexive euclidean relation is both symmetric and transitive.

```
namespace hidden
variables {α : Type*} (r : α → α → Prop)

definition reflexive  : Prop := ∀ (a : α), r a a
```

(continues on next page)

(continued from previous page)

```
definition symmetric  : Prop := ∀ {a b : α}, r a b → r b a
definition transitive : Prop :=
  ∀ {a b c : α}, r a b → r b c → r a c
definition euclidean  : Prop :=
  ∀ {a b c : α}, r a b → r a c → r b c

variable {r}

theorem th1 (reflr : reflexive r) (euclr : euclidean r) :
  symmetric r :=
assume a b : α, assume : r a b,
show r b a, from euclr this (reflr _)

theorem th2 (symmr : symmetric r) (euclr : euclidean r) :
  transitive r :=
assume (a b c : α), assume (rab : r a b) (rbc : r b c),
euclr (symmr rab) rbc

-- error:
/-
theorem th3 (reflr : reflexive r) (euclr : euclidean r) :
  transitive r :=
th2 (th1 reflr euclr) euclr
-/

theorem th3 (reflr : reflexive r) (euclr : euclidean r) :
  transitive r :=
@th2 _ _ (@th1 _ _ reflr @euclr) @euclr
end hidden
```

The results are broken down into small steps: `th1` shows that a relation that is reflexive and euclidean is symmetric, and `th2` shows that a relation that is symmetric and euclidean is transitive. Then `th3` combines the two results. But notice that we have to manually disable the implicit arguments in `th1`, `th2`, and `euclr`, because otherwise too many implicit arguments are inserted. The problem goes away if we use weak implicit arguments:

```
variables {α : Type*} (r : α → α → Prop)

definition reflexive  : Prop := ∀ (a : α), r a a
definition symmetric  : Prop := ∀ ⦃a b : α⦄, r a b → r b a
definition transitive : Prop :=
  ∀ ⦃a b c : α⦄, r a b → r b c → r a c
definition euclidean  : Prop :=
  ∀ ⦃a b c : α⦄, r a b → r a c → r b c

variable {r}

theorem th1 (reflr : reflexive r) (euclr : euclidean r) :
  symmetric r :=
assume a b : α, assume : r a b,
show r b a, from euclr this (reflr _)

theorem th2 (symmr : symmetric r) (euclr : euclidean r) :
  transitive r :=
assume (a b c : α), assume (rab : r a b) (rbc : r b c),
euclr (symmr rab) rbc
```

(continues on next page)

(continued from previous page)

```
theorem th3 (reflr : reflexive r) (euclr : euclidean r) :
  transitive r :=
th2 (th1 reflr euclr) euclr
```

There is a third kind of implicit argument that is denoted with square brackets, [and]. These are used for type classes, as explained in Chapter 10.

6.6 Notation

Identifiers in Lean can include any alphanumeric characters, including Greek characters (other than Π , Σ , and λ , which, as we have seen, have a special meaning in the dependent type theory). They can also include subscripts, which can be entered by typing _ followed by the desired subscripted character.

Lean's parser is extensible, which is to say, we can define new notation.

```
notation `[` a `**` b `]` := a * b + 1

def mul_square (a b : ℕ) := a * a * b * b

infix `<*>`:50 := mul_square

#reduce [2 ** 3]
#reduce 2 <*> 3
```

In this example, the `notation` command defines a complex binary notation for multiplying and adding one. The `infix` command declares a new infix operator, with precedence 50, which associates to the left. (More precisely, the token is given left-binding power 50.) The command `infixr` defines notation which associates to the right, instead.

If you declare these notations in a namespace, the notation is only available when the namespace is open. You can declare temporary notation using the keyword `local`, in which case the notation is available in the current file, and moreover, within the scope of the current `namespace` or `section`, if you are in one.

```
local notation `[` a `**` b `]` := a * b + 1
local infix `<*>`:50 := λ a b : ℕ, a * a * b * b
```

Lean's core library declares the left-binding powers of a number of common symbols.

https://github.com/leanprover/lean/blob/master/library/init/core.lean

You are welcome to overload these symbols for your own use.

You can direct the pretty-printer to suppress notation with the command `set_option pp.notation false`. You can also declare notation to be used for input purposes only with the `[parsing_only]` attribute:

```
notation [parsing_only] `[` a `**` b `]` := a * b + 1

variables a b : ℕ
#check [a ** b]
```

The output of the `#check` command displays the expression as `a * b + 1`. Lean also provides mechanisms for iterated notation, such as `[a, b, c, d, e]` to denote a list with the indicated elements. See the discussion of `list` in the next chapter for an example.

The possibility of declaring parameters in a section also makes it possible to define local notation that depends on those parameters. In the example below, as long as the parameter `m` is fixed, we can write `a ≡ b` for equivalence modulo `m`.

As soon as the section is closed, however, the dependence on `m` becomes explicit, and the notation `a ≡ b` is no longer valid.

```
import data.int.basic

namespace int

def dvd (m n : ℤ) : Prop := ∃ k, n = m * k
instance : has_dvd int := ⟨int.dvd⟩

@[simp]
theorem dvd_zero (n : ℤ) : n ∣ 0 :=
⟨0, by simp⟩

theorem dvd_intro {m n : ℤ} (k : ℤ) (h : n = m * k) : m ∣ n :=
⟨k, h⟩

end int

open int

section mod_m
parameter (m : ℤ)
variables (a b c : ℤ)

definition mod_equiv := (m ∣ b - a)

local infix ≡ := mod_equiv

theorem mod_refl : a ≡ a :=
show m ∣ a - a, by simp

theorem mod_symm (h : a ≡ b) : b ≡ a :=
by cases h with c hc; apply dvd_intro (-c); simp [eq.symm hc]

local attribute [simp] add_assoc add_comm add_left_comm

theorem mod_trans (h₁ : a ≡ b) (h₂ : b ≡ c) : a ≡ c :=
begin
  cases h₁ with d hd, cases h₂ with e he,
  apply dvd_intro (d + e),
  simp [mul_add, eq.symm hd, eq.symm he, sub_eq_add_neg]
end
end mod_m

#check (mod_refl : ∀ (m a : ℤ), mod_equiv m a a)

#check (mod_symm : ∀ (m a b : ℤ), mod_equiv m a b →
                    mod_equiv m b a)

#check (mod_trans :  ∀ (m a b c : ℤ), mod_equiv m a b →
                      mod_equiv m b c → mod_equiv m a c)
```

6.7 Coercions

In Lean, the type of natural numbers, `nat`, is different from the type of integers, `int`. But there is a function `int.of_nat` that embeds the natural numbers in the integers, meaning that we can view any natural number as an integer, when needed. Lean has mechanisms to detect and insert *coercions* of this sort.

```
variables m n : ℕ
variables i j : ℤ

#check i + m      -- i + ↑m : ℤ
#check i + m + j  -- i + ↑m + j : ℤ
#check i + m + n  -- i + ↑m + ↑n : ℤ
```

Notice that the output of the `#check` command shows that a coercion has been inserted by printing an arrow. The latter is notation for the function `coe`; you can type the unicode arrow with `\u` or use `coe` instead. In fact, when the order of arguments is different, you have to insert the coercion manually, because Lean does not recognize the need for a coercion until it has already parsed the earlier arguments.

```
#check ↑m + i        -- ↑m + i : ℤ
#check ↑(m + n) + i  -- ↑(m + n) + i : ℤ
#check ↑m + ↑n + i   -- ↑m + ↑n + i : ℤ
```

In fact, Lean allows various kinds of coercions using type classes; for details, see Section 10.6.

6.8 Displaying Information

There are a number of ways in which you can query Lean for information about its current state and the objects and theorems that are available in the current context. You have already seen two of the most common ones, `#check` and `#reduce`. Remember that `#check` is often used in conjunction with the `@` operator, which makes all of the arguments to a theorem or definition explicit. In addition, you can use the `#print` command to get information about any identifier. If the identifier denotes a definition or theorem, Lean prints the type of the symbol, and its definition. If it is a constant or an axiom, Lean indicates that fact, and shows the type.

```
-- examples with equality
#check eq
#check @eq
#check eq.symm
#check @eq.symm

#print eq.symm

-- examples with and
#check and
#check and.intro
#check @and.intro

-- a user-defined function
def foo {α : Type*} (x : α) : α := x

#check foo
#check @foo
#reduce foo
#reduce (foo nat.zero)
#print foo
```

There are other useful `#print` commands:

```
#print definition             : display definition
#print inductive              : display an inductive type and its constructors
#print notation               : display all notation
#print notation <tokens>      : display notation using any of the tokens
#print axioms                 : display assumed axioms
#print options                : display options set by user
#print prefix <namespace>     : display all declarations in the namespace
#print classes                : display all classes
#print instances <class name> : display all instances of the given class
#print fields <structure>     : display all fields of a structure
```

We will discuss inductive types, structures, classes, instances in the next four chapters. Here are examples of how these commands are used:

```
import algebra.ring

#print notation
#print notation + * -
#print axioms
#print options
#print prefix nat
#print prefix nat.le
#print classes
#print instances ring
#print fields ring
```

The behavior of the generic print command is determined by its argument, so that the following pairs of commands all do the same thing.

```
import algebra.group

#print list.append
#print definition list.append

#print +
#print notation +

#print nat
#print inductive nat

#print group
#print inductive group
```

Moreover, both `#print group` and `#print inductive group` recognize that a group is a structure (see Chapter 9), and so print the fields as well.

6.9 Setting Options

Lean maintains a number of internal variables that can be set by users to control its behavior. The syntax for doing so is as follows:

```
set_option <name> <value>
```

One very useful family of options controls the way Lean's *pretty- printer* displays terms. The following options take an input of true or false:

```
pp.implicit  : display implicit arguments
pp.universes : display hidden universe parameters
pp.coercions : show coercions
pp.notation  : display output using defined notations
pp.beta      : beta reduce terms before displaying them
```

As an example, the following settings yield much longer output:

```
set_option pp.implicit true
set_option pp.universes true
set_option pp.notation false
set_option pp.numerals false

#check 2 + 2 = 4
#reduce (λ x, x + 2) = (λ x, x + 3)
#check (λ x, x + 1) 1
```

The command `set_option pp.all true` carries out these settings all at once, whereas `set_option pp.all false` reverts to the previous values. Pretty printing additional information is often very useful when you are debugging a proof, or trying to understand a cryptic error message. Too much information can be overwhelming, though, and Lean's defaults are generally sufficient for ordinary interactions.

By default, the pretty-printer does not reduce applied lambda-expressions, but this is sometimes useful. The `pp.beta` option controls this feature.

```
set_option pp.beta true
#check (λ x, x + 1) 1
```

6.10 Elaboration Hints

When you ask Lean to process an expression like `λ x y z, f (x + y) z`, you are leaving information implicit. For example, the types of `x`, `y`, and `z` have to be inferred from the context, the notation `+` may be overloaded, and there may be implicit arguments to `f` that need to be filled in as well. Moreover, we will see in Chapter 10 that some implicit arguments are synthesized by a process known as *type class resolution*. And we have also already seen in the last chapter that some parts of an expression can be constructed by the tactic framework.

Inferring some implicit arguments is straightforward. For example, suppose a function `f` has type `Π {α : Type*}, α → α → α` and Lean is trying to parse the expression `f n`, where `n` can be inferred to have type `nat`. Then it is clear that the implicit argument `α` has to be `nat`. However, some inference problems are *higher order*. For example, the substitution operation for equality, `eq.subst`, has the following type:

```
eq.subst : ∀ {α : Sort u} {p : α → Prop} {a b : α},
             a = b → p a → p b
```

Now suppose we are given `a b : ℕ` and `h₁ : a = b` and `h₂ : a * b > a`. Then, in the expression `eq.subst h₁ h₂`, `P` could be any of the following:

- `λ x, x * b > x`
- `λ x, x * b > a`
- `λ x, a * b > x`
- `λ x, a * b > a`

In other words, our intent may be to replace either the first or second `a` in `h₂`, or both, or neither. Similar ambiguities arise in inferring induction predicates, or inferring function arguments. Even second-order unification is known to be undecidable. Lean therefore relies on heuristics to fill in such arguments, and when it fails to guess the right ones, they need to be provided explicitly.

To make matters worse, sometimes definitions need to be unfolded, and sometimes expressions need to be reduced according to the computational rules of the underlying logical framework. Once again, Lean has to rely on heuristics to determine what to unfold or reduce, and when.

There are attributes, however, that can be used to provide hints to the elaborator. One class of attributes determines how eagerly definitions are unfolded: constants can be marked with the attribute `[reducible]`, `[semireducible]`, or `[irreducible]`. Definitions are marked `[semireducible]` by default. A definition with the `[reducible]` attribute is unfolded eagerly; if you think of a definition as serving as an abbreviation, this attribute would be appropriate. The elaborator avoids unfolding definitions with the `[irreducible]` attribute. Theorems are marked `[irreducible]` by default, because typically proofs are not relevant to the elaboration process.

It is worth emphasizing that these attributes are only hints to the elaborator. When checking an elaborated term for correctness, Lean's kernel will unfold whatever definitions it needs to unfold. As with other attributes, the ones above can be assigned with the `local` modifier, so that they are in effect only in the current section or file.

Lean also has a family of attributes that control the elaboration strategy. A definition or theorem can be marked `[elab_with_expected_type]`, `[elab_simple]`. or `[elab_as_eliminator]`. When applied to a definition `f`, these bear on elaboration of an expression `f a b c ...` in which `f` is applied to arguments. With the default attribute, `[elab_with_expected_type]`, the arguments `a`, `b`, `c`, … are elaborating using information about their expected type, inferred from `f` and the previous arguments. In contrast, with `[elab_simple]`, the arguments are elaborated from left to right without propagating information about their types. The last attribute, `[elab_as_eliminator]`, is commonly used for eliminators like recursors, induction principles, and `eq.subst`. It uses a separate heuristic to infer higher-order parameters. We will consider such operations in more detail in the next chapter.

Once again, these attributes can be assigned and reassigned after an object is defined, and you can use the `local` modifier to limit their scope. Moreover, using the `@` symbol in front of an identifier in an expression instructs the elaborator to use the `[elab_simple]` strategy; the idea is that, when you provide the tricky parameters explicitly, you want the elaborator to weigh that information heavily. In fact, Lean offers an alternative annotation, `@@`, which leaves parameters before the first higher-order parameter implicit. For example, `@@eq.subst` leaves the type of the equation implicit, but makes the context of the substitution explicit.

6.11 Using the Library

To use Lean effectively you will inevitably need to make use of definitions and theorems in the library. Recall that the `import` command at the beginning of a file imports previously compiled results from other files, and that importing is transitive; if you import `foo` and `foo` imports `bar`, then the definitions and theorems from `bar` are available to you as well. But the act of opening a namespace, which provides shorter names, does not carry over. In each file, you need to open the namespaces you wish to use.

In general, it is important for you to be familiar with the library and its contents, so you know what theorems, definitions, notations, and resources are available to you. Below we will see that Lean's editor modes can also help you find things you need, but studying the contents of the library directly is often unavoidable. Lean's standard library can be found online, on github:

https://github.com/leanprover/lean/tree/master/library

You can see the contents of the directories and files using github's browser interface. If you have installed Lean on your own computer, you can find the library in the `lean` folder, and explore it with your file manager. Comment headers at the top of each file provide additional information.

Lean's library developers follow general naming guidelines to make it easier to guess the name of a theorem you need, or to find it using tab completion in editors with a Lean mode that supports this, which is discussed in the next section. Identifiers are generally `snake_case`, which is to say, they are composed of words written in lower case separated by underscores. For the most part, we rely on descriptive names. Often the name of theorem simply describes the conclusion:

```
import data.nat.basic

open nat

#check succ_ne_zero
#check @mul_zero
#check @mul_one
#check @sub_add_eq_add_sub
#check @le_iff_lt_or_eq
```

If only a prefix of the description is enough to convey the meaning, the name may be made even shorter:

```
#check @neg_neg
#check pred_succ
```

Sometimes, to disambiguate the name of theorem or better convey the intended reference, it is necessary to describe some of the hypotheses. The word "of" is used to separate these hypotheses:

```
import algebra.ordered_ring

#check @nat.lt_of_succ_le
#check @lt_of_not_ge
#check @lt_of_le_of_ne
#check @add_lt_add_of_lt_of_le
```

Sometimes the word "left" or "right" is helpful to describe variants of a theorem.

```
import algebra.ordered_ring

#check @add_le_add_left
#check @add_le_add_right
```

We can also use the word "self" to indicate a repeated argument:

```
import algebra.group

#check mul_inv_self
#check neg_add_self
```

Remember that identifiers in Lean can be organized into hierarchical namespaces. For example, the theorem named `lt_of_succ_le` in the namespace `nat` has full name `nat.lt_of_succ_le`, but the shorter name is made available by the command `open nat`. We will see in Chapter 7 and Chapter 9 that defining structures and inductive data types in Lean generates associated operations, and these are stored in a namespace with the same name as the type under definition. For example, the product type comes with the following operations:

```
#check @prod.mk
#check @prod.fst
#check @prod.snd
#check @prod.rec
```

The first is used to construct a pair, whereas the next two, `prod.fst` and `prod.snd`, project the two elements. The last, `prod.rec`, provides another mechanism for defining functions on a product in terms of a function on the two components. Names like `prod.rec` are *protected*, which means that one has to use the full name even when the `prod` namespace is open.

With the propositions as types correspondence, logical connectives are also instances of inductive types, and so we tend to use dot notation for them as well:

```
#check @and.intro
#check @and.elim
#check @and.left
#check @and.right
#check @or.inl
#check @or.inr
#check @or.elim
#check @exists.intro
#check @exists.elim
#check @eq.refl
#check @eq.subst
```

CHAPTER

SEVEN

INDUCTIVE TYPES

We have seen that Lean's formal foundation includes basic types, `Prop, Type 0, Type 1, Type 2, ...`, and allows for the formation of dependent function types, Π `x :` α`,` β. In the examples, we have also made use of additional types like `bool`, `nat`, and `int`, and type constructors, like `list`, and product, `×`. In fact, in Lean's library, every concrete type other than the universes and every type constructor other than 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 : Sort u
| constructor₁ : ... → foo
| constructor₂ : ... → foo
...
| constructorₙ : ... → foo
```

The intuition is that each constructor specifies a way of building new objects of `foo`, possibly from previously constructed values. The type `foo` consists of nothing more than the objects that are constructed in this way. The first character `|` in an inductive declaration is optional. We can also separate constructors using a comma instead of `|`.

We will see below that the arguments 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 [Dybj94].

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.

7.1 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 `sunday`, `monday`, `...`, `saturday` as being distinct elements of `weekday`, with no other distinguishing properties. The elimination principle, `weekday.rec`, is defined along with the type `weekday` and its constructors. It is also known as a *recursor*, and it is what makes the type "inductive": it allows us to define a function on `weekday` by assigning values corresponding to each constructor. The intuition is that an inductive type is exhaustively generated by the constructors, and has no elements beyond those they construct.

We will use a slight variant of `weekday.rec`, `weekday.rec_on` (also generated automatically), which takes its arguments in a more convenient order. (Note that the shorter names `rec` and `rec_on` are not made available by default when we open the `weekday` namespace. This avoids clashes with the functions of the same names for other inductive types.) We can use `weekday.rec_on` to define a function from `weekday` to the natural numbers:

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

#reduce number_of_day weekday.sunday
#reduce number_of_day weekday.monday
#reduce 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.

```
def 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` and `cases_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 def cases_on := @weekday.cases_on

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

#reduce weekday.number_of_day weekday.sunday

open weekday (renaming cases_on → cases_on)

#reduce number_of_day sunday
#check cases_on
```

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

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

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

#reduce next (next tuesday)
#reduce 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.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
```

Using a tactic proof, we can be even more concise:

```
theorem next_previous (d: weekday) :
  next (previous d) = d :=
```

(continues on next page)

(continued from previous page)

```
by apply weekday.cases_on d; refl
```

Section 7.6 below will introduce additional tactics that are specifically designed to make use of inductive types.

Notice that, under the propositions-as-types correspondence, we can use `cases_on` to prove theorems as well as define functions. In fact, we could equally well have used `rec_on`:

```
theorem next_previous (d: weekday) :
  next (previous d) = d :=
by apply weekday.rec_on d; refl
```

In other words, under the propositions-as-types correspondence, the proof by cases is a kind of definition by recursion, where what is being "defined" is a proof instead of a piece of data.

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

```
namespace hidden
inductive empty : Type

inductive unit : Type
| star : unit

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

(To run these examples, we put them in a namespace called `hidden`, so that a name like `bool` does not conflict with the `bool` in the standard library. This is necessary because these types are part of the Lean "prelude" that is automatically imported when the system is started.)

The type `empty` is an inductive data type 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 you can define a binary operation like `band` using a case split:

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

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

7.2 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:

```
universes u v

inductive prod (α : Type u) (β : Type v)
| mk : α → β → prod

inductive sum (α : Type u) (β : Type v)
| inl : α → sum
| inr : β → sum
```

Notice that we do not include the types α and β in the target of the constructors. 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 α β`, we can assume the input is of the form `prod.mk a b`, and we have to specify the output, in terms of `a` and `b`. We can use this to define the two projections for prod. Remember that the standard library defines notation `α × β` for `prod α β` and `(a, b)` for `prod.mk a b`.

```
def fst {α : Type u} {β : Type v} (p : α × β) : α :=
prod.rec_on p (λ a b, a)

def snd {α : Type u} {β : Type v} (p : α × β) : β :=
prod.rec_on p (λ a b, b)
```

The function `fst` takes a pair, `p`. Applying the recursor `prod.rec_on p (λ 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 α and β to belong to any universe.

Here is another example:

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

#reduce prod_example (tt, 3)
#reduce prod_example (ff, 3)
```

The `cond` function is a boolean conditional: `cond b t1 t2` returns `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 α β`, we have to handle two cases: either the input is of the form `inl a`, in which case we have to specify an output value in terms of `a`, or the input is of the form `inr b`, in which case we have to specify an output value in terms of `b`.

```
def sum_example (s : ℕ ⊕ ℕ) : ℕ :=
sum.cases_on s (λ n, 2 * n) (λ n, 2 * n + 1)

#reduce sum_example (sum.inl 3)
#reduce 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`.

Notice that the product type depends on parameters `α β : Type` which are arguments to the constructors as well as `prod`. Lean detects when these arguments can be inferred from later arguments to a constructor or the return type, and makes them implicit in that case.

In 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 α β` is either of the form `inl a` *or* of the form `inl b`. A constructor with multiple arguments introduces conjunctive information: from an element `prod.mk a b` of `prod α β` we can extract `a` *and* `b`. An arbitrary inductive type can include both features, by having any number of constructors, each of which takes any number of arguments.

As with function definitions, Lean's inductive definition syntax will let you put named arguments to the constructors before the colon:

```
universes u v

inductive prod (α : Type u) (β : Type v)
| mk (fst : α) (snd : β) : prod

inductive sum (α : Type u) (β : Type v)
| inl {} (a : α) : sum
| inr {} (b : β) : sum
```

The results of these definitions are essentially the same as the ones given earlier in this section. Note that in the definition of `sum`, the annotation `{}` refers to the parameters, α and β. As with function definitions, you can use curly braces to specify which arguments are meant to be left implicit.

A type, like `prod`, that has only one constructor is purely conjunctive: the constructor simply packs the list of arguments into a single piece of data, essentially a tuple where the type of subsequent arguments can depend on the type of the initial argument. We can also think of such a type as a "record" or a "structure". In Lean, the keyword `structure` can be used to define such an inductive type as well as its projections, at the same time.

```
structure prod (α β : Type*) :=
mk :: (fst : α) (snd : β)
```

This example simultaneously introduces the inductive type, `prod`, its constructor, `mk`, the usual eliminators (`rec` and `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:

```
structure color := (red : nat) (green : nat) (blue : nat)
def yellow := color.mk 255 255 0
#reduce 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 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 Chapter 9.

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

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

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

```
inductive option (α : Type*)
| none {} : option
| some    : α → option

inductive inhabited (α : Type*)
| mk : α → inhabited
```

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

An element of `inhabited α` is simply a witness to the fact that there is an element of α. Later, we will see that `inhabited` is an example of a *type class* in Lean: Lean can be instructed that suitable base types are inhabited, and can automatically infer that other constructed types are inhabited on that basis.

As exercises, we encourage you to develop a notion of composition for partial functions from α to β and β to γ, and show that it behaves as expected. We also encourage you to show that `bool` and `nat` are inhabited, that the product of two inhabited types is inhabited, and that the type of functions to an inhabited type is inhabited.

7.3 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 `hp : 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 {α : Type*} (q : α → Prop) : Prop
| intro : ∀ (a : α), q a → Exists

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

Keep in mind that the notation `∃ x : α, p` is syntactic sugar for `Exists (λ x : α, p)`.

The definitions of `false`, `true`, `and`, and `or` are perfectly analogous to the definitions of `empty`, `unit`, `prod`, and `sum`. The difference is that the first group yields elements of `Prop`, and the second yields elements of `Type u` for some `u`. In a similar way, `∃ x : α, p` is a `Prop`-valued variant of `Σ x : α, p`.

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

```
inductive subtype {α : Type*} (p : α → Prop)
| mk : Π x : α, p x → subtype
```

In fact, in Lean, `subtype` is defined using the structure command:

```
structure subtype {α : Sort u} (p : α → Prop) :=
(val : α) (property : p val)

section
variables {α : Type u} (p : α → Prop)

#check subtype p
#check { x : α // p x}
end
```

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

7.4 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, since it describes the kind of object we wish to construct. 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
```

(continues on next page)

(continued from previous page)

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

-- try it out
#reduce 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:

```
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
```

We will explain how the `instance` command works in Chapter 10. 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.rec_on n
  (show 0 + 0 = 0, from rfl)
  (assume n,
    assume ih : 0 + n = n,
    show 0 + succ n = succ n, from
      calc
        0 + succ n = succ (0 + n) : rfl
          ... = succ n : by rw ih)
```

Notice that, once again, when `nat.rec_on` is used in the context of a proof, it is really the induction principle in disguise. The `rewrite` and `simp` tactics tend to be very effective in proofs like these. In this case, each can be used to reduce the proof to a one-liner:

```
theorem zero_add (n : ℕ) : 0 + n = n :=
nat.rec_on n rfl (λ n ih, by rw [add_succ, ih])

theorem zero_add' (n : ℕ) : 0 + n = n :=
nat.rec_on n rfl (λ n ih, by simp only [add_succ, ih])
```

The second example would be misleading without the `only` modifier, because `zero_add` is in fact declared to be a simplification rule in the standard library. Using `only` guarantees that `simp` only uses the identities listed.

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.rec_on k
  (show m + n + 0 = m + (n + 0), from rfl)
```

(continues on next page)

(continued from previous page)

```
  (assume 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 rw ih
          ... = m + succ (n + k) : rfl
          ... = m + (n + succ k) : rfl)
```

One again, there is a one-line proof:

```
theorem add_assoc (m n k : ℕ) : m + n + k = m + (n + k) :=
nat.rec_on k rfl (λ k ih, by simp only [add_succ, ih])
```

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.rec_on n
  (show m + 0 = 0 + m, by rw [nat.zero_add, nat.add_zero])
  (assume n,
    assume ih : m + n = n + m,
    calc
      m + succ n = succ (m + n) : rfl
        ... = succ (n + m) : by rw ih
        ... = succ n + m : sorry)
```

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

```
theorem succ_add (m n : nat) : succ m + n = succ (m + n) :=
nat.rec_on n
  (show succ m + 0 = succ (m + 0), from rfl)
  (assume 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 rw ih
          ... = succ (m + succ n) : rfl)
```

We can then replace the `sorry` in the previous proof with `succ_add`. Yet again, the proofs can be compressed:

```
theorem add_assoc (m n k : ℕ) : m + n + k = m + (n + k) :=
nat.rec_on k rfl (λ k ih, by simp only [add_succ, ih])

theorem succ_add (m n : nat) : succ m + n = succ (m + n) :=
nat.rec_on n rfl (λ n ih, by simp only [add_succ, ih])

theorem add_comm (m n : nat) : m + n = n + m :=
nat.rec_on n
  (by simp only [zero_add, add_zero])
  (λ n ih, by simp only [add_succ, ih, succ_add])
```

7.5 Other Recursive Data Types

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

```
inductive list (α : Type*)
| nil {} : list
| cons : α → list → list

namespace list

variable {α : Type*}

notation h :: t  := cons h t

def append (s t : list α) : list α :=
list.rec t (λ x l u, x::u) s

notation s ++ t := append s t

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

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

end list
```

A list of elements of type α is either the empty list, `nil`, or an element `h : α` followed by a list `t : list α`. 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 α` to specify the type α.

Lean allows us to define iterative notation for lists:

```
inductive list (α : Type*)
| nil {} : list
| cons : α → 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 int)
end

end list
```

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

As an exercise, prove the following:

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

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

Try also defining the function `length : Π {α : Type*}, list α → 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

def succ (t : cbtree) : cbtree :=
sup (λ n, t)

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

end cbtree
```

7.6 Tactics for Inductive Types

Given the fundamental importance of inductive types in Lean, it should not be surprising that there are a number of tactics designed to work with them effectively. We describe some of them here.

The `cases` tactic works on elements of an inductively defined type, and does what the name suggests: it decomposes the element according to each of the possible constructors. In its most basic form, it is applied to an element `x` in the local context. It then reduces the goal to cases in which `x` is replaced by each of the constructions.

```
open nat
variable p : ℕ → Prop

example (hz : p 0) (hs : ∀ n, p (succ n)) : ∀ n, p n :=
begin
  intro n,
  cases n,
  { exact hz },  -- goal is p 0
  apply hs       -- goal is a : ℕ ⊢ p (succ a)
end
```

There are extra bells and whistles. For one thing, `cases` allows you to choose the names for the arguments to the constructors using a `with` clause. In the next example, for example, we choose the name `m` for the argument to `succ`, so that the second case refers to `succ m`. More importantly, the cases tactic will detect any items in the local context that depend on the target variable. It reverts these elements, does the split, and reintroduces them. In the example below, notice that the hypothesis `h : n ≠ 0` becomes `h : 0 ≠ 0` in the first branch, and `h : succ m ≠ 0` in the second.

```
open nat

example (n : ℕ) (h : n ≠ 0) : succ (pred n) = n :=
begin
  cases n with m,
  -- first goal: h : 0 ≠ 0 ⊢ succ (pred 0) = 0
    { apply (absurd rfl h) },
  -- second goal: h : succ m ≠ 0 ⊢ succ (pred (succ m)) = succ m
  reflexivity
end
```

Notice that `cases` can be used to produce data as well as prove propositions.

```
def f (n : ℕ) : ℕ :=
begin
  cases n, exact 3, exact 7
end

example : f 0 = 3 := rfl
example : f 5 = 7 := rfl
```

Once again, cases will revert, split, and then reintroduce depedencies in the context.

```
def tuple (α : Type*) (n : ℕ) :=
  { l : list α // list.length l = n }

variables {α : Type*} {n : ℕ}

def f {n : ℕ} (t : tuple α n) : ℕ :=
begin
  cases n, exact 3, exact 7
end

def my_tuple : tuple ℕ 3 := ⟨[0, 1, 2], rfl⟩

example : f my_tuple = 7 := rfl
```

If there are multiple constructors with arguments, you can provide `cases` with a list of all the names, arranged sequentially:

```
inductive foo : Type
| bar1 : ℕ → ℕ → foo
| bar2 : ℕ → ℕ → ℕ → foo

def silly (x : foo) : ℕ :=
begin
  cases x with a b c d e,
  exact b,    -- a, b    are in the context
  exact e     -- c, d, e are in the context
end
```

The syntax of the `with` is unfortunate, in that we have to list the arguments to all the constructors sequentially, making it hard to remember what the constructors are, or what the arguments are supposed to be. For that reason, Lean provides a complementary `case` tactic, which allows one to assign variable names after the fact:

```
inductive foo : Type
| bar1 : ℕ → ℕ → foo
| bar2 : ℕ → ℕ → ℕ → foo

open foo

def silly (x : foo) : ℕ :=
begin
  cases x,
    case bar1 : a b
      { exact b },
    case bar2 : c d e
      { exact e }
end
```

The `case` tactic is clever, in that it will match the constructor to the appropriate goal. For example, we can fill the goals above in the opposite order:

```
inductive foo : Type
| bar1 : ℕ → ℕ → foo
| bar2 : ℕ → ℕ → ℕ → foo

open foo

def silly (x : foo) : ℕ :=
begin
  cases x,
    case bar2 : c d e
      { exact e },
    case bar1 : a b
      { exact b }
end
```

You can also use `cases` with an arbitrary expression. Assuming that expression occurs in the goal, the cases tactic will generalize over the expression, introduce the resulting universally quantified variable, and case on that.

```
open nat
variable p : ℕ → Prop

example (hz : p 0) (hs : ∀ n, p (succ n)) (m k : ℕ) :
  p (m + 3 * k) :=
begin
  cases (m + 3 * k),
  { exact hz },   -- goal is p 0
  apply hs        -- goal is a : ℕ ⊢ p (succ a)
end
```

Think of this as saying "split on cases as to whether `m + 3 * k` is zero or the successor of some number." The result is functionally equivalent to the following:

```
example (hz : p 0) (hs : ∀ n, p (succ n)) (m k : ℕ) :
  p (m + 3 * k) :=
begin
  generalize : m + 3 * k = n,
  cases n,
  { exact hz },   -- goal is p 0
  apply hs        -- goal is a : ℕ ⊢ p (succ a)
```

(continues on next page)

(continued from previous page)

```
end
```

Notice that the expression `m + 3 * k` is erased by generalize; all that matters is whether it is of the form `0` or `succ a`. This form of `cases` will *not* revert any hypotheses that also mention the expression in equation (in this case, `m + 3 * k`). If such a term appears in a hypothesis and you want to generalize over that as well, you need to `revert` it explicitly.

If the expression you case on does not appear in the goal, the `cases` tactic uses `have` to put the type of the expression into the context. Here is an example:

```
example (p : Prop) (m n : ℕ)
  (h₁ : m < n → p) (h₂ : m ≥ n → p) : p :=
begin
  cases lt_or_ge m n with hlt hge,
  { exact h₁ hlt },
  exact h₂ hge
end
```

The theorem `lt_or_ge m n` says `m < n ∨ m ≥ n`, and it is natural to think of the proof above as splitting on these two cases. In the first branch, we have the hypothesis `h₁ : m < n`, and in the second we have the hypothesis `h₂ : m ≥ n`. The proof above is functionally equivalent to the following:

```
example (p : Prop) (m n : ℕ)
  (h₁ : m < n → p) (h₂ : m ≥ n → p) : p :=
begin
  have h : m < n ∨ m ≥ n,
  { exact lt_or_ge m n },
  cases h with hlt hge,
  { exact h₁ hlt },
  exact h₂ hge
end
```

After the first two lines, we have `h : m < n ∨ m ≥ n` as a hypothesis, and we simply do cases on that.

Here is another example, where we use the decidability of equality on the natural numbers to split on the cases `m = n` and `m ≠ n`.

```
#check nat.sub_self

example (m n : ℕ) : m - n = 0 ∨ m ≠ n :=
begin
  cases decidable.em (m = n) with heq hne,
  { rw heq,
    left, exact nat.sub_self n },
  right, exact hne
end
```

Remember that if you `open classical`, you can use the law of the excluded middle for any proposition at all. But using type class inference (see Chapter 10), Lean can actually find the relevant decision procedure, which means that you can use the case split in a computable function.

```
def f (m k : ℕ) : ℕ :=
begin
  cases m - k, exact 3, exact 7
end

example : f 5 7 = 3 := rfl
example : f 10 2 = 7 := rfl
```

Aspects of computability will be discussed in Chapter 11.

Just as the `cases` tactic can be used to carry out proof by cases, the `induction` tactic can be used to carry out proofs by induction. The syntax is similar to that of `cases`, except that the argument can only be a term in the local context. Here is an example:

```
theorem zero_add (n : ℕ) : 0 + n = n :=
begin
  induction n with n ih,
    refl,
  rw [add_succ, ih]
end
```

As with `cases`, we can use the `case` tactic instead to identify one case at a time and name the arguments:

```
theorem zero_add (n : ℕ) : 0 + n = n :=
begin
  induction n,
  case zero : { refl },
  case succ : n ih { rw [add_succ, ih]}
end

theorem succ_add (m n : ℕ) : succ m + n = succ (m + n) :=
begin
  induction n,
  case zero : { refl },
  case succ : n ih { rw [add_succ, ih] }
end

theorem add_comm (m n : ℕ) : m + n = n + m :=
begin
  induction n,
  case zero : { rw zero_add, refl },
  case succ : n ih { rw [add_succ, ih, succ_add] }
end
```

The name before the colon corresponds to the constructor of the associated inductive type. The cases can appear in any order, and when there are no parameters to rename (for example, as in the `zero` cases above) the colon can be omitted. Once again, we can reduce the proofs of these, as well as the proof of associativity, to one-liners.

```
theorem zero_add (n : ℕ) : 0 + n = n :=
by induction n; simp only [*, add_zero, add_succ]

theorem succ_add (m n : ℕ) : succ m + n = succ (m + n) :=
by induction n; simp only [*, add_zero, add_succ]

theorem add_comm (m n : ℕ) : m + n = n + m :=
by induction n;
     simp only [*, add_zero, add_succ, succ_add, zero_add]

theorem add_assoc (m n k : ℕ) : m + n + k = m + (n + k) :=
by induction k; simp only [*, add_zero, add_succ]
```

We close this section with one last tactic that is designed to facilitate working with inductive types, namely, the `injection` tactic. By design, the elements of an inductive type are freely generated, which is to say, the constructors are injective and have disjoint ranges. The `injection` tactic is designed to make use of this fact:

```
open nat

example (m n k : ℕ) (h : succ (succ m) = succ (succ n)) :
  n + k = m + k :=
begin
  injection h with h',
  injection h' with h'',
  rw h''
end
```

The first instance of the tactic adds `h' : succ m = succ n` to the context, and the second adds `h'' : m = n`. The plural variant, `injections`, applies `injection` to all hypotheses repeatedly. It still allows you to name the results using `with`.

```
example (m n k : ℕ) (h : succ (succ m) = succ (succ n)) :
  n + k = m + k :=
begin
  injections with h' h'',
  rw h''
end

example (m n k : ℕ) (h : succ (succ m) = succ (succ n)) :
  n + k = m + k :=
by injections; simp *
```

The `injection` and `injections` tactics will also detect contradictions that arise when different constructors are set equal to one another, and use them to close the goal.

```
example (m n : ℕ) (h : succ m = 0) : n = n + 7 :=
by injections

example (m n : ℕ) (h : succ m = 0) : n = n + 7 :=
by contradiction

example (h : 7 = 4) : false :=
by injections
```

As the second example shows, the `contradiction` tactic also detects contradictions of this form. But the `contradiction` tactic does not solve the third goal, while `injections` does.

7.7 Inductive Families

We are almost done describing the full range of inductive definitions accepted by Lean. So far, you have seen that Lean allows you to introduce inductive types with any number of recursive constructors. In fact, a single inductive definition can introduce an indexed *family* of inductive types, in a manner we now describe.

An inductive family is an indexed family of types defined by a simultaneous induction of the following form:

```
inductive foo : ... → Sort u :=
| constructor₁ : ... → foo ...
| constructor₂ : ... → foo ...
...
| constructorₙ : ... → foo ...
```

In contrast to ordinary inductive definition, which constructs an element of some `Sort u`, the more general version constructs a function `... → Sort u`, where "`...`" denotes a sequence of argument types, also known as *indices*.

Each constructor then constructs an element of some member of the family. One example is the definition of `vector α n`, the type of vectors of elements of α of length `n`:

```
inductive vector (α : Type u) : nat → Type u
| nil {}                              : vector zero
| cons {n : ℕ} (a : α) (v : vector n) : vector (succ n)
```

Notice that the `cons` constructor takes an element of `vector α n` and returns an element of `vector α (succ n)`, thereby using an element of one member of the family to build an element of another.

A more exotic example is given by the definition of the equality type in Lean:

```
inductive eq {α : Sort u} (a : α) : α → Prop
| refl [] : eq a
```

For each fixed `α : Sort u` and `a : α`, this definition constructs a family of types `eq a x`, indexed by `x : α`. Notably, however, there is only one constructor, `refl`, which is an element of `eq a a`, and the square brackets after the constructor tell Lean to make the argument to `refl` explicit. Intuitively, the only way to construct a proof of `eq a x` is to use reflexivity, in the case where `x` is `a`. Note that `eq a a` is the only inhabited type in the family of types `eq a x`. The elimination principle generated by Lean is as follows:

```
universes u v

#check (@eq.rec_on :
  Π {α : Sort u} {a : α} {C : α → Sort v} {b : α},
    a = b → C a → C b)
```

It is a remarkable fact that all the basic axioms for equality follow from the constructor, `refl`, and the eliminator, `eq.rec_on`. The definition of equality is atypical, however; see the discussion in the next section.

The recursor `eq.rec_on` is also used to define substitution:

```
@[elab_as_eliminator]
theorem subst {α : Type u} {a b : α} {p : α → Prop}
  (h₁ : eq a b) (h₂ : p a) : p b :=
eq.rec h₂ h₁
```

Using the recursor with `h₁ : a = b`, we may assume `a` and `b` are the same, in which case, `p b` and `p a` are the same. The definition of `subst` is marked with an elaboration hint, as described in Section 6.10.

It is not hard to prove that `eq` is symmetric and transitive. In the following example, we prove `symm` and leave as exercise the theorems `trans` and `congr` (congruence).

```
theorem symm {α : Type u} {a b : α} (h : eq a b) : eq b a :=
subst h (eq.refl a)

theorem trans {α : Type u} {a b c : α}
  (h₁ : eq a b) (h₂ : eq b c) : eq a c :=
sorry

theorem congr {α β : Type u} {a b : α} (f : α → β)
  (h : eq a b) : eq (f a) (f b) :=
sorry
```

In the type theory literature, there are further generalizations of inductive definitions, for example, the principles of *induction-recursion* and *induction-induction*. These are not supported by Lean.

7.8 Axiomatic Details

We have described inductive types and their syntax through examples. This section provides additional information for those interested in the axiomatic foundations.

We have seen that the constructor to an inductive type takes *parameters* — intuitively, the arguments that remain fixed throughout the inductive construction — and *indices*, the arguments parameterizing the family of types that is simultaneously under construction. Each constructor should have a Pi type, where the argument types are built up from previously defined types, the parameter and index types, and the inductive family currently being defined. The requirement is that if the latter is present at all, it occurs only *strictly positively*. This means simply that any argument to the constructor in which it occurs is a Pi type in which the inductive type under definition occurs only as the resulting type, where the indices are given in terms of constants and previous arguments.

Since an inductive type lives in `Sort u` for some `u`, it is reasonable to ask *which* universe levels `u` can be instantiated to. Each constructor `c` in the definition of a family `C` of inductive types is of the form

```
c : Π (a : α) (b : β[a]), C a p[a,b]
```

where `a` is a sequence of data type parameters, `b` is the sequence of arguments to the constructors, and `p[a, b]` are the indices, which determine which element of the inductive family the construction inhabits. (Note that this description is somewhat misleading, in that the arguments to the constructor can appear in any order as long as the dependencies make sense.) The constraints on the universe level of `C` fall into two cases, depending on whether or not the inductive type is specified to land in `Prop` (that is, `Sort 0`).

Let us first consider the case where the inductive type is *not* specified to land in `Prop`. Then the universe level `u` is constrained to satisfy the following:

> For each constructor `c` as above, and each `βk[a]` in the sequence `β[a]`, if `βk[a] : Sort v`, we have `u ≥ v`.

In other words, the universe level `u` is required to be at least as large as the universe level of each type that represents an argument to a constructor.

When the inductive type is specified to land in `Prop`, there are no constraints on the universe levels of the constructor arguments. But these universe levels do have a bearing on the elimination rule. Generally speaking, for an inductive type in `Prop`, the motive of the elimination rule is required to be in `Prop`.

There is an exception to this last rule: we are allowed to eliminate from an inductively defined `Prop` to an arbitrary `Sort` when there is only one constructor and each constructor argument is either in `Prop` or an index. The intuition is that in this case the elimination does not make use of any information that is not already given by the mere fact that the type of argument is inhabited. This special case is known as *singleton elimination*.

We have already seen singleton elimination at play in applications of `eq.rec`, the eliminator for the inductively defined equality type. We can use an element `h : eq a b` to cast an element `t' : p a` to `p b` even when `p a` and `p b` are arbitrary types, because the cast does not produce new data; it only reinterprets the data we already have. Singleton elimination is also used with heterogeneous equality and well-founded recursion, which will be discussed in a later chapter.

7.9 Mutual and Nested Inductive Types

We now consider two generalizations of inductive types that are often useful, which Lean supports by "compiling" them down to the more primitive kinds of inductive types described above. In other words, Lean parses the more general definitions, defines auxiliary inductive types based on them, and then uses the auxiliary types to define the ones we really want. Lean's equation compiler, described in the next chapter, is needed to make use of these types effectively. Nonetheless, it makes sense to describe the declarations here, because they are straightforward variations on ordinary inductive definitions.

First, Lean supports *mutually defined* inductive types. The idea is that we can define two (or more) inductive types at the same time, where each one refers to the other(s).

```
mutual inductive even, odd
with even : ℕ → Prop
| even_zero : even 0
| even_succ : ∀ n, odd n → even (n + 1)
with odd : ℕ → Prop
| odd_succ : ∀ n, even n → odd (n + 1)
```

In this example, two types are defined simultaneously: a natural number `n` is `even` if it is `0` or one more than an `odd` number, and `odd` if it is one more than an even number. Under the hood, this definition is compiled down to a single inductive type with an index `i` in a two-valued type (such as `bool`), where `i` encodes which of `even` or `odd` is intended. In the exercises below, you are asked to spell out the details.

A mutual inductive definition can also be used to define the notation of a finite tree with nodes labeled by elements of α:

```
universe u

mutual inductive tree, list_tree (α : Type u)
with tree : Type u
| node : α → list_tree → tree
with list_tree : Type u
| nil {} : list_tree
| cons    : tree → list_tree → list_tree
```

With this definition, one can construct an element of `tree α` by giving an element of α together with a list of subtrees, possibly empty. The list of subtrees is represented by the type `list_tree α`, which is defined to be either the empty list, `nil`, or the `cons` of a tree and an element of `list_tree α`.

This definition is inconvenient to work with, however. It would be much nicer if the list of subtrees were given by the type `list (tree α)`, especially since Lean's library contains a number of functions and theorems for working with lists. One can show that the type `list_tree α` is *isomorphic* to `list (tree α)`, but translating results back and forth along this isomorphism is tedious.

In fact, Lean allows us to define the inductive type we really want:

```
inductive tree (α : Type u)
| mk : α → list tree → tree
```

This is known as a *nested* inductive type. It falls outside the strict specification of an inductive type given in the last section because `tree` does not occur strictly positively among the arguments to `mk`, but, rather, nested inside the `list` type constructor. Under the hood, Lean compiles this down to the mutual inductive type described above, which, in turn, is compiled down to an ordinary inductive type. Lean then automatically builds the isomorphism between `list_tree α` and `list (tree α)`, and defines the constructors for `tree` in terms of the isomorphism.

The types of the constructors for mutual and nested inductive types can be read off from the definitions. Defining functions *from* such types is more complicated, because these also have to be compiled down to more basic operations, making use of the primitive recursors that are associated to the inductive types that are declared under the hood. Lean does its best to hide the details from users, allowing them to use the equation compiler, described in the next section, to define such functions in natural ways.

7.10 Exercises

1. Try defining other operations on the natural numbers, such as multiplication, the predecessor function (with `pred 0 = 0`), truncated subtraction (with `n - m = 0` when `m` is greater than or equal to `n`), and exponentiation. Then try proving some of their basic properties, building on the theorems we have already proved.

   Since many of these are already defined in Lean's core library, you should work within a namespace named `hide`, or something like that, in order to avoid name clashes.

2. Define some operations on lists, like a `length` function or the `reverse` function. Prove some properties, such as the following:

   a. `length (s ++ t) = length s + length t`

   b. `length (reverse t) = length t`

   c. `reverse (reverse t) = t`

3. Define an inductive data type consisting of terms built up from the following constructors:

   - `const n`, a constant denoting the natural number `n`
   - `var n`, a variable, numbered `n`
   - `plus s t`, denoting the sum of `s` and `t`
   - `times s t`, denoting the product of `s` and `t`

   Recursively define a function that evaluates any such term with respect to an assignment of values to the variables.

4. Similarly, define the type of propositional formulas, as well as functions on the type of such formulas: an evaluation function, functions that measure the complexity of a formula, and a function that substitutes another formula for a given variable.

5. Simulate the mutual inductive definition of `even` and `odd` described in Section 7.9 with an ordinary inductive type, using an index to encode the choice between them in the target type.

CHAPTER

EIGHT

INDUCTION AND RECURSION

In the previous chapter, we saw that inductive definitions provide a powerful means of introducing new types in Lean. Moreover, the constructors and the recursors provide the only means of defining functions on these types. By the propositions-as-types correspondence, this means that induction is the fundamental method of proof.

Lean provides natural ways of defining recursive functions, performing pattern matching, and writing inductive proofs. It allows you to define a function by specifying equations that it should satisfy, and it allows you to prove a theorem by specifying how to handle various cases that can arise. Behind the scenes, these descriptions are "compiled" down to primitive recursors, using a procedure that we refer to as the "equation compiler." The equation compiler is not part of the trusted code base; its output consists of terms that are checked independently by the kernel.

8.1 Pattern Matching

The interpretation of schematic patterns is the first step of the compilation process. We have seen that the `cases_on` recursor can be used to define functions and prove theorems by cases, according to the constructors involved in an inductively defined type. But complicated definitions may use several nested `cases_on` applications, and may be hard to read and understand. Pattern matching provides an approach that is more convenient, and familiar to users of functional programming languages.

Consider the inductively defined type of natural numbers. Every natural number is either `zero` or `succ x`, and so you can define a function from the natural numbers to an arbitrary type by specifying a value in each of those cases:

```
open nat

def sub1 : ℕ → ℕ
| zero     := zero
| (succ x) := x

def is_zero : ℕ → Prop
| zero     := true
| (succ x) := false
```

The equations used to define these function hold definitionally:

```
example : sub1 0 = 0 := rfl
example (x : ℕ) : sub1 (succ x) = x := rfl

example : is_zero 0 = true := rfl
example (x : ℕ) : is_zero (succ x) = false := rfl

example : sub1 7 = 6 := rfl
example (x : ℕ) : ¬ is_zero (x + 3) := not_false
```

Instead of zero and succ, we can use more familiar notation:

```
open nat

def sub1 : ℕ → ℕ
| 0      := 0
| (x+1)  := x

def is_zero : ℕ → Prop
| 0      := true
| (x+1)  := false
```

Because addition and the zero notation have been assigned the [pattern] attribute, they can be used in pattern matching. Lean simply normalizes these expressions until the constructors zero and succ are exposed.

Pattern matching works with any inductive type, such as products and option types:

```
universes u v
variables {α : Type u} {β : Type v}

def swap_pair : α × β → β × α
| (a, b) := (b, a)

def foo : ℕ × ℕ → ℕ
| (m, n) := m + n

def bar : option ℕ → ℕ
| (some n) := n + 1
| none     := 0
```

Here we use it not only to define a function, but also to carry out a proof by cases:

```
def bnot : bool → bool
| tt := ff
| ff := tt

theorem bnot_bnot : ∀ (b : bool), bnot (bnot b) = b
| tt := rfl    -- proof that bnot (bnot tt) = tt
| ff := rfl    -- proof that bnot (bnot ff) = ff
```

Pattern matching can also be used to destruct inductively defined propositions:

```
example (p q : Prop) : p ∧ q → q ∧ p
| (and.intro h₁ h₂) := and.intro h₂ h₁

example (p q : Prop) : p ∨ q → q ∨ p
| (or.inl hp) := or.inr hp
| (or.inr hq) := or.inl hq
```

This provides a compact way of unpacking hypotheses that make use of logical connectives.

In all these examples, pattern matching was used to carry out a single case distinction. More interestingly, patterns can involve nested constructors, as in the following examples.

```
open nat

def sub2 : ℕ → ℕ
| zero              := 0
```

(continues on next page)

(continued from previous page)

```
| (succ zero)      := 0
| (succ (succ a)) := a
```

The equation compiler first splits on cases as to whether the input is `zero` or of the form `succ x`. It then does a case split on whether `x` is of the form `zero` or `succ a`. It determines the necessary case splits from the patterns that are presented to it, and raises and error if the patterns fail to exhaust the cases. Once again, we can use arithmetic notation, as in the version below. In either case, the defining equations hold definitionally.

```
def sub2 : ℕ → ℕ
| 0     := 0
| 1     := 0
| (a+2) := a

example : sub2 0 = 0 := rfl
example : sub2 1 = 0 := rfl
example (a : nat) : sub2 (a + 2) = a := rfl

example : sub2 5 = 3 := rfl
```

You can write `#print sub2` to see how the function was compiled to recursors. (Lean will tell you that `sub2` has been defined in terms of an internal auxiliary function, `sub2._main`, but you can print that out too.)

Here are some more examples of nested pattern matching:

```
example {α : Type*} (p q : α → Prop) :
  (∃ x, p x ∨ q x) → (∃ x, p x) ∨ (∃ x, q x)
| (exists.intro x (or.inl px)) := or.inl (exists.intro x px)
| (exists.intro x (or.inr qx)) := or.inr (exists.intro x qx)

def foo : ℕ × ℕ → ℕ
| (0, n)     := 0
| (m+1, 0)   := 1
| (m+1, n+1) := 2
```

The equation compiler can process multiple arguments sequentially. For example, it would be more natural to define the previous example as a function of two arguments:

```
def foo : ℕ → ℕ → ℕ
| 0     n     := 0
| (m+1) 0     := 1
| (m+1) (n+1) := 2
```

Here is another example:

```
def bar : list ℕ → list ℕ → ℕ
| []       []       := 0
| (a :: l) []       := a
| []       (b :: l) := b
| (a :: l) (b :: m) := a + b
```

Note that, with compound expressions, parentheses are used to separate the arguments.

In each of the following examples, splitting occurs on only the first argument, even though the others are included among the list of patterns.

```
def band : bool → bool → bool
| tt a := a
```

(continues on next page)

(continued from previous page)

```
| ff _ := ff

def bor : bool → bool → bool
| tt _ := tt
| ff a := a

def {u} cond {a : Type u} : bool → a → a → a
| tt x y := x
| ff x y := y
```

Notice also that, when the value of an argument is not needed in the definition, you can use an underscore instead. This underscore is known as a *wildcard pattern*, or an *anonymous variable*. In contrast to usage outside the equation compiler, here the underscore does *not* indicate an implicit argument. The use of underscores for wildcards is common in functional programming languages, and so Lean adopts that notation. Section 8.2 expands on the notion of a wildcard, and Section 8.7 explains how you can use implicit arguments in patterns as well.

As described in Chapter 7, inductive data types can depend on parameters. The following example defines the `tail` function using pattern matching. The argument `α : Type` is a parameter and occurs before the colon to indicate it does not participate in the pattern matching. Lean also allows parameters to occur after `:`, but it cannot pattern match on them.

```
def tail1 {α : Type*} : list α → list α
| []       := []
| (h :: t) := t

def tail2 : Π {α : Type*}, list α → list α
| α []       := []
| α (h :: t) := t
```

Despite the different placement of the parameter α in these two examples, in both cases it treated in the same way, in that it does not participate in a case split.

Lean can also handle more complex forms of pattern matching, in which arguments to dependent types pose additional constraints on the various cases. Such examples of *dependent pattern matching* are considered in Section 8.6.

8.2 Wildcards and Overlapping Patterns

Consider one of the examples from the last section:

```
def foo : ℕ → ℕ → ℕ
| 0     n     := 0
| (m+1) 0     := 1
| (m+1) (n+1) := 2
```

The example can be written more concisely:

```
def foo : ℕ → ℕ → ℕ
| 0 n := 0
| m 0 := 1
| m n := 2
```

In the second presentation, the patterns overlap; for example, the pair of arguments `0 0` matches all three cases. But Lean handles the ambiguity by using the first applicable equation, so the net result is the same. In particular, the following equations hold definitionally:

```
variables (m n : nat)

example : foo 0     0     = 0 := rfl
example : foo 0     (n+1) = 0 := rfl
example : foo (m+1) 0     = 1 := rfl
example : foo (m+1) (n+1) = 2 := rfl
```

Since the values of m and n are not needed, we can just as well use wildcard patterns instead.

```
def foo : ℕ → ℕ → ℕ
| 0 _ := 0
| _ 0 := 1
| _ _ := 2
```

You can check that this definition of foo satisfies the same definitional identities as before.

Some functional programming languages support *incomplete patterns*. In these languages, the interpreter produces an exception or returns an arbitrary value for incomplete cases. We can simulate the arbitrary value approach using the inhabited type class. Roughly, an element of inhabited α is a witness to the fact that there is an element of α; in Chapter 10 we will see that Lean can be instructed that suitable base types are inhabited, and can automatically infer that other constructed types are inhabited on that basis. On this basis, the standard library provides an arbitrary element, arbitrary α, of any inhabited type.

We can also use the type option α to simulate incomplete patterns. The idea is to return some a for the provided patterns, and use none for the incomplete cases. The following example demonstrates both approaches.

```
def f1 : ℕ → ℕ → ℕ
| 0  _  := 1
| _  0  := 2
| _  _  := arbitrary ℕ   -- the "incomplete" case

variables (a b : ℕ)

example : f1 0     0     = 1 := rfl
example : f1 0     (a+1) = 1 := rfl
example : f1 (a+1) 0     = 2 := rfl
example : f1 (a+1) (b+1) = arbitrary nat := rfl

def f2 : ℕ → ℕ → option ℕ
| 0  _  := some 1
| _  0  := some 2
| _  _  := none          -- the "incomplete" case

example : f2 0     0     = some 1 := rfl
example : f2 0     (a+1) = some 1 := rfl
example : f2 (a+1) 0     = some 2 := rfl
example : f2 (a+1) (b+1) = none   := rfl
```

The equation compiler is clever. If you leave out any of the cases in the following definition, the error message will let you know what has not been covered.

```
def bar : ℕ → list ℕ → bool → ℕ
| 0       _        ff := 0
| 0       (b :: _) _  := b
| 0       []       tt := 7
| (a+1)  []       ff := a
| (a+1)  []       tt := a + 1
| (a+1)  (b :: _) _  := a + b
```

It will also use an "if … then … else" instead of a `cases_on` in appropriate situations.

```
def foo : char → ℕ
| 'A' := 1
| 'B' := 2
| _   := 3

#print foo._main
```

8.3 Structural Recursion and Induction

What makes the equation compiler powerful is that it also supports recursive definitions. In the next three sections, we will describe, respectively:

- structurally recursive definitions
- well-founded recursive definitions
- mutually recursive definitions

Generally speaking, the equation compiler processes input of the following form:

```
def foo (a : α) : Π (b : β), γ
| [patterns₁] := t₁
...
| [patternsₙ] := tₙ
```

Here `(a : α)` is a sequence of parameters, `(b : β)` is the sequence of arguments on which pattern matching takes place, and γ is any type, which can depend on `a` and `b`. Each line should contain the same number of patterns, one for each element of β. As we have seen, a pattern is either a variable, a constructor applied to other patterns, or an expression that normalizes to something of that form (where the non-constructors are marked with the `[pattern]` attribute). The appearances of constructors prompt case splits, with the arguments to the constructors represented by the given variables. In Section 8.6, we will see that it is sometimes necessary to include explicit terms in patterns that are needed to make an expression type check, though they do not play a role in pattern matching. These are called "inaccessible terms," for that reason. But we will not need to use such inaccessible terms before Section 8.6.

As we saw in the last section, the terms $t_1, \ldots, t_n$ can make use of any of the parameters `a`, as well as any of the variables that are introduced in the corresponding patterns. What makes recursion and induction possible is that they can also involve recursive calls to `foo`. In this section, we will deal with *structural recursion*, in which the arguments to `foo` occurring on the right-hand side of the `:=` are subterms of the patterns on the left-hand side. The idea is that they are structurally smaller, and hence appear in the inductive type at an earlier stage. Here are some examples of structural recursion from the last chapter, now defined using the equation compiler:

```
def add : nat → nat → nat
| m zero     := m
| m (succ n) := succ (add m n)

local infix ` + ` := add

theorem add_zero (m : nat) : m + zero = m := rfl
theorem add_succ (m n : nat) : m + succ n = succ (m + n) := rfl

theorem zero_add : ∀ n, zero + n = n
| zero     := rfl
| (succ n) := congr_arg succ (zero_add n)
```

(continues on next page)

(continued from previous page)

```
def mul : nat → nat → nat
| n zero     := zero
| n (succ m) := mul n m + n
```

The proof of `zero_add` makes it clear that proof by induction is really a form of induction in Lean.

The example above shows that the defining equations for `add` hold definitionally, and the same is true of `mul`. The equation compiler tries to ensure that this holds whenever possible, as is the case with straightforward structural induction. In other situations, however, reductions hold only *propositionally*, which is to say, they are equational theorems that must be applied explicitly. The equation compiler generates such theorems internally. They are not meant to be used directly by the user; rather, the *simp* and *rewrite* tactics are configured to use them when necessary. Thus both of the following proofs of *zero_add* work:

```
theorem zero_add : ∀ n, zero + n = n
| zero     := by simp [add]
| (succ n) := by simp [add, zero_add n]

theorem zero_add' : ∀ n, zero + n = n
| zero     := by rw [add]
| (succ n) := by rw [add, zero_add' n]
```

In fact, because in this case the defining equations hold definitionally, we can use *dsimp*, the simplifier that uses definitional reductions only, to carry out the first step.

```
theorem zero_add : ∀ n, zero + n = n
| zero     := by dsimp [add]; reflexivity
| (succ n) := by dsimp [add]; rw [zero_add n]
```

As with definition by pattern matching, parameters to a structural recursion or induction may appear before the colon. Such parameters are simply added to the local context before the definition is processed. For example, the definition of addition may also be written as follows:

```
def add (m : nat) : nat → nat
| zero     := m
| (succ n) := succ (add n)
```

This may seem a little odd, but you should read the definition as follows: "Fix `m`, and define the function which adds something to `m` recursively, as follows. To add zero, return `m`. To add the successor of `n`, first add `n`, and then take the successor." The mechanism for adding parameters to the local context is what makes it possible to process match expressions within terms, as described in Section 8.8.

A more interesting example of structural recursion is given by the Fibonacci function `fib`.

```
def fib : nat → nat
| 0     := 1
| 1     := 1
| (n+2) := fib (n+1) + fib n

example : fib 0 = 1 := rfl
example : fib 1 = 1 := rfl
example (n : nat) : fib (n + 2) = fib (n + 1) + fib n := rfl

example : fib 7 = 21 := rfl
example : fib 7 = 21 :=
begin
  dsimp [fib],   -- expands fib 7 as a sum of 1's
```

(continues on next page)

(continued from previous page)

```
  reflexivity
end
```

Here, the value of the fib function at n + 2 (which is definitionally equal to succ (succ n)) is defined in terms of the values at n + 1 (which is definitionally equivalent to succ n) and the value at n. This is a notoriously inefficient way of computing the fibonacci function, however, with an execution time that is exponential in n. Here is a better way:

```
def fib_aux : nat → nat × nat
| 0       := (0, 1)
| (n + 1) := let p := fib_aux n in (p.2, p.1 + p.2)

def fib (n : nat) := (fib_aux n).1

#eval fib 100
```

Another good example of a recursive definition is the list append function.

```
def append {α : Type*} : list α → list α → list α
| []     l := l
| (h::t) l := h :: append t l

example : append [(1 : ℕ), 2, 3] [4, 5] = [1, 2, 3, 4, 5] := rfl
```

Here is another: it adds elements of the first list to elements of the second list, until one of the two lists runs out.

```
def {u} list_add {α : Type u} [has_add α] :
  list α → list α → list α
| []       _        := []
| _        []       := []
| (a :: l) (b :: m) := (a + b) :: list_add l m

#eval list_add [1, 2, 3] [4, 5, 6, 6, 9, 10]
```

You are encouraged to experiment with similar examples in the exercises below.

8.4 Well-Founded Recursion and Induction

Dependent type theory is powerful enough to encode and justify well-founded recursion. Let us start with the logical background that is needed to understand how it works.

Lean's standard library defines two predicates, acc r a and well_founded r, where r is a binary relation on a type α, and a is an element of type α.

```
universe u
variable α : Sort u
variable r : α → α → Prop

#check (acc r : α → Prop)

#check (well_founded r : Prop)
```

The first, acc, is an inductively defined predicate. According to its definition, acc r x is equivalent to ∀ y, r y x → acc r y. If you think of r y x as denoting a kind of order relation y ≺ x, then acc r x says that x is accessible from below, in the sense that all its predecessors are accessible. In particular, if x has no predecessors, it is accessible. Given any type α, we should be able to assign a value to each accessible element of α, recursively, by assigning values to all its predecessors first.

The statement that `r` is well founded, denoted `well_founded r`, is exactly the statement that every element of the type is accessible. By the above considerations, if `r` is a well-founded relation on a type α, we should have a principle of well-founded recursion on α, with respect to the relation `r`. And, indeed, we do: the standard library defines `well_founded.fix`, which serves exactly that purpose.

```
universes u v
variable α : Sort u
variable r : α → α → Prop
variable h : well_founded r

variable C : α → Sort v
variable F : Π x, (Π (y : α), r y x → C y) → C x

def f : Π (x : α), C x := well_founded.fix h F
```

There is a long cast of characters here, but the first block we have already seen: the type, α, the relation, `r`, and the assumption, `h`, that `r` is well founded. The variable `C` represents the motive of the recursive definition: for each element `x : α`, we would like to construct an element of `C x`. The function `F` provides the inductive recipe for doing that: it tells us how to construct an element `C x`, given elements of `C y` for each predecessor `y` of `x`.

Note that `well_founded.fix` works equally well as an induction principle. It says that if $\prec$ is well founded and you want to prove `∀ x, C x`, it suffices to show that for an arbitrary `x`, if we have `∀ y ≺ x, C y`, then we have `C x`.

Lean knows that the usual order < on the natural numbers is well founded. It also knows a number of ways of constructing new well founded orders from others, for example, using lexicographic order.

Here is essentially the definition of division on the natural numbers that is found in the standard library.

```
open nat

def div_rec_lemma {x y : ℕ} : 0 < y ∧ y ≤ x → x - y < x :=
λ h, sub_lt (lt_of_lt_of_le h.left h.right) h.left

def div.F (x : ℕ) (f : Π x₁, x₁ < x → ℕ → ℕ) (y : ℕ) : ℕ :=
if h : 0 < y ∧ y ≤ x then
  f (x - y) (div_rec_lemma h) y + 1
else
  zero

def div := well_founded.fix lt_wf div.F
```

The definition is somewhat inscrutable. Here the recursion is on `x`, and `div.F x f : ℕ → ℕ` returns the "divide by `y`" function for that fixed `x`. You have to remember that the second argument to `div.F`, the recipe for the recursion, is a function that is supposed to return the divide by `y` function for all values x_1 smaller than `x`.

The equation compiler is designed to make definitions like this more convenient. It accepts the following:

```
def div : ℕ → ℕ → ℕ
| x y :=
  if h : 0 < y ∧ y ≤ x then
    have x - y < x,
      from sub_lt (lt_of_lt_of_le h.left h.right) h.left,
    div (x - y) y + 1
  else
    0
```

When the equation compiler encounters a recursive definition, it first tries structural recursion, and only when that fails, does it fall back on well-founded recursion. In this case, detecting the possibility of well-founded recursion on the natural numbers, it uses the usual lexicographic ordering on the pair `(x, y)`. The equation compiler in and of itself is not clever enough to derive that `x - y` is less than `x` under the given hypotheses, but we can help it out by putting this fact in the local context. The equation compiler looks in the local context for such information, and, when it finds it, puts it to good use.

The defining equation for `div` does *not* hold definitionally, but the equation is available to `rewrite` and `simp`. The simplifier will loop if you apply it blindly, but `rewrite` will do the trick.

```
example (x y : ℕ) :
  div x y = if 0 < y ∧ y ≤ x then div (x - y) y + 1 else 0 :=
by rw [div]

example (x y : ℕ) (h : 0 < y ∧ y ≤ x) :
  div x y = div (x - y) y + 1 :=
by rw [div, if_pos h]
```

The following example is similar: it converts any natural number to a binary expression, represented as a list of 0's and 1's. We have to provide the equation compiler with evidence that the recursive call is decreasing, which we do here with a `sorry`. The `sorry` does not prevent the bytecode evaluator from evaluating the function successfully.

```
def nat_to_bin : ℕ → list ℕ
| 0       := [0]
| 1       := [1]
| (n + 2) :=
  have (n + 2) / 2 < n + 2, from sorry,
  nat_to_bin ((n + 2) / 2) ++ [n % 2]

#eval nat_to_bin 1234567
```

As a final example, we observe that Ackermann's function can be defined directly, because it is justified by the well foundedness of the lexicographic order on the natural numbers.

```
def ack : nat → nat → nat
| 0     y     := y+1
| (x+1) 0     := ack x 1
| (x+1) (y+1) := ack x (ack (x+1) y)

#eval ack 3 5
```

Lean's mechanisms for guessing a well-founded relation and then proving that recursive calls decrease are still in a rudimentary state. They will be improved over time. When they work, they provide a much more convenient way of defining functions than using `well_founded.fix` manually. When they don't, the latter is always available as a backup.

8.5 Mutual Recursion

Lean also supports mutual recursive definitions. The syntax is similar to that for mutual inductive types, as described in Section 7.9. Here is an example:

```
mutual def even, odd
with even : nat → bool
| 0     := tt
| (a+1) := odd a
with odd : nat → bool
```

(continues on next page)

(continued from previous page)

```
| 0     := ff
| (a+1) := even a

example (a : nat) : even (a + 1) = odd a :=
by simp [even]

example (a : nat) : odd (a + 1) = even a :=
by simp [odd]

lemma even_eq_not_odd : ∀ a, even a = bnot (odd a) :=
begin
  intro a, induction a,
  simp [even, odd],
  simp [*, even, odd]
end
```

What makes this a mutual definition is that `even` is defined recursively in terms of `odd`, while `odd` is defined recursively in terms of `even`. Under the hood, this is compiled as a single recursive definition. The internally defined function takes, as argument, an element of a sum type, either an input to `even`, or an input to `odd`. It then returns an output appropriate to the input. To define that function, Lean uses a suitable well-founded measure. The internals are meant to be hidden from users; the canonical way to make use of such definitions is to use `rewrite` or `simp`, as we did above.

Mutual recursive definitions also provide natural ways of working with mutual and nested inductive types, as described in Section 7.9. Recall the definition of `even` and `odd` as mutual inductive predicates, as presented as an example there:

```
mutual inductive even, odd
with even : ℕ → Prop
| even_zero : even 0
| even_succ : ∀ n, odd n → even (n + 1)
with odd : ℕ → Prop
| odd_succ : ∀ n, even n → odd (n + 1)
```

The constructors, `even_zero`, `even_succ`, and `odd_succ` provide positive means for showing that a number is even or odd. We need to use the fact that the inductive type is generated by these constructors to know that the zero is not odd, and that the latter two implications reverse. As usual, the constructors are kept in a namespace that is named after the type being defined, and the command `open even odd` allows us to access them move conveniently.

```
open even odd

theorem not_odd_zero : ¬ odd 0.

mutual theorem even_of_odd_succ, odd_of_even_succ
with even_of_odd_succ : ∀ n, odd (n + 1) → even n
| _ (odd_succ n h) := h
with odd_of_even_succ : ∀ n, even (n + 1) → odd n
| _ (even_succ n h) := h
```

For another example, suppose we use a nested inductive type to define a set of terms inductively, so that a term is either a constant (with a name given by a string), or the result of applying a constant to a list of constants.

```
inductive term
| const : string → term
| app   : string → list term → term
```

We can then use a mutual recursive definition to count the number of constants occurring in a term, as well as the number occurring in a list of terms.

```
open term

mutual def num_consts, num_consts_lst
with num_consts : term → nat
| (term.const n)   := 1
| (term.app n ts) := num_consts_lst ts
with num_consts_lst : list term → nat
| []       := 0
| (t::ts) := num_consts t + num_consts_lst ts

def sample_term := app "f" [app "g" [const "x"], const "y"]

#eval num_consts sample_term
```

8.6 Dependent Pattern Matching

All the examples of pattern matching we considered in Section 8.1 can easily be written using `cases_on` and `rec_on`. However, this is often not the case with indexed inductive families such as `vector α n`, since case splits impose constraints on the values of the indices. Without the equation compiler, we would need a lot of boilerplate code to define very simple functions such as `map`, `zip`, and `unzip` using recursors. To understand the difficulty, consider what it would take to define a function `tail` which takes a vector `v : vector α (succ n)` and deletes the first element. A first thought might be to use the `cases_on` function:

```
universe u

inductive vector (α : Type u) : nat → Type u
| nil {} : vector 0
| cons   : Π {n}, α → vector n → vector (n+1)

namespace vector
local notation h :: t := cons h t

#check @vector.cases_on
-- Π {α : Type*}
--   {C : Π (a : ℕ), vector α a → Type*}
--   {a : ℕ}
--   (n : vector α a),
--   (e1 : C 0 nil)
--   (e2 : Π {n : ℕ} (a : α) (a_1 : vector α n),
--           C (n + 1) (cons a a_1)),
--   C a n

end vector
```

But what value should we return in the `nil` case? Something funny is going on: if `v` has type `vector α (succ n)`, it *can't* be nil, but it is not clear how to tell that to `cases_on`.

One solution is to define an auxiliary function:

```
def tail_aux {α : Type*} {n m : ℕ} (v : vector α m) :
    m = n + 1 → vector α n :=
vector.cases_on v
  (assume H : 0 = n + 1, nat.no_confusion H)
  (assume m (a : α) w : vector α m,
```

(continues on next page)

(continued from previous page)

```
    assume H : m + 1 = n + 1,
      nat.no_confusion H (λ H1 : m = n, eq.rec_on H1 w))

def tail {α : Type*} {n : ℕ} (v : vector α (n+1)) :
  vector α n :=
tail_aux v rfl
```

In the `nil` case, `m` is instantiated to `0`, and `no_confusion` makes use of the fact that `0 = succ n` cannot occur. Otherwise, `v` is of the form `a :: w`, and we can simply return `w`, after casting it from a vector of length `m` to a vector of length `n`.

The difficulty in defining `tail` is to maintain the relationships between the indices. The hypothesis `e : m = n + 1` in `tail_aux` is used to communicate the relationship between `n` and the index associated with the minor premise. Moreover, the `zero = n + 1` case is unreachable, and the canonical way to discard such a case is to use `no_confusion`.

The `tail` function is, however, easy to define using recursive equations, and the equation compiler generates all the boilerplate code automatically for us. Here are a number of similar examples:

```
def head {α : Type*} : Π {n}, vector α (n+1) → α
| n (h :: t) := h

def tail {α : Type*} : Π {n}, vector α (n+1) → vector α n
| n (h :: t) := t

lemma eta {α : Type*} :
  ∀ {n} (v : vector α (n+1)), head v :: tail v = v
| n (h :: t) := rfl

def map {α β γ : Type*} (f : α → β → γ) :
  Π {n}, vector α n → vector β n → vector γ n
| 0     nil       nil       := nil
| (n+1) (a :: va) (b :: vb) := f a b :: map va vb

def zip {α β : Type*} :
  Π {n}, vector α n → vector β n → vector (α × β) n
| 0     nil       nil       := nil
| (n+1) (a :: va) (b :: vb) := (a, b) :: zip va vb
```

Note that we can omit recursive equations for "unreachable" cases such as `head nil`. The automatically generated definitions for indexed families are far from straightforward. For example:

```
#print map
#print map._main
```

The `map` function is even more tedious to define by hand than the `tail` function. We encourage you to try it, using `rec_on`, `cases_on` and `no_confusion`.

8.7 Inaccessible Terms

Sometimes an argument in a dependent matching pattern is not essential to the definition, but nonetheless has to be included to specialize the type of the expression appropriately. Lean allows users to mark such subterms as *inaccessible* for pattern matching. These annotations are essential, for example, when a term occurring in the left-hand side is neither a variable nor a constructor application, because these are not suitable targets for pattern matching. We can view such inaccessible terms as "don't care" components of the patterns. You can declare a subterm inaccessible by writing `.(t)`. If the inaccessible term can be inferred, you can also write `._`.

The following example can be found in [GoMM06]. We declare an inductive type that defines the property of "being in the image of `f`". You can view an element of the type `image_of f b` as evidence that `b` is in the image of `f`, whereby the constructor `imf` is used to build such evidence. We can then define any function `f` with an "inverse" which takes anything in the image of `f` to an element that is mapped to it. The typing rules forces us to write `f a` for the first argument, but this term is neither a variable nor a constructor application, and plays no role in the pattern-matching definition. To define the function `inverse` below, we *have to* mark `f a` inaccessible.

```
universe u
variables {α β : Type u}

inductive image_of (f : α → β) : β → Type u
| imf : Π a, image_of (f a)

open image_of

def inverse {f : α → β} : Π b, image_of f b → α
| .(f a) (imf a) := a
```

In the example above, the inaccessible annotation makes it clear that `f` is *not* a pattern matching variable.

Inaccessible terms can be used to clarify and control definitions that make use of dependent pattern matching. Consider the following definition of the function `vector.add,` which adds two vectors of elements of a type, assuming that type has an associated addition function:

```
universe u

inductive vector (α : Type u) : ℕ → Type u
| nil {} : vector 0
| cons   : Π {n}, α → vector n → vector (n+1)

namespace vector
local notation h :: t := cons h t

variable {α : Type u}

def add [has_add α] : Π {n : ℕ}, vector α n → vector α n → vector α n
| 0     nil        nil        := nil
| (n+1) (cons a v) (cons b w) := cons (a + b) (add v w)

end vector
```

The argument `{n : ℕ}` has to appear after the colon, because it cannot be held fixed throughout the definition. When implementing this definition, the equation compiler starts with a case distinction as to whether the first argument is `0` or of the form `n+1`. This is followed by nested case splits on the next two arguments, and in each case the equation compiler rules out the cases are not compatible with the first pattern.

But, in fact, a case split is not required on the first argument; the `cases_on` eliminator for `vector` automatically abstracts this argument and replaces it by `0` and `n + 1` when we do a case split on the second argument. Using inaccessible terms, we can prompt the equation compiler to avoid the case split on n:

```
def add [has_add α] : Π {n : ℕ}, vector α n → vector α n → vector α n
| ._ nil        nil        := nil
| ._ (cons a v) (cons b w) := cons (a + b) (add v w)
```

Marking the position as an inaccessible implicit argument tells the equation compiler first, that the form of the argument should be inferred from the constraints posed by the other arguments, and, second, that the first argument should *not* participate in pattern matching.

Using explicit inaccessible terms makes it even clearer what is going on.

```
def add [has_add α] : Π {n : ℕ}, vector α n → vector α n → vector α n
| .(0)   nil                  nil        := nil
| .(n+1) (@cons .(α) n a v) (cons b w) := cons (a + b) (add v w)
```

We have to introduce the variable n in the pattern @cons .(α) n a v, since it is involved in the pattern match over that argument. In contrast, the parameter α is held fixed; we could have left it implicit by writing ._ instead. The advantage to naming the variable there is that we can now use inaccessible terms in the first position to display the values that were inferred implicitly in the previous example.

8.8 Match Expressions

Lean also provides a compiler for *match-with* expressions found in many functional languages. It uses essentially the same infrastructure used to compile recursive equations.

```
def is_not_zero (m : ℕ) : bool :=
match m with
| 0     := ff
| (n+1) := tt
end
```

This does not look very different from an ordinary pattern matching definition, but the point is that a match can be used anywhere in an expression, and with arbitrary arguments.

```
variable {α : Type*}
variable p : α → bool

def filter : list α → list α
| []       := []
| (a :: l) :=
  match p a with
  |  tt := a :: filter l
  |  ff := filter l
  end

example : filter is_not_zero [1, 0, 0, 3, 0] = [1, 3] := rfl
```

Here is another example:

```
def foo (n : ℕ) (b c : bool) :=
5 + match n - 5, b && c with
    | 0,      tt := 0
    | m+1,    tt := m + 7
    | 0,      ff := 5
```

(continues on next page)

(continued from previous page)

```
    | m+1,     ff := m + 3
    end

#eval foo 7 tt ff

example : foo 7 tt ff = 9 := rfl
```

Notice that with multiple arguments, the syntax for the match statement is markedly different from that used for pattern matching in an ordinary recursive definition. Because arbitrary terms are allowed in the `match`, parentheses are not enough to set the arguments apart; if we wrote `(n - 5) (b && c)`, it would be interpreted as the result of applying `n - 5` to `b && c`. Instead, the arguments are separated by commas. Then, for consistency, the patterns on each line are separated by commas as well.

Lean uses the `match` construct internally to implement a pattern-matching `assume`, as well as a pattern-matching `let`. Thus, all four of these definitions have the same net effect.

```
def bar₁ : ℕ × ℕ → ℕ
| (m, n) := m + n

def bar₂ (p : ℕ × ℕ) : ℕ :=
match p with (m, n) := m + n end

def bar₃ : ℕ × ℕ → ℕ :=
λ ⟨m, n⟩, m + n

def bar₄ (p : ℕ × ℕ) : ℕ :=
let ⟨m, n⟩ := p in m + n
```

The second definition also illustrates the fact that in a match with a single pattern, the vertical bar is optional. These variations are equally useful for destructing propositions:

```
variables p q : ℕ → Prop

example : (∃ x, p x) → (∃ y, q y) →
  ∃ x y, p x ∧ q y
| ⟨x, px⟩ ⟨y, qy⟩ := ⟨x, y, px, qy⟩

example (h₀ : ∃ x, p x) (h₁ : ∃ y, q y) :
  ∃ x y, p x ∧ q y :=
match h₀, h₁ with
⟨x, px⟩, ⟨y, qy⟩ := ⟨x, y, px, qy⟩
end

example : (∃ x, p x) → (∃ y, q y) →
  ∃ x y, p x ∧ q y :=
λ ⟨x, px⟩ ⟨y, qy⟩, ⟨x, y, px, qy⟩

example (h₀ : ∃ x, p x) (h₁ : ∃ y, q y) :
  ∃ x y, p x ∧ q y :=
let ⟨x, px⟩ := h₀,
    ⟨y, qy⟩ := h₁ in
⟨x, y, px, qy⟩
```

8.9 Exercises

1. Use pattern matching to prove that the composition of surjective functions is surjective:

```
open function

#print surjective

universes u v w
variables {α : Type u} {β : Type v} {γ : Type w}
open function

lemma surjective_comp {g : β → γ} {f : α → β}
  (hg : surjective g) (hf : surjective f) :
surjective (g ∘ f) := sorry
```

2. Open a namespace `hidden` to avoid naming conflicts, and use the equation compiler to define addition, multiplication, and exponentiation on the natural numbers. Then use the equation compiler to derive some of their basic properties.

3. Similarly, use the equation compiler to define some basic operations on lists (like the `reverse` function) and prove theorems about lists by induction (such as the fact that `reverse (reverse l) = l` for any list `l`).

4. Define your own function to carry out course-of-value recursion on the natural numbers. Similarly, see if you can figure out how to define `well_founded.fix` on your own.

5. Following the examples in Section 8.6, define a function that will append two vectors. This is tricky; you will have to define an auxiliary function.

6. Consider the following type of arithmetic expressions. The idea is that `var n` is a variable, v_n, and `const n` is the constant whose value is `n`.

```
inductive aexpr : Type
| const : ℕ → aexpr
| var : ℕ → aexpr
| plus : aexpr → aexpr → aexpr
| times : aexpr → aexpr → aexpr

open aexpr

def sample_aexpr : aexpr :=
plus (times (var 0) (const 7)) (times (const 2) (var 1))
```

Here `sample_aexpr` represents `(v₀ + 7) * (2 + v₁)`.

Write a function that evaluates such an expression, evaluating each `var n` to `v n`.

```
def aeval (v : ℕ → ℕ) : aexpr → ℕ
| (const n)     := sorry
| (var n)       := v n
| (plus e₁ e₂)   := sorry
| (times e₁ e₂) := sorry

def sample_val : ℕ → ℕ
| 0 := 5
| 1 := 6
| _ := 0
```

(continues on next page)

(continued from previous page)

```
-- Try it out. You should get 47 here.
-- #eval aeval sample_val sample_aexpr
```

Implement "constant fusion," a procedure that simplifies subterms like `5 + 7` to `12`. Using the auxiliary function `simp_const`, define a function "fuse": to simplify a plus or a times, first simplify the arguments recursively, and then apply `simp_const` to try to simplify the result.

```
def simp_const : aexpr → aexpr
| (plus (const n₁) (const n₂))  := const (n₁ + n₂)
| (times (const n₁) (const n₂)) := const (n₁ * n₂)
| e                             := e

def fuse : aexpr → aexpr := sorry

theorem simp_const_eq (v : ℕ → ℕ) :
  ∀ e : aexpr, aeval v (simp_const e) = aeval v e :=
sorry

theorem fuse_eq (v : ℕ → ℕ) :
  ∀ e : aexpr, aeval v (fuse e) = aeval v e :=
sorry
```

The last two theorems show that the definitions preserve the value.

CHAPTER

NINE

STRUCTURES AND RECORDS

We have seen that Lean's foundational system includes inductive types. We have, moreover, noted that it is a remarkable fact that it is possible to construct a substantial edifice of mathematics based on nothing more than the type universes, Pi types, and inductive types; everything else follows from those. The Lean standard library contains many instances of inductive types (e.g., `nat`, `prod`, `list`), and even the logical connectives are defined using inductive types.

Remember that a non-recursive inductive type that contains only one constructor is called a *structure* or *record*. The product type is a structure, as is the dependent product type, that is, the Sigma type. In general, whenever we define a structure `S`, we usually define *projection* functions that allow us to "destruct" each instance of `S` and retrieve the values that are stored in its fields. The functions `prod.pr1` and `prod.pr2`, which return the first and second elements of a pair, are examples of such projections.

When writing programs or formalizing mathematics, it is not uncommon to define structures containing many fields. The `structure` command, available in Lean, provides infrastructure to support this process. When we define a structure using this command, Lean automatically generates all the projection functions. The `structure` command also allows us to define new structures based on previously defined ones. Moreover, Lean provides convenient notation for defining instances of a given structure.

9.1 Declaring Structures

The structure command is essentially a "front end" for defining inductive data types. Every `structure` declaration introduces a namespace with the same name. The general form is as follows:

```
structure <name> <parameters> <parent-structures> : Sort u :=
  <constructor> :: <fields>
```

Most parts are optional. Here is an example:

```
structure point (α : Type*) :=
mk :: (x : α) (y : α)
```

Values of type `point` are created using `point.mk a b`, and the fields of a point `p` are accessed using `point.x p` and `point.y p`. The structure command also generates useful recursors and theorems. Here are some of the constructions generated for the declaration above.

```
#check point              -- a Type
#check point.rec_on       -- the eliminator
#check point.x            -- a projection / field accessor
#check point.y            -- a projection / field accessor
```

You can obtain the complete list of generated constructions using the command `#print prefix`.

```
#print prefix point
```

Here are some simple theorems and expressions that use the generated constructions. As usual, you can avoid the prefix `point` by using the command `open point`.

```
#reduce point.x (point.mk 10 20)
#reduce point.y (point.mk 10 20)

open point

example (α : Type*) (a b : α) : x (mk a b) = a :=
rfl

example (α : Type*) (a b : α) : y (mk a b) = b :=
rfl
```

Given `p : point nat`, the notation `p.x` is shorthand for `point.x p`. This provides a convenient way of accessing the fields of a structure.

```
def p := point.mk 10 20

#check p.x  -- nat
#reduce p.x  -- 10
#reduce p.y  -- 20
```

If the constructor is not provided, then a constructor is named `mk` by default.

```
structure prod (α : Type*) (β : Type*) :=
(pr1 : α) (pr2 : β)

#check prod.mk
```

The dot notation is convenient not just for accessing the projections of a record, but also for applying functions defined in a namespace with the same name. Recall from Section 3.3.1 that if `p` has type `point`, the expression `p.foo` is interpreted as `point.foo p`, assuming that the first non-implicit argument to `foo` has type `point`. The expression `p.add q` is therefore shorthand for `point.add p q` in the example below.

```
structure point (α : Type*) :=
mk :: (x : α) (y : α)

namespace point

def add (p q : point ℕ) := mk (p.x + q.x) (p.y + q.y)

end point

def p : point ℕ := point.mk 1 2
def q : point ℕ := point.mk 3 4

#reduce p.add q  -- {x := 4, y := 6}
```

In the next chapter, you will learn how to define a function like `add` so that it works generically for elements of `point` α rather than just `point ℕ`, assuming α has an associated addition operation.

More generally, given an expression `p.foo x y z`, Lean will insert `p` at the first non-implicit argument to `foo` of type `point`. For example, with the definition of scalar multiplication below, `p.smul 3` is interpreted as `point.smul 3 p`.

```
structure point (α : Type*) :=
mk :: (x : α) (y : α)

def point.smul (n : ℕ) (p : point ℕ) :=
point.mk (n * p.x) (n * p.y)

def p : point ℕ := point.mk 1 2

#reduce p.smul 3  -- {x := 3, y := 6}
```

It is common to use a similar trick with the `list.map` function, which takes a list as its second non-implicit argument:

```
#check @list.map
-- Π {α : Type u_1} {β : Type u_2}, (α → β) → list α → list β

def l : list nat := [1, 2, 3]
def f : nat → nat := λ x, x * x

#eval l.map f  -- [1, 4, 9]
```

Here `l.map f` is interpreted as `list.map f l`.

If you have a structure definition that depends on a type, you can make it polymorphic over universe levels using a previously declared universe variable, declaring a universe variable on the fly, or using an underscore:

```
universe u

structure point (α : Type u) :=
mk :: (x : α) (y : α)

structure {v} point2 (α : Type v) :=
mk :: (x : α) (y : α)

structure point3 (α : Type _) :=
mk :: (x : α) (y : α)

#check @point
#check @point2
#check @point3
```

The three variations have the same net effect. The annotations in the next example force the parameters α and β to be types from the same universe, and set the return type to also be in the same universe.

```
structure {u} prod (α : Type u) (β : Type u) :
  Type u :=
(pr1 : α) (pr2 : β)

set_option pp.universes true
#check prod.mk
```

The `set_option` command above instructs Lean to display the universe levels. We can use the anonymous constructor notation to build structure values whenever the expected type is known.

```
structure {u} prod (α : Type u) (β : Type u) :
  Type u :=
(pr1 : α) (pr2 : β)
```

(continues on next page)

(continued from previous page)

```
example : prod nat nat :=
⟨1, 2⟩

#check (⟨1, 2⟩ : prod nat nat)
```

9.2 Objects

We have been using constructors to create elements of a structure type. For structures containing many fields, this is often inconvenient, because we have to remember the order in which the fields were defined. Lean therefore provides the following alternative notations for defining elements of a structure type.

```
{ structure-name . (<field-name> := <expr>)* }
or
{ (<field-name> := <expr>)* }
```

The prefix `structure-name .` can be omitted whenever the name of the structure can be inferred from the expected type. For example, we use this notation to define "points." The order that the fields are specified does not matter, so all the expressions below define the same point.

```
structure point (α : Type*) :=
mk :: (x : α) (y : α)

#check { point . x := 10, y := 20 }  -- point ℕ
#check { point . y := 20, x := 10 }
#check ({x := 10, y := 20} : point nat)

example : point nat :=
{ y := 20, x := 10 }
```

If the value of a field is not specified, Lean tries to infer it. If the unspecified fields cannot be inferred, Lean signs an error indicating the corresponding placeholder could not be synthesized.

```
structure my_struct :=
mk :: {α : Type*} {β : Type*} (a : α) (b : β)

#check { my_struct . a := 10, b := true }
```

Record update is another common operation which amounts to creating a new record object by modifying the value of one or more fields in an old one. Lean allows you to specify that unassigned fields in the specification of a record should be taken from a previous defined record object `r` by adding the annotation `..r` after the field assignments. If more than one record object is provided, then they are visited in order until Lean finds one the contains the unspecified field. Lean raises an error if any of the field names remain unspecified after all the objects are visited.

```
structure point (α : Type*) :=
mk :: (x : α) (y : α)

def p : point nat :=
{x := 1, y := 2}

#reduce {y := 3, ..p}  -- {x := 1, y := 3}
#reduce {x := 4, ..p}  -- {x := 4, y := 2}

structure point3 (α : Type*) :=
```

(continues on next page)

(continued from previous page)

```
mk :: (x : α) (y : α) (z : α)

def q : point3 nat :=
{x := 5, y := 5, z := 5}

def r : point3 nat := {x := 6, ..p, ..q}

#print r  -- {x := 6, y := p.y, z := q.z}
#reduce r  -- {x := 6, y := 2, z := 5}
```

9.3 Inheritance

We can *extend* existing structures by adding new fields. This feature allow us to simulate a form of *inheritance*.

```
structure point (α : Type*) :=
mk :: (x : α) (y : α)

inductive color
| red | green | blue

structure color_point (α : Type*) extends point α :=
mk :: (c : color)
```

In the next example, we define a structure using multiple inheritance, and then define an object using objects of the parent structures.

```
structure point (α : Type*) :=
(x : α) (y : α) (z : α)

structure rgb_val :=
(red : nat) (green : nat) (blue : nat)

structure red_green_point (α : Type*) extends point α, rgb_val :=
(no_blue : blue = 0)

def p   : point nat := {x := 10, y := 10, z := 20}
def rgp : red_green_point nat :=
{red := 200, green := 40, blue := 0, no_blue := rfl, ..p}

example : rgp.x   = 10 := rfl
example : rgp.red = 200 := rfl
```

CHAPTER

TEN

TYPE CLASSES

We have seen that Lean's elaborator provides helpful automation, filling in information that is tedious to enter by hand. In this section we will explore a simple but powerful technical device known as *type class inference*, which provides yet another mechanism for the elaborator to supply missing information.

The notion of a *type class* originated with the *Haskell* programming language. In that context, it is often used to associate operations, like a canonical addition or multiplication operation, to a data type. Many of the original uses carry over, but, as we will see, the realm of interactive theorem proving raises even more possibilities for their use.

10.1 Type Classes and Instances

Any family of types can be marked as a *type class*. We can then declare particular elements of a type class to be *instances*. These provide hints to the elaborator: any time the elaborator is looking for an element of a type class, it can consult a table of declared instances to find a suitable element.

More precisely, there are three steps involved:

- First, we declare a family of inductive types to be a type class.
- Second, we declare instances of the type class.
- Finally, we mark some implicit arguments with square brackets instead of curly brackets, to inform the elaborator that these arguments should be inferred by the type class mechanism.

Let us start with a simple example. Many theorems hold under the additional assumption that a type is inhabited, which is to say, it has at least one element. For example, if α is a type, `∃ x : α, x = x` is true only if α is inhabited. Similarly, it often happens that we would like a definition to return a default element in a "corner case." For example, we would like the expression `head l` to be of type α when `l` is of type `list α`; but then we are faced with the problem that `head l` needs to return an "arbitrary" element of α in the case where `l` is the empty list, `nil`.

The standard library defines a type class `inhabited : Type → Type` to enable type class inference to infer a "default" or "arbitrary" element of an inhabited type. In the example below, we use a namespace `hidden` as usual to avoid conflicting with the definitions in the standard library.

Let us start with the first step of the program above, declaring an appropriate class:

```
class inhabited (α : Type _) :=
(default : α)
```

The command `class` above is shorthand for

```
@[class] structure inhabited (α : Type _) :=
(default : α)
```

An element of the class `inhabited α` is simply an expression of the form `inhabited.mk a`, for some element `a : α`. The projection `inhabited.default` will allow us to "extract" such an element of α from an element of `inhabited α`.

The second step of the program is to populate the class with some instances:

```
instance Prop_inhabited : inhabited Prop :=
inhabited.mk true

instance bool_inhabited : inhabited bool :=
inhabited.mk tt

instance nat_inhabited : inhabited nat :=
inhabited.mk 0

instance unit_inhabited : inhabited unit :=
inhabited.mk ()
```

In the Lean standard library, we regularly use the anonymous constructor when defining instances. It is particularly useful when the class name is long.

```
instance Prop_inhabited : inhabited Prop :=
⟨true⟩

instance bool_inhabited : inhabited bool :=
⟨tt⟩

instance nat_inhabited : inhabited nat :=
⟨0⟩

instance unit_inhabited : inhabited unit :=
⟨()⟩
```

These declarations simply record the definitions `Prop_inhabited`, `bool_inhabited`, `nat_inhabited`, and `unit_inhabited` on a list of instances. Whenever the elaborator is looking for a value to assign to an argument `?M` of type `inhabited α` for some α, it can check the list for a suitable instance. For example, if it looking for an instance of `inhabited Prop`, it will find `Prop_inhabited`.

The final step of the program is to define a function that infers an element `s : inhabited α` and puts it to good use. The following function simply extracts the corresponding element `a : α`:

```
def default (α : Type*) [s : inhabited α] : α :=
@inhabited.default α s
```

This has the effect that given a type expression α, whenever we write `default α`, we are really writing `default α ?s`, leaving the elaborator to find a suitable value for the metavariable `?s`. When the elaborator succeeds in finding such a value, it has effectively produced an element of type α, as though by magic.

```
#check default Prop  -- Prop
#check default nat   -- ℕ
#check default bool  -- bool
#check default unit  -- unit
```

In general, whenever we write `default α`, we are asking the elaborator to synthesize an element of type α.

Notice that we can "see" the value that is synthesized with `#reduce`:

```
#reduce default Prop  -- true
#reduce default nat   -- 0
#reduce default bool  -- ff
#reduce default unit  -- ()
```

Sometimes we want to think of the default element of a type as being an *arbitrary* element, whose specific value should not play a role in our proofs. For that purpose, we can write `arbitrary α` instead of `default α`. The definition of `arbitrary` is the same as that of default, but is marked `irreducible` to discourage the elaborator from unfolding it. This does not preclude proofs from making use of the value, however, so the use of `arbitrary` rather than `default` functions primarily to signal intent.

10.2 Chaining Instances

If that were the extent of type class inference, it would not be all that impressive; it would be simply a mechanism of storing a list of instances for the elaborator to find in a lookup table. What makes type class inference powerful is that one can *chain* instances. That is, an instance declaration can in turn depend on an implicit instance of a type class. This causes class inference to chain through instances recursively, backtracking when necessary, in a Prolog-like search.

For example, the following definition shows that if two types α and β are inhabited, then so is their product:

```
instance prod_inhabited
    {α β : Type*} [inhabited α] [inhabited β] :
  inhabited (prod α β) :=
⟨(default α, default β)⟩
```

With this added to the earlier instance declarations, type class instance can infer, for example, a default element of `nat × bool`:

```
#check default (nat × bool)
#reduce default (nat × bool)
```

Given the expression `default (nat × bool)`, the elaborator is called on to infer an implicit argument `?M : inhabited (nat × bool)`. The instance `prod_inhabited` reduces this to inferring `?M1 : inhabited nat` and `?M2 : inhabited bool`. The first one is solved by the instance `nat_inhabited`. The second uses `bool_inhabited`.

Similarly, we can inhabit function spaces with suitable constant functions:

```
instance inhabited_fun (α : Type*) {β : Type*} [inhabited β] :
  inhabited (α → β) :=
⟨(λ a : α, default β)⟩

#check default (nat → nat × bool)
#reduce default (nat → nat × bool)
```

In this case, type class inference finds the default element `λ (a : nat), (0, tt)`.

As an exercise, try defining default instances for other types, such as sum types and the list type.

10.3 Inferring Notation

We now consider the application of type classes that motivates their use in functional programming languages like Haskell, namely, to overload notation in a principled way. In Lean, a symbol like + can be given entirely unrelated meanings, a phenomenon that is sometimes called "ad-hoc" overloading. Typically, however, we use the + symbol to denote a binary function from a type to itself, that is, a function of type $\alpha \to \alpha \to \alpha$ for some type α. We can use type classes to infer an appropriate addition function for suitable types α. We will see in the next section that this is especially useful for developing algebraic hierarchies of structures in a formal setting.

The standard library declares a type class `has_add α` as follows:

```
class has_add (α : Type*) :=
(add : α → α → α)

def add {α : Type*} [has_add α] : α → α → α := has_add.add

notation a ` + ` b := add a b
```

The class `has_add α` is supposed to be inhabited exactly when there is an appropriate addition function for α. The `add` function is designed to find an instance of `has_add α` for the given type, α, and apply the corresponding binary addition function. The notation `a + b` thus refers to the addition that is appropriate to the type of `a` and `b`. We can then declare instances for `nat`, and `bool`:

```
instance nat_has_add : has_add nat :=
⟨nat.add⟩

instance bool_has_add : has_add bool :=
⟨bor⟩

#check 2 + 2     -- nat
#check tt + ff   -- bool
```

As with `inhabited`, the power of type class inference stems not only from the fact that the class enables the elaborator to look up appropriate instances, but also from the fact that it can chain instances to infer complex addition operations. For example, assuming that there are appropriate addition functions for types α and β, we can define addition on $\alpha \times \beta$ pointwise:

```
instance prod_has_add {α : Type u} {β : Type v}
    [has_add α] [has_add β] :
  has_add (α × β) :=
⟨λ ⟨a₁, b₁⟩ ⟨a₂, b₂⟩, ⟨a₁+a₂, b₁+b₂⟩⟩

#check (1, 2) + (3, 4)     -- ℕ × ℕ
#reduce  (1, 2) + (3, 4)   -- (4, 6)
```

We can similarly define pointwise addition of functions:

```
instance fun_has_add {α : Type u} {β : Type v} [has_add β] :
  has_add (α → β) :=
⟨λ f g x, f x + g x⟩

#check (λ x : nat, 1) + (λ x, 2)    -- ℕ → ℕ
#reduce (λ x : nat, 1) + (λ x, 2)    -- λ (x : ℕ), 3
```

As an exercise, try defining instances of `has_add` for lists, and show that they work as expected.

10.4 Decidable Propositions

Let us consider another example of a type class defined in the standard library, namely the type class of `decidable` propositions. Roughly speaking, an element of `Prop` is said to be decidable if we can decide whether it is true or false. The distinction is only useful in constructive mathematics; classically, every proposition is decidable. But if we use the classical principle, say, to define a function by cases, that function will not be computable. Algorithmically speaking, the `decidable` type class can be used to infer a procedure that effectively determines whether or not the proposition is true. As a result, the type class supports such computational definitions when they are possible while at the same time allowing a smooth transition to the use of classical definitions and classical reasoning.

In the standard library, `decidable` is defined formally as follows:

```
class inductive decidable (p : Prop) : Type
| is_false : ¬p → decidable
| is_true  :  p → decidable
```

Logically speaking, having an element `t : decidable p` is stronger than having an element `t : p ∨ ¬p`; it enables us to define values of an arbitrary type depending on the truth value of `p`. For example, for the expression `if p then a else b` to make sense, we need to know that `p` is decidable. That expression is syntactic sugar for `ite p a b`, where `ite` is defined as follows:

```
def ite (c : Prop) [d : decidable c] {α : Type*}
  (t e : α) : α :=
decidable.rec_on d (λ hnc, e) (λ hc, t)
```

The standard library also contains a variant of `ite` called `dite`, the dependent if-then-else expression. It is defined as follows:

```
def dite (c : Prop) [d : decidable c] {α : Type*}
  (t : c → α) (e : ¬ c → α) : α :=
decidable.rec_on d (λ hnc : ¬ c, e hnc) (λ hc : c, t hc)
```

That is, in `dite c t e`, we can assume `hc : c` in the "then" branch, and `hnc : ¬ c` in the "else" branch. To make `dite` more convenient to use, Lean allows us to write `if h : c then t else e` instead of `dite c (λ h : c, t) (λ h : ¬ c, e)`.

Without classical logic, we cannot prove that every proposition is decidable. But we can prove that *certain* propositions are decidable. For example, we can prove the decidability of basic operations like equality and comparisons on the natural numbers and the integers. Moreover, decidability is preserved under propositional connectives:

```
#check @and.decidable
-- Π {p q : Prop} [hp : decidable p] [hq : decidable q],
--   decidable (p ∧ q)

#check @or.decidable
#check @not.decidable
#check @implies.decidable
```

Thus we can carry out definitions by cases on decidable predicates on the natural numbers:

```
open nat

def step (a b x : ℕ) : ℕ :=
if x < a ∨ x > b then 0 else 1

set_option pp.implicit true
#print definition step
```

Turning on implicit arguments shows that the elaborator has inferred the decidability of the proposition x < a ∨ x > b, simply by applying appropriate instances.

With the classical axioms, we can prove that every proposition is decidable. You can import the classical axioms and make the generic instance of decidability available by including this at the top of your file:

```
open classical
local attribute [instance] prop_decidable
```

Thereafter decidable p has an instance for every p, and the elaborator infers that value quickly. Thus all theorems in the library that rely on decidability assumptions are freely available when you want to reason classically. In Chapter 11, we will see that using the law of the excluded middle to define functions can prevent them from being used computationally. If that is important to you, it is best to use sections to limit the use of prop_decidable to places where it is really needed. Alternatively, you can can assign prop_decidable a low priority:

```
open classical
local attribute [instance, priority 10] prop_decidable
```

The guarantees that Lean will favor other instances and fall back on prop_decidable only after other attempts to infer decidability have failed.

The decidable type class also provides a bit of small-scale automation for proving theorems. The standard library introduces the following definitions and notation:

```
def as_true (c : Prop) [decidable c] : Prop :=
if c then true else false

def of_as_true {c : Prop} [h₁ : decidable c] (h₂ : as_true c) :
  c :=
match h₁, h₂ with
| (is_true h_c),  h₂ := h_c
| (is_false h_c), h₂ := false.elim h₂
end

notation `dec_trivial` := of_as_true (by tactic.triv)
```

They work as follows. The expression as_true c tries to infer a decision procedure for c, and, if it is successful, evaluates to either true or false. In particular, if c is a true closed expression, as_true c will reduce definitionally to true. On the assumption that as_true c holds, of_as_true produces a proof of c. The notation dec_trivial puts it all together: to prove a target c, it applies of_as_true and then uses the triv tactic to prove as_true c. By the previous observations, dec_trivial will succeed any time the inferred decision procedure for c has enough information to evaluate, definitionally, to the is_true case. Here is an example of how dec_trivial can be used:

```
example : 1 ≠ 0 ∧ (5 < 2 ∨ 3 < 7) := dec_trivial
```

Try changing the 3 to 10, thereby rendering the expression false. The resulting error message complains that of_as_true (1 ≠ 0 ∧ (5 < 2 ∨ 10 < 7)) is not definitionally equal to true.

10.5 Managing Type Class Inference

You can ask Lean for information about the classes and instances that are currently in scope:

```
#print classes
#print instances inhabited
```

If you are ever in a situation where you need to supply an expression that Lean can infer by type class inference, you can ask Lean to carry out the inference using the tactic `apply_instance` or the expression `infer_instance`:

```
def foo : has_add nat := by apply_instance
def bar : inhabited (nat → nat) := by apply_instance

def baz : has_add nat := infer_instance
def bla : inhabited (nat → nat) := infer_instance

#print foo    -- nat.has_add
#reduce foo   -- (unreadable)

#print bar    -- pi.inhabited ℕ
#reduce bar   -- {default := λ (a : ℕ), 0}

#print baz    -- infer_instance
#reduce baz   -- (same as for #reduce foo)

#print bla    -- infer_instance
#reduce bla   -- {default := λ (a : ℕ), 0}
```

In fact, you can use Lean's `(t : T)` notation to specify the class whose instance you are looking for, in a concise manner:

```
#reduce (by apply_instance : inhabited ℕ)
#reduce (infer_instance : inhabited ℕ)
```

Sometimes Lean can't find an instance because the class is buried under a definition. For example, with the core library, Lean cannot find an instance of `inhabited (set α)`. We can declare one explicitly:

```
-- fails
-- example {α : Type*} : inhabited (set α) :=
-- by apply_instance

def inhabited.set (α : Type*) : inhabited (set α) := ⟨∅⟩

#print inhabited.set     -- λ {α : Type u}, {default := ∅}
#reduce inhabited.set ℕ  -- {default := λ (a : ℕ), false}
```

Alternatively, we can help Lean out by unfolding the definition. The type `set α` is defined to be `α → Prop`. Lean knows that `Prop` is inhabited, and this is enough for it to be able to infer an element of the function type.

```
def inhabited.set (α : Type*) : inhabited (set α) :=
by unfold set; apply_instance

#print inhabited.set
  -- λ (α : Type u), eq.mpr _ (pi.inhabited α)
#reduce inhabited.set ℕ
  -- {default := λ (a : ℕ), true}
```

Using the `dunfold` tactic instead of `unfold` yields a slightly different expression (try it!), since `dunfold` uses definitional reduction to unfold the definition, rather than an explicit rewrite.

At times, you may find that the type class inference fails to find an expected instance, or, worse, falls into an infinite loop and times out. To help debug in these situations, Lean enables you to request a trace of the search:

```
set_option trace.class_instances true
```

If you are using VS Code, you can read the results by hovering over the relevant theorem or definition, or opening the messages window with `Ctrl-Shift-Enter`. In Emacs, you can use `C-c C-x` to run an independent Lean process on your file, and the output buffer will show a trace every time the type class resolution procedure is subsequently triggered.

You can also limit the search depth (the default is 32):

```
set_option class.instance_max_depth 5
```

Remember also that in both the VS Code and Emacs editor modes, tab completion works in `set_option`, to help you find suitable options.

As noted above, the type class instances in a given context represent a Prolog-like program, which gives rise to a backtracking search. Both the efficiency of the program and the solutions that are found can depend on the order in which the system tries the instance. Instances which are declared last are tried first. Moreover, if instances are declared in other modules, the order in which they are tried depends on the order in which namespaces are opened. Instances declared in namespaces which are opened later are tried earlier.

You can change the order that type classes instances are tried by assigning them a *priority*. When an instance is declared, it is assigned a priority value `std.priority.default`, defined to be 1000 in module `init.core` in the standard library. You can assign other priorities when defining an instance, and you can later change the priority with the `attribute` command. The following example illustrates how this is done:

```
class foo :=
(a : nat) (b : nat)

@[priority std.priority.default+1]
instance i1 : foo :=
⟨1, 1⟩

instance i2 : foo :=
⟨2, 2⟩

example : foo.a = 1 := rfl

@[priority std.priority.default+20]
instance i3 : foo :=
⟨3, 3⟩

example : foo.a = 3 := rfl

attribute [instance, priority 10] i3

example : foo.a = 1 := rfl

attribute [instance, priority std.priority.default-10] i1

example : foo.a = 2 := rfl
```

10.6 Coercions using Type Classes

The most basic type of coercion maps elements of one type to another. For example, a coercion from `nat` to `int` allows us to view any element `n : nat` as an element of `int`. But some coercions depend on parameters; for example, for any type α, we can view any element `l : list α` as an element of `set α`, namely, the set of elements occurring in the list. The corresponding coercion is defined on the "family" of types `list α`, parameterized by α.

Lean allows us to declare three kinds of coercions:

- from a family of types to another family of types
- from a family of types to the class of sorts
- from a family of types to the class of function types

The first kind of coercion allows us to view any element of a member of the source family as an element of a corresponding member of the target family. The second kind of coercion allows us to view any element of a member of the source family as a type. The third kind of coercion allows us to view any element of the source family as a function. Let us consider each of these in turn.

In Lean, coercions are implemented on top of the type class resolution framework. We define a coercion from α to β by declaring an instance of `has_coe α β`. For example, we can define a coercion from `bool` to `Prop` as follows:

```
instance bool_to_Prop : has_coe bool Prop :=
⟨λ b, b = tt⟩
```

This enables us to use boolean terms in if-then-else expressions:

```
#reduce if tt then 3 else 5
#reduce if ff then 3 else 5
```

We can define a coercion from `list α` to `set α` as follows:

```
def list.to_set {α : Type*} : list α → set α
| []      := ∅
| (h::t)  := {h} ∪ list.to_set t

instance list_to_set_coe (α : Type*) :
  has_coe (list α) (set α) :=
⟨list.to_set⟩

def s : set nat  := {1, 2}
def l : list nat := [3, 4]

#check s ∪ l -- set nat
```

Coercions are only considered if the given and expected types do not contain metavariables at elaboration time. In the following example, when we elaborate the union operator, the type of `[3, 2]` is `list ?m`, and a coercion will not be considered since it contains metavariables.

```
/- The following #check command produces an error. -/
-- #check s ∪ [3, 2]
```

We can work around this issue by using a type ascription.

```
#check s ∪ [(3:nat), 2]
-- or
#check s ∪ ([3, 2] : list nat)
```

In the examples above, you may have noticed the symbol ↑ produced by the `#check` commands. It is the lift operator, `↑t` is notation for `coe t`. We can use this operator to force a coercion to be introduced in a particular place. It is also helpful to make our intent clear, and work around limitations of the coercion resolution system.

```
#check s ∪ ↑[3, 2]

variables n m : nat
variable i : int
#check i + ↑n + ↑m
#check i + ↑(n + m)
#check ↑n + i
```

In the first two examples, the coercions are not strictly necessary since Lean will insert implicit nat → int coercions. However, `#check n + i` would raise an error, because the expected type of `i` is nat in order to match the type of n, and no int → nat coercion exists). In the third example, we therefore insert an explicit ↑ to coerce `n` to `int`.

The standard library defines a coercion from subtype `{x : α // p x}` to α as follows:

```
instance coe_subtype {α : Type*} {p : α → Prop} :
  has_coe {x // p x} α :=
⟨λ s, subtype.val s⟩
```

Lean will also chain coercions as necessary. Actually, the type class `has_coe_t` is the transitive closure of `has_coe`. You may have noticed that the type of `coe` depends on `has_lift_t`, the transitive closure of the type class `has_lift`, instead of `has_coe_t`. Every instance of `has_coe_t` is also an instance of `has_lift_t`, but the elaborator only introduces automatically instances of `has_coe_t`. That is, to be able to coerce using an instance of `has_lift_t`, we must use the operator ↑. In the standard library, we have the following instance:

```
namespace hidden
universes u v

instance lift_list {a : Type u} {b : Type v} [has_lift_t a b] :
  has_lift (list a) (list b) :=
⟨λ l, list.map (@coe a b _) l⟩

variables s : list nat
variables r : list int
#check ↑s ++ r

end hidden
```

It is not an instance of `has_coe` because lists are frequently used for writing programs, and we do not want a linear-time operation to be silently introduced by Lean, and potentially mask mistakes performed by the user. By forcing the user to write ↑, she is making her intent clear to Lean.

Let us now consider the second kind of coercion. By the *class of sorts*, we mean the collection of universes `Type u`. A coercion of the second kind is of the form

```
c : Π x1 : A1, ..., xn : An, F x1 ... xn → Type u
```

where `F` is a family of types as above. This allows us to write `s : t` whenever `t` is of type `F a1 ... an`. In other words, the coercion allows us to view the elements of `F a1 ... an` as types. This is very useful when defining algebraic structures in which one component, the carrier of the structure, is a `Type`. For example, we can define a semigroup as follows:

```
universe u
```

(continues on next page)

(continued from previous page)

```
structure Semigroup : Type (u+1) :=
(carrier : Type u)
(mul : carrier → carrier → carrier)
(mul_assoc : ∀ a b c : carrier,
               mul (mul a b) c = mul a (mul b c))

instance Semigroup_has_mul (S : Semigroup) :
  has_mul (S.carrier) :=
⟨S.mul⟩
```

In other words, a semigroup consists of a type, `carrier`, and a multiplication, `mul`, with the property that the multiplication is associative. The `instance` command allows us to write `a * b` instead of `Semigroup.mul S a b` whenever we have `a b : S.carrier`; notice that Lean can infer the argument `S` from the types of `a` and `b`. The function `Semigroup.carrier` maps the class `Semigroup` to the sort `Type u`:

```
#check Semigroup.carrier
```

If we declare this function to be a coercion, then whenever we have a semigroup `S : Semigroup`, we can write `a : S` instead of `a : S.carrier`:

```
instance Semigroup_to_sort : has_coe_to_sort Semigroup :=
{S := Type u, coe := λ S, S.carrier}

example (S : Semigroup) (a b c : S) :
  (a * b) * c = a * (b * c) :=
Semigroup.mul_assoc _ a b c
```

It is the coercion that makes it possible to write `(a b c : S)`. Note that, we define an instance of `has_coe_to_sort Semigroup` instead of `has_coe Semigroup Type`. The reason is that when Lean needs a coercion to sort, it only knows it needs a type, but, in general, the universe is not known. The field `S` in the class `has_coe_to_sort` is used to specify the universe we are coercing too.

By the *class of function types*, we mean the collection of Pi types `Π z : B, C`. The third kind of coercion has the form

```
c : Π x1 : A1, ..., xn : An, y : F x1 ... xn, Π z : B, C
```

where `F` is again a family of types and `B` and `C` can depend on `x1, ..., xn, y`. This makes it possible to write `t s` whenever `t` is an element of `F a1 ... an`. In other words, the coercion enables us to view elements of `F a1 ... an` as functions. Continuing the example above, we can define the notion of a morphism between semigroups `S1` and `S2`. That is, a function from the carrier of `S1` to the carrier of `S2` (note the implicit coercion) that respects the multiplication. The projection `morphism.mor` takes a morphism to the underlying function:

```
instance Semigroup_to_sort : has_coe_to_sort Semigroup :=
{S := _, coe := λ S, S.carrier}

structure morphism (S1 S2 : Semigroup) :=
(mor : S1 → S2)
(resp_mul : ∀ a b : S1, mor (a * b) = (mor a) * (mor b))

#check @morphism.mor
```

As a result, it is a prime candidate for the third type of coercion.

```
instance morphism_to_fun (S1 S2 : Semigroup) :
  has_coe_to_fun (morphism S1 S2) :=
```

(continues on next page)

(continued from previous page)

```
{ F   := λ _, S1 → S2,
  coe := λ m, m.mor }

lemma resp_mul {S1 S2 : Semigroup}
    (f : morphism S1 S2) (a b : S1) :
  f (a * b) = f a * f b :=
f.resp_mul a b

example (S1 S2 : Semigroup) (f : morphism S1 S2) (a : S1) :
  f (a * a * a) = f a * f a * f a :=
calc
  f (a * a * a) = f (a * a) * f a : by rw [resp_mul f]
            ... = f a * f a * f a : by rw [resp_mul f]
```

With the coercion in place, we can write `f (a * a * a)` instead of `morphism.mor f (a * a * a)`. When the `morphism`, `f`, is used where a function is expected, Lean inserts the coercion. Similar to `has_coe_to_sort`, we have yet another class `has_coe_to_fun` for this class of coercions. The field `F` is used to specify the function type we are coercing to. This type may depend on the type we are coercing from.

Finally, `⇑f` and `↥S` are notations for `coe_fn f` and `coe_sort S`. They are the coercion operators for the function and sort classes.

We can instruct Lean's pretty-printer to hide the operators ↑ and ⇑ with `set_option`.

```
theorem test (S1 S2 : Semigroup)
    (f : morphism S1 S2) (a : S1) :
  f (a * a * a) = f a * f a * f a :=
calc
  f (a * a * a) = f (a * a) * f a : by rw [resp_mul f]
            ... = f a * f a * f a : by rw [resp_mul f]

#check @test
set_option pp.coercions false
#check @test
```

CHAPTER

ELEVEN

AXIOMS AND COMPUTATION

We have seen that the version of the Calculus of Constructions that has been implemented in Lean includes dependent function types, inductive types, and a hierarchy of universes that starts with an impredicative, proof-irrelevant `Prop` at the bottom. In this chapter, we consider ways of extending the CIC with additional axioms and rules. Extending a foundational system in such a way is often convenient; it can make it possible to prove more theorems, as well as make it easier to prove theorems that could have been proved otherwise. But there can be negative consequences of adding additional axioms, consequences which may go beyond concerns about their correctness. In particular, the use of axioms bears on the computational content of definitions and theorems, in ways we will explore here.

Lean is designed to support both computational and classical reasoning. Users that are so inclined can stick to a "computationally pure" fragment, which guarantees that closed expressions in the system evaluate to canonical normal forms. In particular, any closed computationally pure expression of type ℕ, for example, will reduce to a numeral.

Lean's standard library defines an additional axiom, propositional extensionality, and a quotient construction which in turn implies the principle of function extensionality. These extensions are used, for example, to develop theories of sets and finite sets. We will see below that using these theorems can block evaluation in Lean's kernel, so that closed terms of type ℕ no longer evaluate to numerals. But Lean erases types and propositional information when compiling definitions to bytecode for its virtual machine evaluator, and since these axioms only add new propositions, they are compatible with that computational interpretation. Even computationally inclined users may wish to use the classical law of the excluded middle to reason about computation. This also blocks evaluation in the kernel, but it is compatible with compilation to bytecode.

The standard library also defines a choice principle that is entirely antithetical to a computational interpretation, since it magically produces "data" from a proposition asserting its existence. Its use is essential to some classical constructions, and users can import it when needed. But expressions that use this construction to produce data do not have computational content, and in Lean we are required to mark such definitions as `noncomputable` to flag that fact.

Using a clever trick (known as Diaconescu's theorem), one can use propositional extensionality, function extensionality, and choice to derive the law of the excluded middle. As noted above, however, use of the law of the excluded middle is still compatible with bytecode compilation and code extraction, as are other classical principles, as long as they are not used to manufacture data.

To summarize, then, on top of the underlying framework of universes, dependent function types, and inductive types, the standard library adds three additional components:

- the axiom of propositional extensionality
- a quotient construction, which implies function extensionality
- a choice principle, which produces data from an existential proposition.

The first two of these block normalization within Lean, but are compatible with bytecode evaluation, whereas the third is not amenable to computational interpretation. We will spell out the details more precisely below.

11.1 Historical and Philosophical Context

For most of its history, mathematics was essentially computational: geometry dealt with constructions of geometric objects, algebra was concerned with algorithmic solutions to systems of equations, and analysis provided means to compute the future behavior of systems evolving over time. From the proof of a theorem to the effect that "for every `x`, there is a `y` such that …", it was generally straightforward to extract an algorithm to compute such a `y` given `x`.

In the nineteenth century, however, increases in the complexity of mathematical arguments pushed mathematicians to develop new styles of reasoning that suppress algorithmic information and invoke descriptions of mathematical objects that abstract away the details of how those objects are represented. The goal was to obtain a powerful "conceptual" understanding without getting bogged down in computational details, but this had the effect of admitting mathematical theorems that are simply *false* on a direct computational reading.

There is still fairly uniform agreement today that computation is important to mathematics. But there are different views as to how best to address computational concerns. From a *constructive* point of view, it is a mistake to separate mathematics from its computational roots; every meaningful mathematical theorem should have a direct computational interpretation. From a *classical* point of view, it is more fruitful to maintain a separation of concerns: we can use one language and body of methods to write computer programs, while maintaining the freedom to use a nonconstructive theories and methods to reason about them. Lean is designed to support both of these approaches. Core parts of the library are developed constructively, but the system also provides support for carrying out classical mathematical reasoning.

Computationally, the purest part of dependent type theory avoids the use of `Prop` entirely. Inductive types and dependent function types can be viewed as data types, and terms of these types can be "evaluated" by applying reduction rules until no more rules can be applied. In principle, any closed term (that is, term with no free variables) of type ℕ should evaluate to a numeral, `succ (... (succ zero)...)`.

Introducing a proof-irrelevant `Prop` and marking theorems irreducible represents a first step towards separation of concerns. The intention is that elements of a type `p : Prop` should play no role in computation, and so the particular construction of a term `t : p` is "irrelevant" in that sense. One can still define computational objects that incorporate elements of type `Prop`; the point is that these elements can help us reason about the effects of the computation, but can be ignored when we extract "code" from the term. Elements of type `Prop` are not entirely innocuous, however. They include equations `s = t :` α for any type α, and such equations can be used as casts, to type check terms. Below, we will see examples of how such casts can block computation in the system. However, computation is still possible under an evaluation scheme that erases propositional content, ignores intermediate typing constraints, and reduces terms until they reach a normal form. This is precisely what Lean's virtual machine does.

Having adopted a proof-irrelevant `Prop`, one might consider it legitimate to use, for example, the law of the excluded middle, `p ∨ ¬p`, where `p` is any proposition. Of course, this, too, can block computation according to the rules of CIC, but it does not block bytecode evaluation, as described above. It is only the choice principles discussed in Section 11.5 that completely erase the distinction between the proof-irrelevant and data-relevant parts of the theory.

11.2 Propositional Extensionality

Propositional extensionality is the following axiom:

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

It asserts that when two propositions imply one another, they are actually equal. This is consistent with set-theoretic interpretations in which any element `a : Prop` is either empty or the singleton set `{*}`, for some distinguished element `*`. The axiom has the effect that equivalent propositions can be substituted for one another in any context:

```
section
variables a b c d e : Prop
variable p : Prop → Prop
```

(continues on next page)

(continued from previous page)

```
theorem thm₁ (h : a ↔ b) : (c ∧ a ∧ d → e) ↔ (c ∧ b ∧ d → e) :=
propext h ▶ iff.refl _

theorem thm₂ (h : a ↔ b) (h₁ : p a) : p b :=
propext h ▶ h₁
end
```

The first example could be proved more laboriously without `propext` using the fact that the propositional connectives respect propositional equivalence. The second example represents a more essential use of `propext`. In fact, it is equivalent to `propext` itself, a fact which we encourage you to prove.

Given any definition or theorem in Lean, you can use the `#print axioms` command to display the axioms it depends on.

```
#print axioms thm₁  -- propext
#print axioms thm₂  -- propext
```

11.3 Function Extensionality

Similar to propositional extensionality, function extensionality asserts that any two functions of type `Π x : α, β x` that agree on all their inputs are equal.

```
universes u₁ u₂

#check (@funext : ∀ {α : Type u₁} {β : α → Type u₂}
           {f₁ f₂ : Π (x : α), β x},
         (∀ (x : α), f₁ x = f₂ x) → f₁ = f₂)
```

From a classical, set-theoretic perspective, this is exactly what it means for two functions to be equal. This is known as an "extensional" view of functions. From a constructive perspective, however, it is sometimes more natural to think of functions as algorithms, or computer programs, that are presented in some explicit way. It is certainly the case that two computer programs can compute the same answer for every input despite the fact that they are syntactically quite different. In much the same way, you might want to maintain a view of functions that does not force you to identify two functions that have the same input / output behavior. This is known as an "intensional" view of functions.

In fact, function extensionality follows from the existence of quotients, which we describe in the next section. In the Lean standard library, therefore, `funext` is thus proved from the quotient construction.

Suppose that for `α : Type` we define the `set α := α → Prop` to denote the type of subsets of α, essentially identifying subsets with predicates. By combining `funext` and `propext`, we obtain an extensional theory of such sets:

```
def set (α : Type*) := α → Prop

namespace set

variable {α : Type*}

definition mem (x : α) (a : set α) := a x
notation e ∈ a := mem e a

theorem setext {a b : set α} (h : ∀ x, x ∈ a ↔ x ∈ b) : a = b :=
funext (assume x, propext (h x))
```

(continues on next page)

(continued from previous page)

```
end set
```

We can then proceed to define the empty set and set intersection, for example, and prove set identities:

```
definition empty : set α := λ x, false
local notation `∅` := empty

definition inter (a b : set α) : set α := λ x, x ∈ a ∧ x ∈ b
notation a ∩ b := inter a b

theorem inter_self (a : set α) : a ∩ a = a :=
setext (assume x, and_self _)

theorem inter_empty (a : set α) : a ∩ ∅ = ∅ :=
setext (assume x, and_false _)

theorem empty_inter (a : set α) : ∅ ∩ a = ∅ :=
setext (assume x, false_and _)

theorem inter.comm (a b : set α) : a ∩ b = b ∩ a :=
setext (assume x, and_comm _ _)
```

The following is an example of how function extensionality blocks computation inside the Lean kernel.

```
import data.nat.basic

def f₁ (x : ℕ) := x
def f₂ (x : ℕ) := 0 + x

theorem feq : f₁ = f₂ := funext (assume x, (zero_add x).symm)

def val : ℕ := eq.rec_on feq (0 : ℕ)

-- complicated!
#reduce val

-- evaluates to 0
#eval val
```

First, we show that the two functions `f₁` and `f₂` are equal using function extensionality, and then we cast `0` of type ℕ by replacing `f₁` by `f₂` in the type. Of course, the cast is vacuous, because ℕ does not depend on `f₁`. But that is enough to do the damage: under the computational rules of the system, we now have a closed term of ℕ that does not reduce to a numeral. In this case, we may be tempted to reduce the expression to `0`. But in nontrivial examples, eliminating cast changes the type of the term, which might make an ambient expression type incorrect. The virtual machine, however, has no trouble evaluating the expression to `0`. Here is a similarly contrived example that shows how `propext` can get in the way.

```
theorem tteq : (true ∧ true) = true := propext (and_true true)

def val : ℕ := eq.rec_on tteq 0

-- complicated!
#reduce val

-- evaluates to 0
```

(continues on next page)

(continued from previous page)

```
#eval val
```

Current research programs, including work on *observational type theory* and *cubical type theory*, aim to extend type theory in ways that permit reductions for casts involving function extensionality, quotients, and more. But the solutions are not so clear cut, and the rules of Lean's underlying calculus do not sanction such reductions.

In a sense, however, a cast does not change the meaning of an expression. Rather, it is a mechanism to reason about the expression's type. Given an appropriate semantics, it then makes sense to reduce terms in ways that preserve their meaning, ignoring the intermediate bookkeeping needed to make the reductions type correct. In that case, adding new axioms in `Prop` does not matter; by proof irrelevance, an expression in `Prop` carries no information, and can be safely ignored by the reduction procedures.

11.4 Quotients

Let α be any type, and let `r` be an equivalence relation on α. It is mathematically common to form the "quotient" `α / r`, that is, the type of elements of α "modulo" `r`. Set theoretically, one can view `α / r` as the set of equivalence classes of α modulo `r`. If `f : α → β` is any function that respects the equivalence relation in the sense that for every `x y : α`, `r x y` implies `f x = f y`, then `f` "lifts" to a function `f' : α / r → β` defined on each equivalence class `⟦x⟧` by `f' ⟦x⟧ = f x`. Lean's standard library extends the Calculus of Constructions with additional constants that perform exactly these constructions, and installs this last equation as a definitional reduction rule.

In its most basic form, the quotient construction does not even require `r` to be an equivalence relation. The following constants are built into Lean:

```
universes u v

constant quot : Π {α : Sort u}, (α → α → Prop) → Sort u

constant quot.mk :
  Π {α : Sort u} (r : α → α → Prop), α → quot r

axiom quot.ind :
  ∀ {α : Sort u} {r : α → α → Prop} {β : quot r → Prop},
    (∀ a, β (quot.mk r a)) → ∀ (q : quot r), β q

constant quot.lift :
  Π {α : Sort u} {r : α → α → Prop} {β : Sort u} (f : α → β),
    (∀ a b, r a b → f a = f b) → quot r → β
```

The first one forms a type `quot r` given a type α by any binary relation `r` on α. The second maps α to `quot α`, so that if `r : α → α → Prop` and `a : α`, then `quot.mk r a` is an element of `quot r`. The third principle, `quot.ind`, says that every element of `quot.mk r a` is of this form. As for `quot.lift`, given a function `f : α → β`, if `h` is a proof that `f` respects the relation `r`, then `quot.lift f h` is the corresponding function on `quot r`. The idea is that for each element `a` in α, the function `quot.lift f h` maps `quot.mk r a` (the `r`-class containing `a`) to `f a`, wherein `h` shows that this function is well defined. In fact, the computation principle is declared as a reduction rule, as the proof below makes clear.

```
variables α β : Type*
variable  r : α → α → Prop
variable  a : α

-- the quotient type
#check (quot r : Type*)
```

(continues on next page)

(continued from previous page)

```
-- the class of a
#check (quot.mk r a : quot r)

variable  f : α → β
variable   h : ∀ a₁ a₂, r a₁ a₂ → f a₁ = f a₂

-- the corresponding function on quot r
#check (quot.lift f h : quot r → β)

-- the computation principle
theorem thm : quot.lift f h (quot.mk r a) = f a := rfl
```

The four constants, `quot`, `quot.mk`, `quot.ind`, and `quot.lift` in and of themselves are not very strong. You can check that the `quot.ind` is satisfied if we take `quot r` to be simply α, and take `quot.lift` to be the identity function (ignoring `h`). For that reason, these four constants are not viewed as additional axioms:

```
#print axioms thm   -- no axioms
```

They are, like inductively defined types and the associated constructors and recursors, viewed as part of the logical framework.

What makes the `quot` construction into a bona fide quotient is the following additional axiom:

```
axiom quot.sound :
  ∀ {α : Type*} {r : α → α → Prop} {a b : α},
    r a b → quot.mk r a = quot.mk r b
```

This is the axiom that asserts that any two elements of α that are related by `r` become identified in the quotient. If a theorem or definition makes use of `quot.sound`, it will show up in the `#print axioms` command.

Of course, the quotient construction is most commonly used in situations when `r` is an equivalence relation. Given `r` as above, if we define `r'` according to the rule `r' a b` iff `quot.mk r a = quot.mk r b`, then it's clear that `r'` is an equivalence relation. Indeed, `r'` is the *kernel* of the function `a ↦ quot.mk r a`. The axiom `quot.sound` says that `r a b` implies `r' a b`. Using `quot.lift` and `quot.ind`, we can show that `r'` is the smallest equivalence relation containing `r`, in the sense that if `r''` is any equivalence relation containing `r`, then `r' a b` implies `r'' a b`. In particular, if `r` was an equivalence relation to start with, then for all `a` and `b` we have `r a b` iff `r' a b`.

To support this common use case, the standard library defines the notion of a *setoid*, which is simply a type with an associated equivalence relation:

```
class setoid (α : Type*) :=
(r : α → α → Prop) (iseqv : equivalence r)

namespace setoid
infix `≈` := setoid.r

variable {α : Type*}
variable [s : setoid α]
include s

theorem refl (a : α) : a ≈ a :=
(@setoid.iseqv α s).left a

theorem symm {a b : α} : a ≈ b → b ≈ a :=
λ h, (@setoid.iseqv α s).right.left h
```

(continues on next page)

(continued from previous page)

```
theorem trans {a b c : α} : a ≈ b → b ≈ c → a ≈ c :=
λ h₁ h₂, (@setoid.iseqv α s).right.right h₁ h₂
end setoid
```

Given a type α, a relation `r` on α, and a proof `p` that `r` is an equivalence relation, we can define `setoid.mk p` as an instance of the setoid class.

```
def quotient {α : Type*} (s : setoid α) :=
@quot α setoid.r
```

The constants `quotient.mk`, `quotient.ind`, `quotient.lift`, and `quotient.sound` are nothing more than the specializations of the corresponding elements of `quot`. The fact that type class inference can find the setoid associated to a type α brings a number of benefits. First, we can use the notation `a ≈ b` (entered with `\approx`) for `setoid.r a b`, where the instance of `setoid` is implicit in the notation `setoid.r`. We can use the generic theorems `setoid.refl`, `setoid.symm`, `setoid.trans` to reason about the relation. Specifically with quotients we can use the generic notation `⟦a⟧` for `quot.mk setoid.r` where the instance of `setoid` is implicit in the notation `setoid.r`, as well as the theorem `quotient.exact`:

```
variables {α : Type*} [setoid α] (a b : α)

#check (quotient.exact : ⟦a⟧ = ⟦b⟧ → a ≈ b)
```

Together with `quotient.sound`, this implies that the elements of the quotient correspond exactly to the equivalence classes of elements in α.

Recall that in the standard library, `α × β` represents the Cartesian product of the types α and β. To illustrate the use of quotients, let us define the type of *unordered* pairs of elements of a type α as a quotient of the type `α × α`. First, we define the relevant equivalence relation:

```
private definition eqv {α : Type*} (p₁ p₂ : α × α) : Prop :=
(p₁.1 = p₂.1 ∧ p₁.2 = p₂.2) ∨ (p₁.1 = p₂.2 ∧ p₁.2 = p₂.1)

infix `~` := eqv
```

The next step is to prove that `eqv` is in fact an equivalence relation, which is to say, it is reflexive, symmetric and transitive. We can prove these three facts in a convenient and readable way by using dependent pattern matching to perform case-analysis and break the hypotheses into pieces that are then reassembled to produce the conclusion.

```
open or

private theorem eqv.refl {α : Type*} :
  ∀ p : α × α, p ~ p :=
assume p, inl ⟨rfl, rfl⟩

private theorem eqv.symm {α : Type*} :
  ∀ p₁ p₂ : α × α, p₁ ~ p₂ → p₂ ~ p₁
| (a₁, a₂) (b₁, b₂) (inl ⟨a₁b₁, a₂b₂⟩) :=
    inl ⟨symm a₁b₁, symm a₂b₂⟩
| (a₁, a₂) (b₁, b₂) (inr ⟨a₁b₂, a₂b₁⟩) :=
    inr ⟨symm a₂b₁, symm a₁b₂⟩

private theorem eqv.trans {α : Type*} :
  ∀ p₁ p₂ p₃ : α × α, p₁ ~ p₂ → p₂ ~ p₃ → p₁ ~ p₃
| (a₁, a₂) (b₁, b₂) (c₁, c₂)
    (inl ⟨a₁b₁, a₂b₂⟩) (inl ⟨b₁c₁, b₂c₂⟩) :=
  inl ⟨trans a₁b₁ b₁c₁, trans a₂b₂ b₂c₂⟩
```

(continues on next page)

(continued from previous page)

```
| (a₁, a₂) (b₁, b₂) (c₁, c₂)
    (inl ⟨a₁b₁, a₂b₂⟩) (inr ⟨b₁c₂, b₂c₁⟩) :=
  inr ⟨trans a₁b₁ b₁c₂, trans a₂b₂ b₂c₁⟩
| (a₁, a₂) (b₁, b₂) (c₁, c₂)
    (inr ⟨a₁b₂, a₂b₁⟩) (inl ⟨b₁c₁, b₂c₂⟩) :=
  inr ⟨trans a₁b₂ b₂c₂, trans a₂b₁ b₁c₁⟩
| (a₁, a₂) (b₁, b₂) (c₁, c₂)
    (inr ⟨a₁b₂, a₂b₁⟩) (inr ⟨b₁c₂, b₂c₁⟩) :=
  inl ⟨trans a₁b₂ b₂c₁, trans a₂b₁ b₁c₂⟩

private theorem is_equivalence (α : Type*) :
  equivalence (@eqv α) :=
mk_equivalence (@eqv α) (@eqv.refl α) (@eqv.symm α)
  (@eqv.trans α)
```

We open the namespaces `or` and `eq` to be able to use `or.inl`, `or.inr`, and `eq.trans` more conveniently.

Now that we have proved that `eqv` is an equivalence relation, we can construct a `setoid (α × α)`, and use it to define the type `uprod α` of unordered pairs.

```
instance uprod.setoid (α : Type*) : setoid (α × α) :=
setoid.mk (@eqv α) (is_equivalence α)

definition uprod (α : Type*) : Type* :=
quotient (uprod.setoid α)

namespace uprod
definition mk {α : Type*} (a₁ a₂ : α) : uprod α :=
⟦(a₁, a₂)⟧

local notation `{` a₁ `,` a₂ `}` := mk a₁ a₂
end uprod
```

Notice that we locally define the notation `{a₁, a₂}` for ordered pairs as `⟦(a₁, a₂)⟧`. This is useful for illustrative purposes, but it is not a good idea in general, since the notation will shadow other uses of curly brackets, such as for records and sets.

We can easily prove that `{a₁, a₂} = {a₂, a₁}` using `quot.sound`, since we have `(a₁, a₂) ~ (a₂, a₁)`.

```
theorem mk_eq_mk {α : Type*} (a₁ a₂ : α) :
  {a₁, a₂} = {a₂, a₁} :=
quot.sound (inr ⟨rfl, rfl⟩)
```

To complete the example, given `a : α` and `u : uprod α`, we define the proposition `a ∈ u` which should hold if `a` is one of the elements of the unordered pair `u`. First, we define a similar proposition `mem_fn a u` on (ordered) pairs; then we show that `mem_fn` respects the equivalence relation `eqv` with the lemma `mem_respects`. This is an idiom that is used extensively in the Lean standard library.

```
private definition mem_fn {α : Type*} (a : α) :
  α × α → Prop
| (a₁, a₂) := a = a₁ ∨ a = a₂

-- auxiliary lemma for proving mem_respects
private lemma mem_swap {α : Type*} {a : α} :
  ∀ {p : α × α}, mem_fn a p = mem_fn a (⟨p.2, p.1⟩)
| (a₁, a₂) := propext (iff.intro
    (λ l : a = a₁ ∨ a = a₂,
```

(continues on next page)

(continued from previous page)

```
      or.elim l (λ h₁, inr h₁) (λ h₂, inl h₂))
    (λ r : a = a₂ ∨ a = a₁,
      or.elim r (λ h₁, inr h₁) (λ h₂, inl h₂)))

private lemma mem_respects {α : Type*} :
  ∀ {p₁ p₂ : α × α} (a : α),
    p₁ ~ p₂ → mem_fn a p₁ = mem_fn a p₂
| (a₁, a₂) (b₁, b₂) a (inl ⟨a₁b₁, a₂b₂⟩) :=
  by { dsimp at a₁b₁, dsimp at a₂b₂, rw [a₁b₁, a₂b₂] }
| (a₁, a₂) (b₁, b₂) a (inr ⟨a₁b₂, a₂b₁⟩) :=
  by { dsimp at a₁b₂, dsimp at a₂b₁, rw [a₁b₂, a₂b₁],
       apply mem_swap }

def mem {α : Type*} (a : α) (u : uprod α) : Prop :=
quot.lift_on u (λ p, mem_fn a p) (λ p₁ p₂ e, mem_respects a e)

local infix `∈` := mem

theorem mem_mk_left {α : Type*} (a b : α) : a ∈ {a, b} :=
inl rfl

theorem mem_mk_right {α : Type*} (a b : α) : b ∈ {a, b} :=
inr rfl

theorem mem_or_mem_of_mem_mk {α : Type*} {a b c : α} :
  c ∈ {a, b} → c = a ∨ c = b :=
λ h, h
```

For convenience, the standard library also defines `quotient.lift₂` for lifting binary functions, and `quotient.ind₂` for induction on two variables.

We close this section with some hints as to why the quotient construction implies function extenionality. It is not hard to show that extensional equality on the `Π x : α, β x` is an equivalence relation, and so we can consider the type `extfun α β` of functions "up to equivalence." Of course, application respects that equivalence in the sense that if `f₁` is equivalent to `f₂`, then `f₁ a` is equal to `f₂ a`. Thus application gives rise to a function `extfun_app : extfun α β → Π x : α, β x`. But for every `f`, `extfun_app ⟦f⟧` is definitionally equal to `λ x, f x`, which is in turn definitionally equal to `f`. So, when `f₁` and `f₂` are extensionally equal, we have the following chain of equalities:

```
f₁ = extfun_app ⟦f₁⟧ = extfun_app ⟦f₂⟧ = f₂
```

As a result, `f₁` is equal to `f₂`.

11.5 Choice

To state the final axiom defined in the standard library, we need the `nonempty` type, which is defined as follows:

```
class inductive nonempty (α : Sort*) : Prop
| intro : α → nonempty
```

Because `nonempty α` has type `Prop` and its constructor contains data, it can only eliminate to `Prop`. In fact, `nonempty α` is equivalent to `∃ x : α, true`:

```
example (α : Type*) : nonempty α ↔ ∃ x : α, true :=
iff.intro (λ ⟨a⟩, ⟨a, trivial⟩) (λ ⟨a, h⟩, ⟨a⟩)
```

Our axiom of choice is now expressed simply as follows:

```
axiom choice {α : Sort*} : nonempty α → α
```

Given only the assertion h that α is nonempty, choice h magically produces an element of α. Of course, this blocks any meaningful computation: by the interpretation of Prop, h contains no information at all as to how to find such an element.

This is found in the classical namespace, so the full name of the theorem is classical.choice. The choice principle is equivalent to the principle of *indefinite description*, which can be expressed with subtypes as follows:

```
noncomputable theorem indefinite_description
    {α : Sort*} (p : α → Prop) :
  (∃ x, p x) → {x // p x} :=
λ h, choice (let ⟨x, px⟩ := h in ⟨⟨x, px⟩⟩)
```

Because it depends on choice, Lean cannot generate bytecode for indefinite_description, and so requires us to mark the definition as noncomputable. Also in the classical namespace, the function some and the property some_spec decompose the two parts of the output of indefinite_description:

```
noncomputable def some {a : Sort*} {p : a → Prop}
  (h : ∃ x, p x) : a :=
subtype.val (indefinite_description p h)

theorem some_spec {a : Sort*} {p : a → Prop}
  (h : ∃ x, p x) : p (some h) :=
subtype.property (indefinite_description p h)
```

The choice principle also erases the distinction between the property of being nonempty and the more constructive property of being inhabited:

```
noncomputable theorem inhabited_of_nonempty {α : Type*} :
  nonempty α → inhabited α :=
λ h, choice (let ⟨a⟩ := h in ⟨⟨a⟩⟩)
```

In the next section, we will see that propext, funext, and choice, taken together, imply the law of the excluded middle and the decidability of all propositions. Using those, one can strengthen the principle of indefinite description as follows:

```
#check (@strong_indefinite_description :
        Π {α : Sort*} (p : α → Prop),
          nonempty α → {x // (∃ (y : α), p y) → p x})
```

Assuming the ambient type α is nonempty, strong_indefinite_description p produces an element of α satisfying p if there is one. The data component of this definition is conventionally known as *Hilbert's epsilon function*:

```
#check (@epsilon : Π {α : Sort*} [nonempty α],
                    (α → Prop) → α)

#check (@epsilon_spec : ∀ {a : Sort*} {p : a → Prop}
           (hex : ∃ (y : a), p y),
         p (@epsilon _ (nonempty_of_exists hex) p))
```

11.6 The Law of the Excluded Middle

The law of the excluded middle is the following

```
#check (@em : ∀ (p : Prop), p ∨ ¬p)
```

Diaconescu's theorem states that the axiom of choice is sufficient to derive the law of excluded middle. More precisely, it shows that the law of the excluded middle follows from `classical.choice`, `propext`, and `funext`. We sketch the proof that is found in the standard library.

First, we import the necessary axioms, fix a parameter, `p`, and define two predicates `U` and `V`:

```
open classical

section diaconescu
parameter  p : Prop

def U (x : Prop) : Prop := x = true ∨ p
def V (x : Prop) : Prop := x = false ∨ p

lemma exU : ∃ x, U x := ⟨true, or.inl rfl⟩
lemma exV : ∃ x, V x := ⟨false, or.inl rfl⟩

end diaconescu
```

If `p` is true, then every element of `Prop` is in both `U` and `V`. If `p` is false, then `U` is the singleton `true`, and `V` is the singleton `false`.

Next, we use `some` to choose an element from each of `U` and `V`:

```
noncomputable def u := some exU
noncomputable def v := some exV

lemma u_def : U u := some_spec exU
lemma v_def : V v := some_spec exV
```

Each of `U` and `V` is a disjunction, so `u_def` and `v_def` represent four cases. In one of these cases, `u = true` and `v = false`, and in all the other cases, `p` is true. Thus we have:

```
lemma not_uv_or_p : u ≠ v ∨ p :=
or.elim u_def
  (assume hut : u = true,
    or.elim v_def
      (assume hvf : v = false,
        have hne : u ≠ v,
          from eq.symm hvf ▶ eq.symm hut ▶ true_ne_false,
        or.inl hne)
      (assume hp : p, or.inr hp))
  (assume hp : p, or.inr hp)
```

On the other hand, if `p` is true, then, by function extensionality and propositional extensionality, `U` and `V` are equal. By the definition of `u` and `v`, this implies that they are equal as well.

```
lemma p_implies_uv : p → u = v :=
assume hp : p,
have hpred : U = V, from
  funext (assume x : Prop,
```

(continues on next page)

(continued from previous page)

```
    have hl : (x = true ∨ p) → (x = false ∨ p), from
      assume a, or.inr hp,
    have hr : (x = false ∨ p) → (x = true ∨ p), from
      assume a, or.inr hp,
    show (x = true ∨ p) = (x = false ∨ p), from
      propext (iff.intro hl hr)),
have h₀ : ∀ exU exV,
    @classical.some _ U exU = @classical.some _ V exV,
  from hpred ▸ λ exU exV, rfl,
show u = v, from h₀ _ _
```

Putting these last two facts together yields the desired conclusion:

```
theorem em : p ∨ ¬p :=
have h : ¬(u = v) → ¬p, from mt p_implies_uv,
  or.elim not_uv_or_p
    (assume hne : ¬(u = v), or.inr (h hne))
    (assume hp : p, or.inl hp)
```

Consequences of excluded middle include double-negation elimination, proof by cases, and proof by contradiction, all of which are described in Section 3.5. The law of the excluded middle and propositional extensionality imply propositional completeness:

```
theorem prop_complete (a : Prop) : a = true ∨ a = false :=
or.elim (em a)
  (λ t, or.inl (propext (iff.intro (λ h, trivial) (λ h, t))))
  (λ f, or.inr (propext (iff.intro (λ h, absurd h f)
                                   (λ h, false.elim h))))
```

Together with choice, we also get the stronger principle that every proposition is decidable. Recall that the class of `decidable` propositions is defined as follows:

```
class inductive decidable (p : Prop)
| is_false : ¬ p → decidable
| is_true :  p → decidable
```

In contrast to `p ∨ ¬ p`, which can only eliminate to `Prop`, the type `decidable p` is equivalent to the sum type `p ⊕ ¬ p`, which can eliminate to any type. It is this data that is needed to write an if-then-else expression.

As an example of classical reasoning, we use `some` to show that if `f : α → β` is injective and α is inhabited, then `f` has a left inverse. To define the left inverse `linv`, we use a dependent if-then-else expression. Recall that `if h : c then t else e` is notation for `dite c (λ h : c, t) (λ h : ¬ c, e)`. In the definition of `linv`, choice is used twice: first, to show that `(∃ a : A, f a = b)` is "decidable," and then to choose an `a` such that `f a = b`. Notice that we make `prop_decidable` a local instance to justify the if-then-else expression. (See also the discussion in Section 10.4.)

```
open classical function
local attribute [instance] prop_decidable

noncomputable definition linv {α β : Type*} [h : inhabited α]
  (f : α → β) : β → α :=
λ b : β, if ex : (∃ a : α, f a = b) then some ex else arbitrary α

theorem linv_comp_self {α β : Type*} {f : α → β}
    [inhabited α] (inj : injective f) :
  linv f ∘ f = id :=
```

(continues on next page)

(continued from previous page)

```
funext (assume a,
  have ex  : ∃ a₁ : α, f a₁ = f a, from exists.intro a rfl,
  have   feq : f (some ex) = f a, from some_spec ex,
  calc linv f (f a) = some ex :  dif_pos ex
             ...    = a       :  inj feq)
```

From a classical point of view, `linv` is a function. From a constructive point of view, it is unacceptable; because there is no way to implement such a function in general, the construction is not informative.

BIBLIOGRAPHY

[CoHu88] Thierry Coquand and Gerard Huet. The calculus of constructions. Inf. Comput., 76(2-3):95–120, February 1988.

[Dybj94] Peter Dybjer. Inductive families. Formal Asp. Comput., 6(4):440–465, 1994.

[GoMM06] Healfdene Goguen, Conor McBride, and James McKinna. Eliminating dependent pattern matching. In Kokichi Futatsugi, Jean-Pierre Jouannaud, and José Meseguer, editors, Algebra, Meaning, and Computation, Essays Dedicated to Joseph A. Goguen on the Occasion of His 65th Birthday, volume 4060 of Lecture Notes in Computer Science, pages 521–540. Springer, 2006.