Theorem Proving in Lean

Jeremy Avigad
Leonardo de Moura
Soonho Kong

Version d0dd6d0, updated at 2017-01-30 19:53:44 -0500

Contents

1

Introduction

Please note that this is the tutorial for Lean 2, which allows the use of homotopy type theory (HoTT). It is *not* the tutorial for the current version of Lean.

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.

About Lean

The *Lean* project was launched by Leonardo de Moura at Microsoft Research Redmond in 2012. It is an ongoing, long-term effort, and much of the potential for automation will be realized only gradually over time. Lean is released under the Apache 2.0 license, a permissive open source license that permits others to use and extend the code and mathematical libraries freely.

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

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

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. 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.

In fact, Lean supports two versions of dependent type theory. The first is a variant of a system known as the *Calculus of Inductive Constructions*[1, 4], or *CIC*. This is the system used by Lean's standard library, and is the focus of this tutorial. The second version of dependent type theory implements an axiomatic framework for homotopy type theory, which we will discuss in a later chapter.

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

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

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

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

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

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

Acknowledgments

This tutorial is an open access project maintained on Github. Many people have contributed to the effort, providing corrections, suggestions, examples, and text. We are grateful to Ulrik Buchholz, Nathan Carter, Amine Chaieb, Floris van Doorn, Anthony Hart, Sean Leather, Christopher John Mazey, Daniel Velleman, and Théo Zimmerman for their contributions, and we apologize to those whose names we have inadvertently omitted.

2

Dependent Type Theory

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

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.

```
import standard
open bool nat

/- 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"
```

The first command, `import standard`, tells Lean that we intend to use the standard library. The next command, `open bool nat`, tells Lean that we will use constants, facts, and notations from the theory of the booleans and the theory of natural numbers. In technical terms, `bool` and `nat` are *namespaces*; you will learn more about them later. To shorten the examples, we will usually hide the relevant imports when they have already been made explicit in a previous example.

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

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

What makes simple type theory powerful is that one can build new types out of others. For example, if `A` and `B` are types, `A → B` denotes the type of functions from `A` to `B`, and `A × B` denotes the cartesian product, that is, the type of ordered pairs consisting of an element of `A` paired with an element of `B`.

```
open prod   -- makes notation for the product available

constants m n : nat

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

```
constant g' : nat → (nat → nat)  -- has the same type as g!
constant h : nat × nat → nat

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

check f               -- ℕ → ℕ
check f n             -- ℕ
check g m n           -- ℕ
check g m             -- ℕ → ℕ
check pair m n        -- ℕ × ℕ
check pr1 p           -- ℕ
check pr2 p           -- ℕ
check pr1 (pair m n)  -- ℕ
check pair (pr1 p) n  -- ℕ × ℕ
check F f             -- ℕ
```

The symbol ℕ is notation for `nat`; you can enter it by typing `\nat`. 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, `pair m n` denotes the ordered pair of `m` and `n`, and if `p` is a pair, `pr1 p` and `pr2 p` denote the two projections.

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                -- Type₁
check bool               -- Type₁
check nat → bool          -- Type₁
check nat × bool          -- Type₁
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 $\texttt{Type}_1$. We will explain the subscripted `1` in a moment. We can also declare new constants and constructors for types:

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

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

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

```
constants A B : Type

check prod            -- Type → Type → Type
check prod A          -- Type → Type
check prod A B        -- Type
check prod nat nat    -- Type₁
```

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

```
import data.list
open list

constant A : Type

check list       -- Type → Type
check list A     -- Type
check list nat   -- Type₁
```

We will see that the ability to treat type constructors as instances of ordinary mathematical functions is a powerful feature of dependent type theory.

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. But there is a circularity lurking nearby. `Type` itself is an expression like `nat`; if `nat` has a type, shouldn't `Type` have a type as well?

```
check Type      -- Type
```

Lean's output seems to indicates that `Type` is an element of itself. But this is misleading. Russell's paradox shows that it is inconsistent with the other axioms of set theory to assume the existence of a set of all sets, and one can derive a similar paradox in dependent type theory. So, is Lean inconsistent?

What is going on is that Lean's foundational fragment actually has a hierarchy of types,

```
Type.{1} : Type.{2} : Type.{3} : ....
```

Think of `Type.{1}` as a universe of "small" or "ordinary" types. `Type.{2}` is then a larger universe of types, which contains `Type.{1}` as an element. When we declare a constant `A : Type`, Lean implicitly creates a variable `u`, and declares `A : Type.{u}`. In other words, `A` is a type in some unspecified universe. The expression `A` is then *polymorphic*; whenever it appears, Lean silently tries to infer which universe `A` lives in, maintaining as much generality as possible.

You can ask Lean's pretty printer to make this information explicit, and use additional annotations to specify universe levels explicitly.

```
constants A B : Type
check A                -- A : Type
check B                -- B : Type
check Type             -- Type : Type
check Type → Type      -- Type → Type : Type

set_option pp.universes true    -- display universe information

check A                -- A.{l_1} : Type.{l_1}
check B                -- B.{l_1} : Type.{l_1}
check Type             -- Type.{l_1} : Type.{l_1 + 1}
check Type → Type      -- Type.{l_1} → Type.{l_2} : Type.{imax (l_1+1) (l_2+1)}

universe u
constant C : Type.{u}
check C                -- C : Type.{u}
check A → C            -- A.{l_1} → C : Type.{imax l_1 u}

universe variable v
constants D E : Type
check D → E            -- D.{l_1} → E.{l_2} : Type.{imax l_1 l_2}
check D.{v} → E.{v}    -- D.{v} → E.{v} : Type.{v}
```

The command `universe u` creates a fixed universe parameter. In contrast, in the last example, the universe variable `v` is only used to put `D` and `E` in the same type universe. When `D.{v} → E.{v}` occurs in a more elaborate context, Lean is constrained to assign the same universe parameter to both.

You should not worry about the meaning of `imax` right now. Universe contraints are subtle, but the good news is that Lean handles them pretty well. As a result, in ordinary situations you can ignore the universe parameters and simply write `Type`, leaving the "universe management" to Lean.

Function Abstraction and Evaluation

We have seen that if we have `m n : nat`, then we have `pair 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 `pr1 p : nat` and `pr2 p : nat`. This gives us a way of "using" a pair, by extracting its two components.

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

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

```
import data.nat data.bool
open nat bool

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

Here are some more abstract examples:

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

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

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

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

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

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

check λ x : A, x        -- the identity function on A
check λ x : A, b        -- a constant function on A
check λ x : A, g (f x)  -- the composition of g and f
check λ x, g (f x)      -- (Lean can figure out the type of x)

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

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

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

Think about what these expressions mean. The last, for example, denotes the function that takes three types, `A`, `B`, and `C`, and two functions, `g : B → C` and `f : A → B`, and returns the composition of `g` and `f`. (Making sense of the type of this function requires an understanding of dependent products, which we will explain below.) Within a lambda expression `λ x : A, t`, the variable `x` is a "bound variable": it is really a placeholder, whose "scope" does not extend beyond `t`. For example, the variable `b` in the expression `λ (b : B) (x : A), x` has nothing to do with the constant `b` declared earlier. In fact, the expression denotes the same function as `λ (u : B) (z : A), z`. Formally, the expressions that are the same up to a renaming of bound variables are called *alpha equivalent*, and are considered "the same." Lean recognizes this equivalence.

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

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

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

check (λ v u x, v (u x)) g f a      -- C

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

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

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

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

eval (λ v u x, v (u x)) g f a      -- g (f a)

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

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

```
import data.nat data.prod data.bool
open nat prod bool

constants m n : nat
constant b : bool

print "reducing pairs"
eval pr1 (pair m n)   -- m
eval pr2 (pair m n)   -- n

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

print "reducing arithmetic expressions"
eval n + 0            -- n
eval n + 2            -- succ (succ n)
eval (2 : nat) + 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.

Introducing Definitions

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

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

definition gfa : C := g (f a)

check gfa    -- C
print gfa    -- g (f a)

-- We can omit the type when Lean can figure it out.
definition gfa' := g (f a)
print gfa'

definition gfha := g (f (h a))
print gfha

definition g_comp_f : A → C := λ x, g (f x)
print g_comp_f
```

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

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

```
definition g_comp_f (x : A) : C := g (f x)
print g_comp_f
```

The net effect is the same as the previous definition.

Here are some more examples of definitions, this time in the context of arithmetic:

```
import data.nat
open nat

constants (m n : nat) (p q : bool)

definition m_plus_n : nat := m + n
check m_plus_n
print m_plus_n

-- again, Lean can infer the type
definition m_plus_n' := m + n
```

```
print m_plus_n'

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

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

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

eval do_twice double 2    -- 8
```

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 : ((nat → nat) → (nat → nat)) → (nat → nat) → (nat → nat)` which iterates *its* argument twice, so that `Do_Twice do_twice` a function which iterates *its* input four times, and evaluate `Do_Twice do_twice double 2`.

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

```
import data.prod
open prod

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

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

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`.

```
import data.nat
open nat

constant n : ℕ
check let y := n + n in y * y

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

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

```
constant n : ℕ
check let y := n + n, z := y + y in z * z
```

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.

```
import data.nat
open nat

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

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

Variables and Sections

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

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

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

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

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

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

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

```
variables (A B C : Type)

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

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

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

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

print compose
print do_twice
print do_thrice
```

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

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

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

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

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

```
  definition do_thrice := h (h (h x))
end useful
```

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

You do not have to indent the lines within a section, since Lean treats any blocks of returns, spaces, and tabs equivalently as whitespace. Nor do you have to name a section, which is to say, you can use an anonymous `section` / `end` pair. If you do name a section, however, you have to close it using the same name. Sections can also be nested, which allows you to declare new variables incrementally.

Sections provide us with a general scoping mechanism that governs more than the insertion of variables. For example, recall that the `open` command allows us to invoke identifiers and notation, using *namespaces*, which will be discussed below. The effects of an `open` command are also limited to the section in which it occurs, which provides useful ways of managing the background context while we work with Lean.

Namespaces

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

```
namespace foo
  constant A : Type
  constant a : A
  constant f : A → A

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

  print "inside foo"

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

print "outside the namespace"

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

```
open foo

print "opened foo"

check A
check a
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 module, we will want to open one or more of the namespaces it contains, to have access to the short identifiers, notations, and so on. But sometimes we will want to leave this information hidden, for example, when they conflict with identifiers and notations in another namespace we want to use. Thus namespaces give us a way to manage our working environment.

For example, when we work with the natural numbers, we usually want access to the function `add`, and its associated notation, `+`. The command `open nat` makes these available to us.

```
import data.nat   -- imports the nat module

check nat.add
check nat.zero

open nat -- imports short identifiers, notations, etc. into the context

check add
check zero

constants m n : nat

check m + n
check 0
check m + 0
```

Like sections, namespaces can be nested:

```
namespace foo
  constant A : Type
  constant a : A
  constant f : A → A

  definition fa : A := f a

  namespace bar
    definition ffa : A := 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
  constant A : Type
  constant a : A
  constant f : A → A

  definition fa : A := f a
end foo

check foo.A
check foo.f

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

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

Namespaces and sections serve different purposes: namespaces organize data and sections declare variables for insertion in theorems. A namespace can be viewed as a special kind of section, however. In particular, if you use the `variable` command within a namespace, its scope is limited to the namespace. Similarly, if you use an `open` command within a namespace, its effects disappear when the namespace is closed.

Dependent Types

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

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

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 `nat`, `bool`, or an arbitrary type `A` to behave the same way. So it makes sense to take the type to be the first argument to `cons`, so that for any type, `A`, `cons A` is the insertion function for lists of type `A`. In other words, for every `A`, `cons A` is the function that takes an element `a : A` and a list `l : list A`, and returns a new list, so we have `cons A a l : list A`.

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

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

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

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

```
namespace hide
constant list : Type → Type

namespace list
  constant cons : Π A : Type, A → list A → list A -- type the product as "\Pi"
  constant nil : Π A : Type, list A            -- the empty list
  constant head : Π A : Type, list A → A        -- returns the first element
```

```
  constant tail : Π A : Type, list A → list A  -- returns the remainder
  constant append : Π A : Type, list A → list A → list A -- concatenates two lists
end list
end hide
```

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

```
import data.list
open list

check list      -- Type → Type

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

There is a subtlety in the definition of `head`: 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 Type Classes.

Vector operations are handled similarly:

```
import data.nat
open nat

constant vec : Type → nat → Type

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

In the coming chapters, you will come across many instances of dependent types. Here we will mention just one more important and illustrative example, the *Sigma types*, `Σ x : A, B x`, sometimes also known as *dependent pairs*. These are, in a sense, companions to the Pi types. The type `Σ x : A, B x` denotes the type of pairs `sigma.mk a b` where `a : A` and `b : B a`. You can also use angle brackets `<a, b>` as notation for `sigma.mk a b`. (To type these brackets, use the shortcuts `\<` and `\>`.) Just as Pi types `Π x : A, B x` generalize the notion of a function type `A → B` by allowing `B` to depend on `A`, Sigma types `Σ x : A, B x` generalize the cartesian product `A × B` in the same way: in the expression `sigma.mk a b`, the type of the second element of the pair, `b : B a`, depends on the first element of the pair, `a : A`.

```
import data.sigma
open sigma

variable A : Type
variable B : A → Type
variable a : A
variable b : B a

check sigma.mk a b  -- Σ (a : A), B a
check ⟨a, b⟩         -- Σ (a : A), B a
check pr1 ⟨a, b⟩     -- A
check pr₁ ⟨a, b⟩     -- alternative notation; use \_1 for the subscript
check pr2 ⟨a, b⟩     -- B (pr₁ ⟨a, b⟩)
check pr₂ ⟨a, b⟩     -- alternative notation

eval pr1 ⟨a, b⟩      -- a
eval pr2 ⟨a, b⟩      -- b
```

Note, by the way, that the identifiers `pr1` and `pr2` are also used for the cartesian product type. The notations are made available when you open the namespaces `prod` and `sigma` respectively; if you open both, the identifier is simply overloaded. Without opening the namespaces, you can refer to them as `prod.pr1`, `prod.pr2`, `sigma.pr1`, and `sigma.pr2`.

If you open the namespaces `prod.ops` and `sigma.ops`, you can, moreover, use additional convenient notation for the projections:

```
import data.sigma data.prod

variable A : Type
variable B : A → Type
variable a : A
variable b : B a
variables C D : Type
variables (c : C) (d : D)

open sigma.ops
open prod.ops

eval (a, b).1
eval (a, b).2
eval ⟨c, d⟩.1
eval ⟨c, d⟩.2
```

Implicit Arguments

Suppose we have an implementation of lists as described above.

```
namespace hide
constant list : Type → Type

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

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

```
open hide.list

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

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

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

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

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

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

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

```
end list

open hide.list

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

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

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

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

check ident      -- ?A → ?A

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

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

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

In the first `check` command, the inscription `?A` indicates that the type of `ident` depends on a "placeholder," or "metavariable," that should, in general, be inferred from the context. The output of the second `check` command is somewhat verbose: it indicates that the placeholder, `?A`, can itself depend on any of the variables `A`, `B`, `a`, and `b` that are in the context. If this additional information is annoying, you can suppress it by writing `@ident`, as described below. Alternatively, you can set an option to avoid printing these arguments:

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

set_option pp.metavar_args false
check ident      -- ?A → ?A
```

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

```
section
  variable {A : Type}
  variable x : A
```

```
  definition ident := x
end

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

check ident
check ident a
check ident b
```

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

Lean has very complex mechanisms for instantiating implicit arguments, and we will see that they can be used to infer function types, predicates, and even proofs. The process of instantiating "holes," or "placeholder," in a term is often known as *elaboration*. As this tutorial progresses, we will gradually learn more about what Lean's powerful elaborator can do, and we will discuss the elaborator in depth in Chapter Elaboration and Unification.

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

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

Section More on Implicit Arguments explains another useful annotation, `!`, which makes explicit arguments implicit. In a sense, it is the opposite of `@`, and is most useful in the context of theorem proving, which we will turn to next.

3

Propositions and Proofs

By now, you have seen how to define some elementary notions in dependent type theory. You have also seen that it is possible to import objects that are defined in Lean's library. In this chapter, we will explain how mathematical propositions and proofs are expressed in the language of dependent type theory, so that you can start proving assertions about the objects and notations that have been defined. The encoding we use here is specific to the standard library; we will discuss proofs in *homotopy type theory* in a later chapter.

Propositions as Types

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

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

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

variables p q r : Prop
check and p q                      -- Prop
check or (and p q) r               -- Prop
check implies (and p q) (and q p)  -- Prop
```

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

```
constant Proof : Prop → Type

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

variables p q : Prop
check and_comm p q      -- Proof (implies (and p q) (and q p))
```

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

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

We could represent this as follows:

```
constant modus_ponens (p q : Prop) : Proof (implies p q) →  Proof p → Proof q
```

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

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

We could render this as follows:

```
constant implies_intro (p q : Prop) : (Proof p → Proof q) → Proof (implies p q).
```

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

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

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

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

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

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

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

Lean also supports an alternative *proof relevant kernel*, which forms the basis for homotopy type theory. We will return to this topic in a later chapter.

Working with Propositions as Types

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

```
constants p q : Prop

theorem t1 : p → q → p := λ Hp : p, λ Hq : q, Hp
```

This looks exactly like the definition of the constant function in the last chapter, the only difference being that the arguments are elements of `Prop` rather than `Type`. Intuitively, our proof of `p → q → p` assumes `p` and `q` are true, and uses the first hypothesis (trivially) to establish that the conclusion, `p`, is true.

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

There are a few pragmatic differences between definitions and theorems, however, that you will learn more about in Chapter Building Theories and Proofs. In normal circumstances, it is never necessary to unfold the "definition" of a theorem; by proof irrelevance, any two proofs of that theorem are definitionally equal. Once the proof of a theorem is complete, typically we only need to know that the proof exists; it doesn't matter what the proof is. In light of that fact, Lean tags proofs as *irreducible*, which serves as a hint to the parser (more precisely, the *elaborator*) that there is generally no need to unfold it when processing a file. Moreover, for efficiency purposes, Lean treats theorems as axiomatic constants within the file in which they are defined. This makes it possible to process and check theorems in parallel, since theorems later in a file do not make use of the contents of earlier proofs.

As with definitions, the `print` command will show you the proof of a theorem, with a slight twist: if you want to print a theorem in the same file in which it is defined, you need to use the `reveal` command to force Lean to use the theorem itself, rather than its axiomatic surrogate.

```
theorem t1 : p → q → p := λ Hp : p, λ Hq : q, Hp

reveal t1
print t1
```

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

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

```
theorem t1 : p → q → p :=
assume Hp : p,
assume Hq : q,
Hp
```

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

```
theorem t1 : p → q → p :=
assume Hp : p,
assume Hq : q,
show p, from Hp
```

Adding such extra information can improve the clarity of a proof and help detect errors when writing a proof. The `show` command does nothing more than annotate the type, and, internally, all the presentations of `t1` that we have seen produce the same term. Lean also allows you to use the alternative syntax `lemma` and `corollary` instead of theorem:

```
lemma t1 : p → q → p :=
assume Hp : p,
assume Hq : q,
show p, from Hp
```

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

```
theorem t1 (Hp : p) (Hq : q) : p := Hp

check t1 -- p → q → p
```

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

```
axiom Hp : p

theorem t2 : q → p := t1 Hp
```

Here, the `axiom` command is alternative syntax for `constant`. Declaring a "constant" `Hp : p` is tantamount to declaring that `p` is true, as witnessed by `Hp`. Applying the theorem `t1 : p → q → p` to the fact `Hp : p` that `p` is true yields the theorem `t2 : q → p`.

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

```
theorem t1 (p q : Prop) (Hp : p) (Hq : q) : p := Hp
check t1
```

The type of `t1` is now `∀ p q : Prop, p → q → p`. We can read this as the assertion "for every pair of propositions `p q`, we have `p → q → p`". The symbol ∀ is alternate syntax for Π, and later we will see how Pi types let us model universal quantifiers more generally. For the moment, however, we will focus on theorems in propositional logic, generalized over the propositions. We will tend to work in sections with variables over the propositions, so that they are generalized for us automatically.

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

```
theorem t1 (p q : Prop) (Hp : p) (Hq : q) : p := Hp

variables p q r s : Prop

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

variable H : r → s
check t1 (r → s) (s → r) H  -- (s → r) → r → s
```

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

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

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

```
variables p q r s : Prop

theorem t2 (H1 : q → r) (H2 : p → q) : p → r :=
assume H3 : p,
show r, from H1 (H2 H3)
```

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

Lean allows the alternative syntax `premise` and `premises` for `variable` and `variables`. This makes sense, of course, for variables whose type is an element of `Prop`. The following definition of `t2` has the same net effect as the preceding one.

```
variables p q r s : Prop
premises (H1 : q → r) (H2 : p → q)

theorem t2 : p → r :=
assume H3 : p,
show r, from H1 (H2 H3)
```

Propositional Logic

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

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

They all take values in `Prop`.

```
variables p q : Prop

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

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

In the last chapter we observed that lambda abstraction can be viewed as an "introduction rule" for →. In the current setting, it shows how to "introduce" or establish an implication. Application can be viewed as an "elimination rule," showing how to "eliminate" or use an implication in a proof. The other propositional connectives are defined in the standard library in the file `init.datatypes`, and each comes with its canonical introduction and elimination rules.

Conjunction

The expression `and.intro H1 H2` creates a proof for `p ∧ q` using proofs `H1 : p` and `H2 : q`. It is common to describe `and.intro` as the *and-introduction* rule. In the next example we use `and.intro` to create a proof of `p → q → p ∧ q`.

```
example (Hp : p) (Hq : q) : p ∧ q := and.intro Hp Hq

check assume (Hp : p) (Hq : q), and.intro Hp Hq
```

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

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

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

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

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

```
example (H : p ∧ q) : q ∧ p :=
and.intro (and.right H) (and.left H)
```

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

Disjunction

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

```
example (Hp : p) : p ∨ q := or.intro_left q Hp
example (Hq : q) : p ∨ q := or.intro_right p Hq
```

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

```
example (H : p ∨ q) : q ∨ p :=
or.elim H
  (assume Hp : p,
    show q ∨ p, from or.intro_right q Hp)
  (assume Hq : q,
    show q ∨ p, from or.intro_left p Hq)
```

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

```
example (H : p ∨ q) : q ∨ p := or.elim H (λ Hp, or.inr Hp) (λ Hq, or.inl Hq)
```

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

Negation and Falsity

The expression `not.intro H` produces a proof of `¬p` from `H : p → false`. That is, we obtain `¬p` if we can derive a contradiction from `p`. The expression `not.elim Hnp Hp` produces a proof of `false` from `Hp : p` and `Hnp : ¬p`. The next example uses these rules to produce a proof of `(p → q) → ¬q → ¬p`.

```
example (Hpq : p → q) (Hnq : ¬q) : ¬p :=
not.intro
  (assume Hp : p,
    show false, from not.elim Hnq (Hpq Hp))
```

In the standard library, `¬p` is actually an *abbreviation* for `p → false`, that is, the fact that `p` implies a contradiction. You can check that `not.intro` then amounts to the introduction rule for implication. Similarly, the rule `not.elim`, that is, the principle `¬p → p → false`, corresponds to function application. In other words, `¬p → p → false` is derived by applying the first argument to the second, with the term `assume Hnp, assume Hp, Hnp Hp`. We can thus avoid the use of `not.intro` and `not.elim` entirely, in favor of abstraction and elimination:

```
example (Hpq : p → q) (Hnq : ¬q) : ¬p :=
assume Hp : p, Hnq (Hpq Hp)
```

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

```
example (Hp : p) (Hnp : ¬p) : q := false.elim (Hnp Hp)
```

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

```
example (Hp : p) (Hnp : ¬p) : q := absurd Hp Hnp
```

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

```
example (Hnp : ¬p) (Hq : q) (Hqp : q → p) : r :=
absurd (Hqp Hq) Hnp
```

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

Logical Equivalence

The expression `iff.intro H1 H2` produces a proof of `p ↔ q` from `H1 : p → q` and `H2 : q → p`. The expression `iff.elim_left H` produces a proof of `p → q` from `H : p ↔ q`. Similarly, `iff.elim_right H` produces a proof of `q → p` from `H : p ↔ q`. Here is a proof of `p ∧ q ↔ q ∧ p`:

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

```
  (assume H : q ∧ p,
    show p ∧ q, from and.intro (and.right H) (and.left H))

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

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

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

Introducing Auxiliary Subgoals

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

```
section
  variables p q : Prop

  example (H : p ∧ q) : q ∧ p :=
  have Hp : p, from and.left H,
  have Hq : q, from and.right H,
  show q ∧ p, from and.intro Hq Hp
end
```

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

Classical Logic

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

```
open classical

variable p : Prop
check em p
```

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

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

```
theorem dne {p : Prop} (H : ¬¬p) : p :=
or.elim (em p)
  (assume Hp : p, Hp)
  (assume Hnp : ¬p, absurd Hnp H)
```

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

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

```
example (H : ¬¬p) : p :=
by_cases
  (assume H1 : p, H1)
  (assume H1 : ¬p, absurd H1 H)
```

Or you can carry out a proof by contradiction:

```
example (H : ¬¬p) : p :=
by_contradiction
  (assume H1 : ¬p,
    show false, from H H1)
```

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

```
example (H : ¬ (p ∧ q)) : ¬ p ∨ ¬ q :=
or.elim (em p)
  (assume Hp : p,
    or.inr
      (show ¬q, from
        assume Hq : q,
        H (and.intro Hp Hq)))
  (assume Hp : ¬p,
    or.inl Hp)
```

We will see later that there *are* situations in constructive logic where principles like excluded middle and double-negation elimination are permissible, and Lean supports the use of classical reasoning in such contexts.

There are additional classical axioms that are not included by default in the standard library. We will discuss these in detail in Chapter Axioms and Computation.

Examples of Propositional Validities

Lean's standard library contains proofs of many valid statements of propositional logic, all of which you are free to use in proofs of your own. In this section, we will review some common identities, and encourage you to try proving them on your own using the rules above.

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

```
open classical

variables p q r s : Prop

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

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

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

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

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

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

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

Here is another useful trick. Instead of using `sorry`, you can use an underscore `_` as a placeholder. Recall that this tells Lean that the argument is implicit, and should be filled in automatically. If Lean tries to do so and fails, it returns with an error message "don't know how to synthesize placeholder." This is followed by the type of the term it is expecting, and all the objects and hypothesis available in the context. In other words, for each unresolved placeholder, Lean reports the subgoal that needs to be filled at that point. You can then construct a proof by incrementally filling in these placeholders.

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

```
open classical

variables p q r : Prop

-- distributivity
example : p ∧ (q ∨ r) ↔ (p ∧ q) ∨ (p ∧ r) :=
iff.intro
  (assume H : p ∧ (q ∨ r),
   have Hp : p, from and.left H,
   or.elim (and.right H)
     (assume Hq : q,
       show (p ∧ q) ∨ (p ∧ r), from or.inl (and.intro Hp Hq))
     (assume Hr : r,
       show (p ∧ q) ∨ (p ∧ r), from or.inr (and.intro Hp Hr)))
  (assume H : (p ∧ q) ∨ (p ∧ r),
   or.elim H
     (assume Hpq : p ∧ q,
       have Hp : p, from and.left Hpq,
       have Hq : q, from and.right Hpq,
       show p ∧ (q ∨ r), from and.intro Hp (or.inl Hq))
     (assume Hpr : p ∧ r,
       have Hp : p, from and.left Hpr,
       have Hr : r, from and.right Hpr,
       show p ∧ (q ∨ r), from and.intro Hp (or.inr Hr)))

-- an example that requires classical reasoning
example : ¬(p ∧ ¬q) → (p → q) :=
assume H : ¬(p ∧ ¬q),
```

```
assume Hp : p,
show q, from
  or.elim (em q)
    (assume Hq : q, Hq)
    (assume Hnq : ¬q, absurd (and.intro Hp Hnq) H)
```

4

Quantifiers and Equality

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

The Universal Quantifier

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

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

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

The elimination rule states:

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

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

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

The elimination rule states:

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

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

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

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

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

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

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

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

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

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

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

variables (a b c : A)
variables (Hab : r a b) (Hbc : r b c)

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

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

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

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

variables (a b c : A)
variables (Hab : r a b) (Hbc : r b c)

check trans_r
check trans_r Hab
check trans_r Hab Hbc
```

The advantage is that we can simply write `trans_r Hab Hbc` as a proof of `r a c`. The disadvantage is that Lean does not have enough information to infer the types of the arguments in the expressions `trans_r` and `trans_r Hab`. The output of the `check` command contains expressions like `?z A r trans_r a b c Hab Hbc`. Such an expression indicates an arbitrary value, that may depend on any of the values listed (in this case, all the variables in the local context).

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

```
variables (A : Type) (r : A → A → Prop)

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

example (a b c d : A) (Hab : r a b) (Hcb : r c b) (Hcd : r c d) : r a d :=
trans_r (trans_r Hab (symm_r Hcb)) Hcd
```

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

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

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

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

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

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

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

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

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

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

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

The idea is as follows. If `j` is not `0`, then `Π x : A, B` is an element of `Type.{max i j}`. In other words, the type of dependent functions from `A` to `B` "lives" in the universe with smallest index greater-than or equal to the indices of the universes of `A` and `B`. Suppose, however, that `B` is of `Type.{0}`, that is, an element of `Prop`. In that case, `Π x : A, B` is an element of `Type.{0}` as well, no matter which type universe `A` lives in. In other words, if `B` is a proposition depending on `A`, then `∀ x : A, B` is again a proposition. This reflects the interpretation of `Prop` as the type of propositions rather than data, and it is what makes `Prop` *impredicative*. In contrast to the standard kernel, such a `Prop` is absent from Lean's kernel for homotopy type theory.

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

Equality

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

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

```
check eq.refl    -- ∀ (a : ?A), a = a
check eq.symm    -- ?a = ?b → ?b = ?a
check eq.trans   -- ?a = ?b → ?b = ?c → ?a = ?c
```

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

```
variables (A : Type) (a b c d : A)
premises (Hab : a = b) (Hcb : c = b) (Hcd : c = d)

example : a = d :=
eq.trans (eq.trans Hab (eq.symm Hcb)) Hcd
```

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

```
open eq

example : a = d := trans (trans Hab (symm Hcb)) Hcd
```

Lean even defines convenient notation for writing proofs like this:

```
variables (A : Type) (a b c d : A)
premises (Hab : a = b) (Hcb : c = b) (Hcd : c = d)

-- BEGIN
open eq.ops

example : a = d := Hab · Hcb⁻¹ · Hcd
```

You can use `\tr` to enter the transitivity dot, and `\sy` to enter the inverse/symmetry symbol.

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

```
import data.nat data.prod
open nat prod

variables (A B : Type)

example (f : A → B) (a : A) : (λ x, f x) a = f a := eq.refl _
example (a : A) (b : A) : pr1 (a, b) = a := eq.refl _
example : 2 + 3 = (5 : ℕ) := eq.refl _
```

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

```
example (f : A → B) (a : A) : (λ x, f x) a = f a := rfl
example (a : A) (b : A) : pr1 (a, b) = a := rfl
example : 2 + 3 = (5 : ℕ) := rfl
```

Equality is much more than an equivalence relation, however. It has the important property that every assertion respects the equivalence, in the sense that we can substitute equal expressions without changing the truth value. That is, given `H1 : a = b` and `H2 : P a`, we can construct a proof for `P b` using substitution: `eq.subst H1 H2`.

```
example (A : Type) (a b : A) (P : A → Prop) (H1 : a = b) (H2 : P a) : P b :=
eq.subst H1 H2

example (A : Type) (a b : A) (P : A → Prop) (H1 : a = b) (H2 : P a) : P b :=
H1 ▸ H2
```

The triangle in the second presentation is, once again, made available by opening `eq.ops`, and you can use `\t` to enter it. The term `H1 ▸ H2` is just notation for `eq.subst H1 H2`. This notation is used extensively in the Lean standard library.

Here is an example of a calculation in the natural numbers that uses substitution combined with associativity, commutativity, and distributivity of the natural numbers. Of course, carrying out such calculations require being able to invoke such supporting theorems. You can find a number of identities involving the natural numbers in the associated library files, for example, in the module data.nat.basic. In the next chapter, we will have more to say about how to find theorems in Lean's library.

```
import data.nat
open nat eq.ops algebra

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 !left_distrib,
have H2 : (x + y) * (x + y) = x * x + y * x + (x * y + y * y),
  from (right_distrib x y x) ▶ !right_distrib ▶ H1,
!add.assoc⁻¹ ▶ H2
```

The exclamation mark infers explicit arguments to a theorem from the context. For more information, see Section More on Implicit Arguments. In the statement of the example, remember that addition implicitly associates to the left, so the last step of the proof puts the right-hand side of H2 in the required form.

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

The Calculation Environment

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

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

Each <proof>_i is a proof for <expr>_{i-1} op_i <expr>_i. The <proof>_i may also be of the form { <pr> }, where <pr> is a proof for some equality a = b. The form { <pr> } is just syntactic sugar for eq.subst <pr> (eq.refl <expr>_{i-1}) In other words, we are claiming we can obtain <expr>_i by replacing a with b in <expr>_{i-1}.

Here is an example:

```
import data.nat
open nat algebra

variables (a b c d e : nat)
variable H1 : a = b
variable H2 : b = c + 1
variable H3 : c = d
variable H4 : e = 1 + d

theorem T : a = e :=
calc
  a     = b      : H1
```

```
    ... = c + 1  : H2
    ... = d + 1  : {H3}
    ... = 1 + d  : add.comm d 1
    ... =  e     : eq.symm H4
```

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

```
import data.nat
open nat algebra

theorem T2 (a b c : nat) (H1 : a = b) (H2 : b = c + 1) : a ≠ 0 :=
calc
  a     = b      : H1
    ... = c + 1  : H2
    ... = succ c : add_one c
    ... ≠ 0       : succ_ne_zero c
```

Lean offers some nice additional features. If the justification for a line of a calculational proof is `foo`, Lean will try adding implicit arguments if `foo` alone fails to do the job. If that doesn't work, Lean will try the symmetric version, `foo⁻¹`, again adding arguments if necessary. If that doesn't work, Lean proceeds to try `{foo}` and `{foo⁻¹}`, again, adding arguments if necessary. This can simplify the presentation of a `calc` proof considerably. Consider, for example, the following proof of the identity in the last section:

```
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  : left_distrib
    ... = x * x + y * x + (x + y) * y            : right_distrib
    ... = x * x + y * x + (x * y + y * y)        : right_distrib
    ... = x * x + y * x + x * y + y * y          : add.assoc
```

As an exercise, we suggest carrying out a similar expansion of `(x - y) * (x + y)`, using in the appropriate order the theorems `left_distrib`, `mul.comm` and `add.comm` and the theorems `mul_sub_right_distrib` and `add_sub_add_left` in the file data.nat.sub. Note that this exercise is slightly more involved than the previous example, because the subtraction on natural numbers is truncated, so that `n - m` is equal to `0` when `m` is greater than or equal to `n`.

The Simplifier

[TO DO: this section needs to be written. Emphasize that the simplifier can be used in conjunction with calc.]

The Existential Quantifier

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

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

```
import data.nat
open nat

example : ∃ x : ℕ, x > 0 :=
have H : 1 > 0, from succ_pos 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
```

Note that `exists.intro` has implicit arguments: Lean has to infer the predicate `p : A → Prop` in the conclusion `∃ x, p x`. This is not a trivial affair. For example, if we have have `Hg : g 0 0 = 0` and write `exists.intro 0 Hg`, there are many possible values for the predicate `p`, corresponding to the theorems `∃ x, g x x = x`, `∃ x, g x x = 0`, `∃ x, g x 0 = x`, etc. Lean uses the context to infer which one is appropriate. This is illustrated in the following example, in which we set the option `pp.implicit` to true to ask Lean's pretty-printer to show the implicit arguments.

```
import data.nat
open nat

variable g : ℕ → ℕ → ℕ
variable Hg : g 0 0 = 0

theorem gex1 : ∃ x, g x x = x := exists.intro 0 Hg
theorem gex2 : ∃ x, g x 0 = x := exists.intro 0 Hg
theorem gex3 : ∃ x, g 0 0 = x := exists.intro 0 Hg
theorem gex4 : ∃ x, g x x = 0 := exists.intro 0 Hg

set_option pp.implicit true  -- display implicit arguments
check gex1
check gex2
check gex3
check gex4
```

We can view `exists.intro` as an information-hiding operation: we are "hiding" the witness to the body of the assertion. The existential elimination rule, `exists.elim`, performs the opposite operation. It allows us to prove a proposition `q` from `∃ x : A, p x`, by showing that `q` follows from `p w` for an arbitrary value `w`. Roughly speaking, since we know there is an `x` satisfying `p x`, we can give it a name, say, `w`. If `q` does not mention `w`, then showing that `q` follows from `p w` is tantamount to showing the `q` follows from the existence of any such `x`. It may be helpful to compare the exists-elimination rule to the or-elimination rule: the assertion `∃ x : A, p x` can be thought of as a big disjunction of the propositions `p a`, as `a` ranges over all the elements of `A`.

Notice that exists introduction and elimination are very similar to the sigma introduction `sigma.mk` and elimination. The difference is that given `a : A` and `h : p a`, `exists.intro a h` has type `(∃ x : A, p x) : Prop` and `sigma.mk a h` has type `(Σ x : A, p x) : Type`. The similarity between ∃ and Σ is another instance of the Curry-Howard isomorphism.

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
open nat algebra

definition 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 (fun (w1 : nat) (Hw1 : a = 2*w1),
exists.elim H2 (fun (w2 : nat) (Hw2 : b = 2*w2),
  exists.intro (w1 + w2)
    (calc
      a + b = 2*w1 + b        : Hw1
        ... = 2*w1 + 2*w2     : Hw2
        ... = 2*(w1 + w2)     : left_distrib)))
```

Lean provides syntactic sugar for `exists.elim`. The expression

```
obtain <var1> <var2>, from <expr1>,
<expr2>
```

translates to exists.elim <expr1> (λ <var1> <var2>, <expr2>). With this syntax, the example above can be presented in a more natural way:

```
theorem even_plus_even {a b : nat} (H1 : is_even a) (H2 : is_even b) :
  is_even (a + b) :=
obtain (w1 : nat) (Hw1 : a = 2*w1), from H1,
obtain (w2 : nat) (Hw2 : b = 2*w2), from H2,
exists.intro (w1 + w2)
  (calc
    a + b = 2*w1 + b        : Hw1
```

```
    ... = 2*w1 + 2*w2    : Hw2
    ... = 2*(w1 + w2)    : left_distrib)
```

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

```
open classical

variables (A : Type) (p : A → Prop)

example (H : ¬ ∀ x, ¬ p x) : ∃ x, p x :=
by_contradiction
  (assume H1 : ¬ ∃ x, p x,
    have H2 : ∀ x, ¬ p x, from
      take x,
      assume H3 : p x,
      have H4 : ∃ x, p x, from exists.intro x H3,
      show false, from H1 H4,
    show false, from H H2)
```

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

```
open classical

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

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

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

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

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

Here are solutions to two of the more difficult ones:

```
example : (∃ x, p x ∨ q x) ↔ (∃ x, p x) ∨ (∃ x, q x) :=
iff.intro
  (assume H : ∃ x, p x ∨ q x,
    obtain a (H1 : p a ∨ q a), from H,
    or.elim H1
      (assume Hpa : p a, or.inl (exists.intro a Hpa))
      (assume Hqa : q a, or.inr (exists.intro a Hqa)))
  (assume H : (∃ x, p x) ∨ (∃ x, q x),
    or.elim H
      (assume Hp : ∃ x, p x,
        obtain a Hpa, from Hp,
        exists.intro a (or.inl Hpa))
      (assume Hq : ∃ x, q x,
        obtain a Hqa, from Hq,
        exists.intro a (or.inr Hqa)))

example : (∃ x, p x → r) ↔ (∀ x, p x) → r :=
iff.intro
  (assume H1 : ∃ x, p x → r,
    assume H2 : ∀ x, p x,
    obtain b (Hb : p b → r), from H1,
    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, exists.intro a (λ H', H1 Hap))
        (assume Hnap : ¬ ∀ x, p x,
          by_contradiction
            (assume Hnex : ¬ ∃ x, p x → r,
              have Hap : ∀ x, p x, from
                take x,
                by_contradiction
                  (assume Hnp : ¬ p x,
                    have Hex : ∃ x, p x → r,
                      from exists.intro x (assume Hp, absurd Hp Hnp),
                    show false, from Hnex Hex),
              show false, from Hnap Hap)))
```

More on the Proof Language

We have seen that keywords like `assume`, `take`, `have`, `show`, and `obtain` 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`:

```
import data.nat
open nat algebra
```

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

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

One can also refer to any element or hypothesis in the context, anonymous or not, by enclosing the type in backticks:

```
example : f 0 ≤ f 3 :=
have f 0 ≤ f 1, from H 0,
have f 0 ≤ f 2, from le.trans `f 0 ≤ f 1` (H 1),
show f 0 ≤ f 3, from le.trans `f 0 ≤ f 2` (H 2)
```

In the last line, for example, the expression `` `f 0 ≤ f 2` `` means "find any element of the context that has type `f 0 ≤ f 2`." In other words, we state the assertion rather than name the variable that witnesses its truth. This can be done anywhere later in the proof:

```
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`)
```

The `suppose` keyword acts as an anonymous assume:

```
example : f 0 ≥ f 1 → f 0 = f 1 :=
suppose f 0 ≥ f 1,
show f 0 = f 1, from le.antisymm (H 0) this
```

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

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

```
example : f 0 ≥ f 1 → f 1 ≥ f 2 → f 0 = f 2 :=
suppose f 0 ≥ f 1,
suppose f 1 ≥ f 2,
have f 0 ≥ f 2, from 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`.

One can also do an anonymous `assume` by enclosing the statement in backticks.

```
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`
```

This is slightly weaker than using `suppose`, because we can no longer use the identifier `this`. But the mechanism is more general: it can be used with other binders, like `take` and `obtains`.

If more than one element of the context has the named type, the expression is ambiguous:

```
definition imp_self (p : Prop) : p → p :=
assume `p`, `p`

print imp_self

definition imp_self2 (p : Prop) : p → p → p :=
assume `p` `p`, `p`

print imp_self2
```

The output shows that in the second example, it is the second argument that is chosen. Using anonymous binders when data is involved looks somewhat odd:

```
definition idnat : ℕ → ℕ :=
take `ℕ`, `ℕ`

print idnat

definition idnat2 : ℕ → ℕ → ℕ :=
take `ℕ` `ℕ`, `ℕ`

print idnat2
eval idnat2 0 1  -- returns 1
```

But with propositions it is usually quite natural. Here is an example of an anonymous binder used with the `obtain` construction, continuing the examples above.

```
variable f : ℕ → ℕ

example (H : ∀ x : ℕ, f x ≤ f (x + 1)) (H' : ∃ x, f (x + 1) ≤ f x) :
  ∃ x, f (x + 1) = f x :=
obtain x `f (x + 1) ≤ f x`, from H',
exists.intro x
  (show f (x + 1) = f x, from le.antisymm `f (x + 1) ≤ f x` (H x))
```

The following proof that the square root of two is irrational can be found in the standard library. It provides a nice example of the way that proof terms can be structured and made readable using the devices we have discussed here.

```
import data.nat
open nat

theorem sqrt_two_irrational {a b : ℕ} (co : coprime a b) : a^2 ≠ 2 * b^2 :=
assume H : a^2 = 2 * b^2,
have even (a^2),
  from even_of_exists (exists.intro _ H),
have even a,
  from even_of_even_pow this,
obtain (c : ℕ) (aeq : a = 2 * c),
  from exists_of_even this,
have 2 * (2 * c^2) = 2 * b^2,
  by rewrite [-H, aeq, *pow_two, mul.assoc, mul.left_comm c],
have 2 * c^2 = b^2,
  from eq_of_mul_eq_mul_left dec_trivial this,
have even (b^2),
  from even_of_exists (exists.intro _ (eq.symm this)),
have even b,
  from even_of_even_pow this,
have 2 | gcd a b,
  from dvd_gcd (dvd_of_even `even a`) (dvd_of_even `even b`),
have 2 | (1 : ℕ),
  by rewrite [gcd_eq_one_of_coprime co at this]; exact this,
show false, from absurd `2 | 1` dec_trivial
```

5

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.

Displaying Information

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

```
import data.nat

-- 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

-- examples with addition
open nat
check add
check @add
eval add 3 2
print definition add

-- a user-defined function
definition foo {A : Type} (x : A) : A := x

check foo
check @foo
eval foo
eval (foo @nat.zero)
print foo
```

There are other useful `print` commands:

```
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 or emacs mode
print prefix <namespace>    : display all declarations in the namespace
print coercions             : display all coercions
print coercions <source>    : display only the coercions from <source>
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 classes, instances, and structures in Chapter Type Classes. Here are examples of how the `print` commands are used:

```
import standard algebra.ring
open prod sum int nat algebra

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

Another useful command, although the implementation is still rudimentary at this stage, is the `find decl` command. This can be used to find theorems whose conclusion matches a given pattern. The syntax is as follows:

```
find_decl <pattern>  [, filter]*
```

where `<pattern>` is an expression with "holes" (underscores), and a filter is of the form

```
+ id  (id is a substring of the declaration)
- id  (id is not a substring of the declaration)
  id  (id is a substring of the declaration)
```

For example:

```
import data.nat
open nat

find_decl ((_ * _) = (_ * _))
find_decl (_ * _) = _, +assoc
find_decl (_ * _) = _, -assoc

find_decl _ < succ _, +imp, -le
```

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
```

In Lean, *coercions* can be inserted automatically to cast an element of one data type to another, for example, to cast an element of `nat` to an element of `int`. We will say more about them later in this chapter. This list is not exhaustive; you can see a complete list by typing `set_option pp.` and then using tab-completion in the Emacs mode for Lean, also discussed below.

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

```
import data.nat
open nat

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

check 2 + 2 = 4
eval (λ x, x + 2) = (λ x, x + 3)

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

Pretty printing additional information is often very useful when you are debugging a proof, or trying to understand a cryptic error message. Too much information can be overwhelming, though, and Lean's defaults are generally sufficient for ordinary interactions.

Using the Library

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

The command `import standard` imports the essential parts of the standard library, and by now you have seen many of the namespaces you will need. For example, you should `open nat` for notation when you are working with the natural numbers, and `open int` when you are working with the integers. In general, however, 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 Emacs mode can also help you find things you need, but studying the contents of the library directly is often unavoidable.

Lean has two libraries. Here we will focus on the standard library, which offers a conventional mathematical framework. We will discuss the library for homotopy type theory in a later chapter.

There are a number of ways to explore the contents of the standard library. You can find the file structure 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.

Alternatively, there are "markdown" files in the library that provide links to the same files but list them in a more natural order, and provide additional information and annotations.

https://github.com/leanprover/lean/blob/master/library/library.md

You can again browse these through the github interface, or with a markdown reader on your computer.

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 Lean's Emacs mode, which is discussed in the next section. To start with, common "axiomatic" properties of an operation like conjunction or multiplication are put in a namespace that begins with the name of the operation:

```
import standard algebra.ordered_ring
open nat algebra

check and.comm
check mul.comm
check and.assoc
check mul.assoc
check @mul.left_cancel -- multiplication is left cancelative
```

In particular, this includes `intro` and `elim` operations for logical connectives, and properties of relations:

```
check and.intro
check and.elim
check or.intro_left
check or.intro_right
check or.elim

check eq.refl
check eq.symm
check eq.trans
```

For the most part, however, we rely on descriptive names. Often the name of theorem simply describes the conclusion:

```
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:

```
check lt_of_succ_le
check @lt_of_not_ge
check @lt_of_le_of_ne
check @add_lt_add_of_lt_of_le
```

Sometimes abbreviations or alternative descriptions are easier to work with. For example, we use `pos`, `neg`, `nonpos`, `nonneg` rather than `zero_lt`, `lt_zero`, `le_zero`, and `zero_le`.

```
check @mul_pos
check @mul_nonpos_of_nonneg_of_nonpos
check @add_lt_of_lt_of_nonpos
check @add_lt_of_nonpos_of_lt
```

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

```
check @add_le_add_left
check @add_le_add_right
check @le_of_mul_le_mul_left
check @le_of_mul_le_mul_right
```

Lean's Emacs Mode

This tutorial is designed to be read alongside Lean's web-browser interface, which runs a Javascript-compiled version of Lean inside your web browser. But there is a much more powerful interface to Lean that runs as a special mode in the Emacs text editor. Our goal in this section is to consider some of the advantages and features of the Emacs interface.

If you have never used the Emacs text editor before, you should spend some time experimenting with it. Emacs is an extremely powerful text editor, but it can also be overwhelming. There are a number of introductory tutorials on the web. See, for example:

- A Guided Tour of Emacs
- Absolute Beginner's Guide to Emacs

- Introduction to Emacs Course (PDF)

You can get pretty far simply using the menus at the top of the screen for basic editing and file management. Those menus list keyboard-equivalents for the commands. Notation like "C-x", short for "control x," means "hold down the control key while typing x." The notation "M-x", short for "Meta x," means "hold down the Alt key while typing x," or, equivalently, "press the Esc key, followed by x." For example, the "File" menu lists "C-c C-s" as a keyboard-equivalent for the "save file" command.

There are a number of benefits to using the native version of Lean instead of the web interface. Perhaps the most important is file management. The web interface imports the entire standard library internally, which is why some examples in this tutorial have to put examples in a namespace, "hide," to avoid conflicting with objects already defined in the standard library. Moreover, the web interface only operates on one file at a time. Using the Emacs editor, you can create and edit Lean theory files anywhere on your file system, as with any editor or word processor. From these files, you can import pieces of the library at will, as well as your own theories, defined in separate files.

To use the Emacs with Lean, you simply need to create a file with the extension ".lean" and edit it. (For files that should be checked in the homotopy type theory framework, use ".hlean" instead.) For example, you can create a file by typing `emacs my_file.lean` in a terminal window, in the directory where you want to keep the file. Assuming everything has been installed correctly, Emacs will start up in Lean mode, already checking your file in the background.

You can then start typing, or copy any of the examples in this tutorial. (In the latter case, make sure you include the `import` and `open` commands that are sometimes hidden in the text.) Lean mode offers syntax highlighting, so commands, identifiers, and so on are helpfully color-coded. Any errors that Lean detects are subtly underlined in red, and the editor displays an exclamation mark in the left margin. As you continue to type and eliminate errors, these annotations magically disappear.

If you put the cursor on a highlighted error, Emacs displays the error message in at the bottom of the frame. Alternatively, if you type `C-c ! l` while in Lean mode, Emacs opens a new window with a list of compilation errors. Lean relies on an Emacs mode, *Flycheck*, for this functionality, as evidenced by the letters "FlyC" that appear in the Emacs information line. An asterisk next to these letters indicates that Flycheck is actively checking the file, using Lean. Flycheck offers a number of commands that begin with `C-c !`. For example, `C-c ! n` moves the cursor to the next error, and `C-c ! p` moves the cursor to the previous error. You can get to a help menu that lists these key bindings by clicking on the "FlyC" tag.

It may be disconcerting to see a perfectly good proof suddenly "break" when you change a single character. Moreover, changes can introduce errors downstream. But the error messages vanish quickly when correctness is restored. Lean is quite fast and caches previous work to speed up compilation, and changes you make are registered almost instantaneously.

The Emacs Lean mode also maintains a continuous dialog with a background Lean process and uses it to present useful information to you. For example, if you put your cursor on any identifier — a theorem name, a defined symbol, or a variable — Emacs displays its type in the information line at the bottom. If you put the cursor on the opening parenthesis of an expression, Emacs displays the type of the expression.

This works even for implicit arguments. If you put your cursor on an underscore symbol, then, assuming Lean's elaborator was successful in inferring the value, Emacs shows you that value and its type. Typing "C-c C-f" replaces the underscore with the inferred value. In cases where Lean is unable to infer a value of an implicit argument, the underscore is highlighted, and the error message indicates the type of the "hole" that needs to be filled. This can be extremely useful when constructing proofs incrementally. One can start typing a "proof sketch," using either `sorry` or an underscore for details you intend to fill in later. Assuming the proof is correct modulo these missing pieces of information, the error message at an unfilled underscore tells you the type of the term you need to construct, typically an assertion you need to justify.

The Lean mode supports tab completion. In a context where Lean expects an identifier (e.g. a theorem name or a defined symbol), if you start typing and then hit the tab key, a popup window suggests possible matches or near-matches for the expression you have typed. This helps you find the theorems you need without having to browse the library. You can also press tab after an `import` command, to see a list of possible imports, or after the `set_option` command, to see a list of options.

If you put your cursor on an identifier and type "C-c C-p", Lean prints the definition of that identifier in a separate buffer. If you put your cursor on an identifier that is defined in Lean's library and hit "M-.", Emacs will take you to the identifier's definition in the library file itself. This works even in an autocompletion popup window: if you start typing an identifier, press the tab key, choose a completion from the list of options, and press "M-.", you are taken to the symbol's definition. When you are done, pressing "M-*" takes you back to your original position.

There are other useful tricks. If you see some notation in a Lean file and you want to know how to enter it from the keyboard, put the cursor on the symbol and type "C-c C-k". You can set common Lean options with "C-c C-o", and you can execute a Lean command using "C-c C-e". These commands and others are summarized in the online documentation:

https://github.com/leanprover/lean/blob/master/src/emacs/README.md

If for some reason the Lean background process does not seem to be responding (for example, the information line no longer shows you type information), type "C-c C-r", or "M-x lean-server-restart-process", or choose "restart lean process" from the Lean menu, and with luck that will set things right again.

This is a good place to mention another trick that is sometimes useful when editing long files. In Lean, the `exit` command halts processing of the file abruptly. If you are making changes at the top of a long file and want to defer checking of the remainder of the file until you are done making those changes, you can temporarily insert an `exit`.

Projects

At this point, it will be helpful to convey more information about the inner workings of Lean. A .lean file (or .hlean file, if you are working on homotopy type theory) consists of instructions that tell Lean how to construct formal terms in dependent type theory. "Processing" this file is a matter of filling in missing or implicit information, constructing the relevant terms, and sending them to the type checker to confirm that they are well-formed and have the specified types. This is analogous to the compilation process for a programming language: the .lean or .hlean file contains the source code that is then compiled down to machine representations of the desired formal objects. Lean stores the output of the compilation process in files with the extension ".olean", for "object Lean".

It is these files that are loaded by the `import` command. When Lean processes an `import` command, it looks for the relevant .olean files in standard places. By default, the search path consists of the root of the standard library (or the hott library, if the file is a .hlean file) and the current directory. You can specify subdirectories using periods in the module name: for example, `import foo.bar.baz` looks for the file "foo/bar/baz.olean" relative to any of the locations listed in the search path. A leading period, as in `import .foo.bar`, indicates that the .olean file in question is specified relative to the current directory. Two leading periods, as in `import ..foo.bar`, indicates that the address is relative to the parent directory, and so on.

If you enter the command `lean -o foo.olean foo.lean` from the command line, Lean processes `foo.lean` and, if it compiles successfully, it stores the output in `foo.olean`. The result is that another file can then `import foo`.

When you are editing a single file with either the web interface or the Emacs Lean mode, however, Lean only checks the file internally, without saving the .olean output. Suppose, then, you wish to build a project that has multiple files. What you really want is that Lean's Emacs mode will build all the relevant .olean files in the background, so that you can import those files freely.

The Emacs mode makes this easy. To start a project that may potentially involve more than one file, choose the folder where you want the project to reside, open an initial file in Emacs, choose "create a new project" from the Lean menu, and press the "open" button. This creates a file, `.project`, which instructs a background process to ensure that whenever you are working on a file in that folder (or any subfolder thereof), compiled versions of all the modules it depends on are available and up to date.

Suppose you are editing `foo.lean`, which imports `bar`. You can switch to `bar.lean` and make additions or corrections to that file, then switch back to `foo` and continue working.

The process `linja`, based on the `ninja` build system, ensures that `bar` is recompiled and that an up-to-date version is available to `foo`.

Incidentally, outside of Emacs, from a terminal window, you can type `linja` anywhere in your project folder to ensure that all your files have compiled `.olean` counterparts, and that they are up to date.

Notation and Abbreviations

Lean's parser is an instance of a Pratt parser, a non-backtracking parser that is fast and flexible. You can read about Pratt parsers in a number of places online, such as here:

> http://en.wikipedia.org/wiki/Pratt_parser http://eli.thegreenplace.net/2010/01/02/top-down-operator-precedence-parsing

Identifiers 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 moreover extensible, which is to say, we can define new notation.

```
import data.nat
open nat

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

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

infix `<*>`:50 := mul_square

eval [2 ** 3]
eval 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
```

The file `reserved_notation.lean` in the `init` folder of the library declares the left-binding powers of a number of common symbols that are used in the library.

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

You are welcome to overload these symbols for your own use, but you cannot change their right-binding power.

Remember that 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:

```
import data.nat
open nat

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.

Notation in Lean can be *overloaded*, which is to say, the same notation can be used for more than one purpose. In that case, Lean's elaborator will try to disambiguate based on context. For example, we have already seen that with the `eq.ops` namespace open, the inverse symbol can be used to denote the symmetric form of an equality. It can also be used to denote the multiplicative inverse:

```
import data.rat
open rat eq.ops

variable r : ℚ

check r⁻¹              -- ℚ
check (eq.refl r)⁻¹    -- r = r
```

Insofar as overloads produce ambiguity, they should be used sparingly. We avoid the use of overloads for arithmetic operations like `+`, `*`, `-`, and `/` by using *type classes*, as described in Chapter Type Classes. In the following, the addition operation denotes a general algebraic operation that can be instantiated to `nat` or `int` as required:

```
import data.nat data.int
open algebra nat int

variables a b : int
```

```
variables m n : nat

check a + b    -- ℤ
check m + n    -- ℕ
print notation +
```

This is sometimes called *parametric polymorphism*, in contrast to *ad hoc polymorphism*, which we are considering here. For example, the notation `++` is used to concatenate both lists and vectors:

```
import data.list data.tuple
open list tuple

variables (A : Type) (m n : ℕ)
variables (v : tuple A m) (w : tuple A n) (s t : list A)

check s ++ t
check v ++ w
```

Where it is necessary to disambiguate, Lean allows you to precede an expression with the notation `#<namespace>` to specify the namespace in which notation is to be interpreted.

```
import data.list data.tuple
open list tuple

variables (A : Type) (m n : ℕ)
variables (v : tuple A m) (w : tuple A n) (s t : list A)

-- BEGIN
check (#list λ x y, x ++ y)
check (#tuple λ x y, x ++ y)
-- END
```

Lean provides an `abbreviation` mechanism that is similar to the notation mechanism.

```
import data.nat
open nat

abbreviation double (x : ℕ) : ℕ := x + x

theorem foo (x : ℕ) : double x = x + x := rfl
check foo
```

An abbreviation is a transient form of definition that is expanded as soon as an expression is processed. As with notation, however, the pretty-printer re-constitutes the expression and prints the type of `foo` as `double x = x + x`. As with notation, you can designate an abbreviation to be `[parsing-only]`, and you can direct the pretty-printer to suppress their use with the command `set_option pp.notation false`. Finally, again as with notation, you can limit the scope of an abbreviation by prefixing the declarations with the `local` modifier.

As the name suggests, abbreviations are intended to be used as convenient shorthand for long expressions. One common use is to abbreviate a long identifier:

```
definition my_long_identity_function {A : Type} (x : A) : A := x
local abbreviation my_id := @my_long_identity_function
```

Coercions

Lean also provides mechanisms to automatically insert *coercions* between types. These are user-defined functions between datatypes that make it possible to "view" one datatype as another. For example, any natural number can be coerced to an integer.

```
check m + n          -- m + n : ℕ
check a + n          -- a + n : ℤ
check n + a          -- n + a : ℤ
check (m + n : ℤ)    -- m + n : ℤ

set_option pp.coercions true

check m + n          -- m + n : ℕ
check a + n          -- a + of_nat n : ℤ
check n + a          -- of_nat n + a : ℤ
check (m + n : ℤ)    -- of_nat (m + n) : ℤ
```

Setting the option `pp.coercions` to `true` makes the coercions explicit. Coercions that are declared in a namespace are only available to the system when the namespace is opened. The notation `(t : T)` constrains Lean to find an interpertation of `t` which gives it a type that is definitionally equal to `T`, thereby allowing you to specify the interpretation of `t` you have in mind. Thus checking `(m + n : ℤ)` forces the insertion of a coercion.

Here is an example of how we can define a coercion from the booleans to the natural numbers.

```
import data.bool data.nat
open bool nat

definition bool.to_nat [coercion] (b : bool) : nat :=
bool.cond b 1 0

eval 2 + ff
eval 2 + tt
eval tt + tt + tt + ff

print coercions        -- show all coercions
print coercions bool   -- show all coercions from bool
```

The tag "coercion" is an *attribute* that is associated with the symbol `bool.to_nat`. It does not change the meaning of `bool.to_nat`. Rather, it associates additional information to the symbol that informs Lean's elaboration algorithm, as discussed in Section Elaboration and Unification. We could also declare `bool.to_nat` to be a coercion after the fact as follows:

```
definition bool.to_nat (b : bool) : nat :=
bool.cond b 1 0

attribute bool.to_nat [coercion]
```

In both cases, the scope of the coercion is the current namespace, so the coercion will be in place whenever the module is imported and the namespace is open. Sometimes it is useful to assign an attribute only temporarily. The `local` modifier ensures that the declaration is only in effect in the current file, and within the current namespace or section:

```
definition bool.to_nat (b : bool) : nat :=
bool.cond b 1 0

local attribute bool.to_nat [coercion]
```

Overloads and coercions introduce "choice points" in the elaboration process, forcing the elaborator to consider multiple options and backtrack appropriately. This can slow down the elaboration process. What is more problematic is that it can make error messages less informative: Lean only reports the result of the last backtracking path, which means the failure that is reported to the user may be due to the wrong interpretation of an overload or coercion. This is why Lean provides mechanism for namespace management: parsing and elaboration go more smoothly when we only import the notation that we need.

Nonetheless, overloading is quite convenient, and often causes no problems. There are various ways to manually disambiguate an expression when necessary. One is to precede the expression with the notation `#<namespace>`, to specify the namespace in which notation is to be interpreted. Another is to replace the notation with an explicit function name. Yet a third is to use the `(t : T)` notation to indicate the intended type.

6

Inductive Types

We have seen that Lean's formal foundation includes basic types, `Prop, Type.{1}, Type.{2}, ...`, and allows for the formation of dependent function types, `Π x : A. B`. In the examples, we have also made use of additional types like `bool`, `nat`, and `int`, and type constructors, like `list`, and product, ×. In fact, in Lean's library, every concrete type other than the universes and every type constructor other than Pi is an instance of a general family of type constructions known as *inductive types*. It is remarkable that it is possible to construct a substantial edifice of mathematics based on nothing more than the type universes, Pi types, and inductive types; everything else follows from those.

Intuitively, an inductive type is built up from a specified list of constructors. In Lean, the syntax for specifying such a type is as follows:

```
inductive foo : Type :=
| 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 [2].

We will provide a number of examples of inductive types. We will also consider slight generalizations of the scheme above, to mutually defined inductive types, and so-called *inductive families*.

As with the logical connectives, every inductive type comes with introduction rules, which show how to construct an element of the type, and elimination rules, which show how to "use" an element of the type in another construction. The analogy to the logical connectives should not come as a surprise; as we will see below, they, too, are examples of inductive type constructions. You have already seen the introduction rules for an inductive type: they are just the constructors that are specified in the definition of the type. The elimination rules provide for a principle of recursion on the type, which includes, as a special case, a principle of induction as well.

In the next chapter, we will describe Lean's function definition package, which provides even more convenient ways to define functions on inductive types and carry out inductive proofs. But because the notion of an inductive type is so fundamental, we feel it is important to start with a low-level, hands-on understanding. We will start with some basic examples of inductive types, and work our way up to more elaborate and complex examples.

Enumerated Types

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

```
inductive weekday : Type :=
| sunday : weekday
| monday : weekday
| tuesday : weekday
| wednesday : weekday
| thursday : weekday
| friday : weekday
| saturday : weekday
```

The `inductive` command creates a new type, `weekday`. The constructors all live in the `weekday` namespace.

```
check weekday.sunday
check weekday.monday

open weekday

check sunday
check monday
```

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

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

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

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

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

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

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

It is often useful to group definitions and theorems related to a structure in a namespace with the same name. For example, we can put the `number_of_day` function in the `weekday` namespace. We are then allowed to use the shorter name when we open the namespace.

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

```
namespace weekday
  local abbreviation cases_on := @weekday.cases_on

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

eval weekday.number_of_day weekday.sunday
```

```
open weekday (renaming cases_on → cases_on)

eval number_of_day sunday
check cases_on
```

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

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

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

  eval next (next tuesday)
  eval next (previous tuesday)

  example : next (previous tuesday) = tuesday := rfl
end weekday
```

How can we prove the general theorem that `next (previous d) = d` for any weekday `d`? The induction principle parallels the recursion principle: we simply have to provide a proof of the claim for each constructor:

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

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

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

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

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

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

```
inductive empty : Type

inductive unit : Type :=
star : unit

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

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

The type `empty` is an inductive datatype with no constructors. The type `unit` has a single element, `star`, and the type `bool` represents the familiar boolean values. As an exercise, you should think about what the introduction and elimination rules for these types do. As a further exercise, we suggest defining boolean operations `band`, `bor`, `bnot` on the boolean, and verifying common identities. Note that defining a binary operation like `band` will require nested cases splits:

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

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

Constructors with Arguments

Enumerated types are a very special case of inductive types, in which the constructors take no arguments at all. In general, a "construction" can depend on data, which is then represented in the constructed argument. Consider the definitions of the product type and sum type in the library:

```
inductive prod (A B : Type) :=
mk : A → B → prod A B

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

For the moment, ignore the annotation `{}` after the constructors `inl` and `inr`; we will explain that below. In the meanwhile, think about what is going on in these examples. The product type has one constructor, `prod.mk`, which takes two arguments. To define a function on `prod A B`, we can assume the input is of the form `prod.mk a b`, and we have to specify the output, in terms of `a` and `b`. We can use this to define the two projections for prod; remember that the standard library defines notation `A × B` for `prod A B` and `(a, b)` for `prod.mk a b`.

```
definition pr1 {A B : Type} (p : A × B) : A :=
prod.rec_on p (λ a b, a)

definition pr2 {A B : Type} (p : A × B) : B :=
prod.rec_on p (λ a b, b)
```

The function `pr1` takes a pair, `p`. Applying the recursor `prod.rec_on p (fun a b, a)` interprets `p` as a pair, `prod.mk a b`, and then uses the second argument to determine what to do with `a` and `b`.

Here is another example:

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

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

The `cond` function is a boolean conditional: `cond b t1 t2` return `t1` if `b` is true, and `t2` otherwise. (It has the same effect as `bool.rec_on b t2 t1`.) The function `prod_example` takes a pair consisting of a boolean, `b`, and a number, `n`, and returns either `2 * n` or `2 * n + 1` according to whether `b` is true or false.

In contrast, the sum type has *two* constructors, `inl` and `inr` (for "insert left" and "insert right"), each of which takes *one* (explicit) argument. To define a function on `sum A B`, we have to handle two cases: either the input is of the form `inl a`, in which case we have to specify an output value in terms of `a`, or the input is of the form `inr b`, in which case we have to specify an output value in terms of `b`.

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

```
eval sum_example (inl 3)
eval sum_example (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`.

In the section after next we will see what happens when the constructor of an inductive type takes arguments from the inductive type itself. What characterizes the examples we consider in this section is that this is not the case: each constructor relies only on previously specified types.

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

A type, like `prod`, with only one constructor is purely conjunctive: the constructor simply packs the list of arguments into a single piece of data, essentially a tuple where the type of subsequent arguments can depend on the type of the initial argument. We can also think of such a type as a “record” or a “structure”. In Lean, these two words are synonymous, and provide alternative syntax for inductive types with a single constructor.

```
structure prod (A B : Type) :=
mk :: (pr1 : A) (pr2 : B)
```

The `structure` command simultaneously introduces the inductive type, `prod`, its constructor, `mk`, the usual eliminators (`rec`, `rec_on`), as well as the projections, `pr1` and `pr2`, as defined above.

If you do not name the constructor, Lean uses `mk` as a default. For example, the following defines a record to store a color as a triple of RGB values:

```
record color := (red : nat) (green : nat) (blue : nat)
definition yellow := color.mk 255 255 0
eval color.red yellow
```

The definition of `yellow` forms the record with the three values shown, and the projection `color.red` returns the red component. The `structure` command is especially useful for defining algebraic structures, and Lean provides substantial infrastructure to support working with them. Here, for example, is the definition of a semigroup:

```
structure Semigroup : Type :=
(carrier : Type)
(mul : carrier → carrier → carrier)
(mul_assoc : ∀ a b c, mul (mul a b) c = mul a (mul b c))
```

We will see more examples in Chapter Structures and Records.

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

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

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

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

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

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

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

inductive inhabited (A : Type) : Type :=
mk : A → inhabited A
```

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

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

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

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 a b

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

You should think about how these give rise to the introduction and elimination rules that you have already seen. There are rules that govern what the eliminator of an inductive type can eliminate *to*, that is, what kinds of types can be the target of a recursor. Roughly speaking, what characterizes inductive types in `Prop` is that one can only eliminate to other types in `Prop`. This is consistent with the understanding that if `P : Prop`, an element `p : P` carries no data. There is a small exception to this rule, however, which we will discuss below, in the section on inductive families.

Even the existential quantifier is inductively defined:

```
inductive Exists {A : Type} (P : A → Prop) : Prop :=
intro : ∀ (a : A), P a → Exists P

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

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

The definitions of `false`, `true`, `and`, and `or` are perfectly analogous to the definitions of `empty`, `unit`, `prod`, and `sum`. The difference is that the first group yields elements of `Prop`, and the second yields elements of `Type.{i}` for `i` greater than 0. In a similar way, `∃ x : A, P` is a `Prop`-valued variant of `Σ x : A, P`.

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

```
inductive subtype {A : Type} (P : A → Prop) : Type :=
tag : Π x : A, P x → subtype P
```

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

Defining the Natural Numbers

The inductively defined types we have seen so far are "flat": constructors wrap data and insert it into a type, and the corresponding recursor unpacks the data and acts on it. Things get much more interesting when the constructors act on elements of the very type being defined. A canonical example is the type `nat` of natural numbers:

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

There are two constructors. We start with `zero : nat`; it takes no arguments, so we have it from the start. In contrast, the constructor `succ` can only be applied to a previously constructed `nat`. Applying it to `zero` yields `succ zero : nat`. Applying it again yields `succ (succ zero) : nat`, and so on. Intuitively, `nat` is the "smallest" type with these constructors, meaning that it is exhaustively (and freely) generated by starting with `zero` and applying `succ` repeatedly.

As before, the recursor for `nat` is designed to define a dependent function `f` from `nat` to any domain, that is, an element `f` of `Π n : nat, C n` for some `C : nat → Type`. It has to handle two cases: the case where the input is `zero`, and the case where the input is of the form `succ n` for some `n : nat`. In the first case, we simply specify a target value with the appropriate type, as before. In the second case, however, the recursor can assume that a value of `f` at `n` has already been computed. As a result, the next argument to the recursor specifies a value for `f (succ n)` in terms of `n` and `f n`. If we check the type of the recursor,

```
check @nat.rec_on
```

we find the following:

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

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

Consider, for example, the addition function `add m n` on the natural numbers. Fixing `m`, we can define addition by recursion on `n`. In the base case, we set `add m zero` to `m`. In the successor step, assuming the value `add m n` is already determined, we define `add m (succ n)` to be `succ (add m n)`.

```
namespace nat

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

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

end nat
```

It is useful to put such definitions into a namespace, `nat`. We can then go on to define familiar notation in that namespace. The two defining equations for addition now hold definitionally:

```
notation 0 := zero
infix `+` := add

theorem add_zero (m : nat) : m + 0 = m := rfl
theorem add_succ (m n : nat) : m + succ n = succ (m + n) := rfl
```

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)`.

```
local abbreviation induction_on := @nat.induction_on

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

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

For another example, let us prove the associativity of addition, `∀ m n k, m + n + k = m + (n + k)`. (The notation `+`, as we have defined it, associates to the left, so `m + n + k` is really `(m + n) + k`.) The hardest part is figuring out which variable to do the induction on. Since addition is defined by recursion on the second argument, `k` is a good guess, and once we make that choice the proof almost writes itself:

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

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

```
theorem add_comm (m n : nat) : m + n = n + m :=
induction_on n
  (show m + 0 = 0 + m, from eq.symm (zero_add m))
  (take n,
    assume IH : m + n = n + m,
    calc
      m + succ n = succ (m + n) : rfl
        ... = succ (n + m) : 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) :=
induction_on n
  (show succ m + 0 = succ (m + 0), from rfl)
  (take n,
    assume IH : succ m + n = succ (m + n),
    show succ m + succ n = succ (m + succ n), from
      calc
        succ m + succ n = succ (succ m + n) : rfl
          ... = succ (succ (m + n)) : IH
          ... = succ (m + succ n) : rfl)
```

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

As an exercise, try defining other operations on the natural numbers, such as multiplication, the predecessor function (with `pred 0 = 0`), truncated subtraction (with `n - m = 0` when `m` is greater than or equal to `n`), and exponentiation. Then try proving some of their basic properties, building on the theorems we have already proved.

```
-- define mul by recursion on the second argument
definition mul (m n : nat) : nat := sorry

infix `*` := mul
```

```
-- these should be proved by rfl
theorem mul_zero (m : nat) : m * 0 = 0 := sorry
theorem mul_succ (m n : nat) : m * (succ n) = m * n + m := sorry

theorem zero_mul (n : nat) : 0 * n = 0 := sorry

theorem mul_distrib (m n k : nat) : m * (n + k) = m * n + m * k := sorry

theorem mul_assoc (m n k : nat) : m * n * k = m * (n * k) := sorry

-- hint: you will need to prove an auxiliary statement
theorem mul_comm (m n : nat) : m * n = n * m := sorry

definition pred (n : nat) : nat := nat.cases_on n zero (fun n, n)

theorem pred_succ (n : nat) : pred (succ n) = n := sorry

theorem succ_pred (n : nat) : n ≠ 0 → succ (pred n) = n := sorry
```

Other Inductive Types

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

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

namespace list

variable {A : Type}

notation h :: t  := cons h t

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

notation s ++ t := append s t

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

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

end list
```

A list of elements of type `A` is either the empty list, `nil`, or an element `h : A` followed by a list `t : list A`. We define the notation `h :: t` to represent the latter. The first element, `h`, is commonly known as the "head" of the list, and the remainder, `t`, is known as the "tail." Recall that the notation `{}` in the definition of the inductive type ensures that the argument to `nil` is implicit. In most cases, it can be inferred from context. When it cannot, we have to write `@nil A` to specify the type `A`.

Lean allows us to define iterative notation for lists:

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

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 ℕ)
end

end list
```

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

As an exercise, prove the following:

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

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

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

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

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

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

```
import data.nat
open nat

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

namespace cbtree

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

definition omega : cbtree :=
```

```
sup (nat.rec leaf (λ n t, succ t))

end cbtree
```

Generalizations

We now consider two generalizations of inductive types that are sometimes useful. 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.

```
inductive tree (A : Type) : Type :=
| node : A → forest A → tree A
with forest : Type :=
| nil  : forest A
| cons : tree A → forest A → forest A
```

In this example, a `tree` with elements labeled from `A` is of the form `node a f`, where `a` is an element of `A` (the label), and `f` a forest. At the same time, a `forest` of trees with elements labeled from `A` is essentially defined to be a list of trees.

A more powerful generalization is given by the possibility of defining inductive type `families`. There are indexed families of types defined by a simultaneous induction of the following form:

```
inductive foo : ... → Type :=
| constructor₁ : ... → foo ...
| constructor₂ : ... → foo ...
...
| constructorₙ : ... → foo ...
```

In contrast to ordinary inductive definition, which construct an element of `Type`, the more general version constructs a function `...  → Type`, where "`...`" denotes a sequence of argument types, also known as *indices*. Each constructor then constructs an element of some type in the family. One example is the definition of `vector A n`, the type of vectors of elements of `A` of length `n`:

```
inductive vector (A : Type) : nat → Type :=
| nil {} : vector A zero
| cons   : Π {n}, A → vector A n → vector A (succ n)
```

Notice that the `cons` constructor takes an element of `vector A n`, and returns an element of `vector A (succ n)`, thereby using an element of one member of the family to build an element of another.

Another example is given by the family of types `fin n`. For each `n`, `fin n` is supposed to denote a generic type of `n` elements:

```
inductive fin : nat → Type :=
| fz : Π n, fin (nat.succ n)
| fs : Π {n}, fin n → fin (nat.succ n)
```

This example may be hard to understand, so you should take the time to think about how it works.

Yet another example is given by the definition of the equality type in the library:

```
inductive eq {A : Type} (a : A) : A → Prop :=
refl : eq a a
```

For each fixed `A : Type` and `a : A`, this definition constructs a family of types `eq a x`, indexed by `x : A`. Notably, however, there is only one constructor, `refl`, which is an element of `eq a a`. Intuitively, the only way to construct a proof of `eq a x` is to use reflexivity, in the case where `x` is `a`. Note that `eq a a` is the only inhabited type in the family of types `eq a x`. The elimination principle generated by Lean says that `eq` is the *least* reflexive relation on `A`. The eliminator/recursor for `eq` is of the following form:

```
eq.rec_on : Π {A : Type} {a : A} {C : A → Type} {b : A}, 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`.

This eliminator illustrates the exception to the fact that inductive definitions living in `Prop` can only eliminate to `Prop`. Because there is only one constructor to `eq`, it carries no information, other than the type is inhabited, and Lean's internal logic allows us to eliminate to an arbitrary `Type`. This is how we define a *cast* operation that casts an element from type `A` into `B` when a proof `p : eq A B` is provided:

```
theorem cast {A B : Type} (p : eq A B) (a : A) : B :=
eq.rec_on p a
```

The recursor `eq.rec_on` is also used to define substitution:

```
theorem subst {A : Type} {a b : A} {P : A → Prop}
  (H₁ : eq a b) (H₂ : P a) : P b :=
eq.rec H₂ H₁
```

Using the recursor with H_1 `: a = b`, we may assume `a` and `b` are the same, in which case, `P b` and `P a` are the same.

It is not hard to prove that `eq` is symmetric and transitive. In the following example, we prove `symm` and leave as exercise the theorems `trans` and `congr` (congruence).

```
theorem symm {A : Type} {a b : A} (H : eq a b) : eq b a :=
subst H (eq.refl a)

theorem trans {A : Type} {a b c : A} (H₁ : eq a b) (H₂ : eq b c) : eq a c :=
sorry

theorem congr {A B : Type} {a b : A} (f : A → B) (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.

Heterogeneous Equality

Given `A : Type` and `B : A → Type`, suppose we want to generalize the congruence theorem `congr` in the previous example to dependent functions `f : Π x : A, B x`. Roughly speaking, we would like to have a theorem that, says that if `a = b`, then `f a = f b`. The first obstacle is stating the theorem: the term `eq (f a) (f b)` is not type correct since `f a` has type `B a`, `f b` has type `B b`, and the equality predicate `eq` expects both arguments to have the same type. Notice that `f a` has type `B a`, so the term `eq.rec_on H (f a)` has type `B b`. You should think of `eq.rec_on H (f a)` as "`f a`, viewed as an element of `B b`." We can then write `eq (eq.rec_on H (f a)) (f b)` to express that `f a` and `f b` are equal, modulo the difference between their types. Here is a proof of the generalized congruence theorem, with this approach:

```
theorem hcongr {A : Type} {B : A → Type} {a b : A} (f : Π x : A, B x)
               (H : eq a b) : eq (eq.rec_on H (f a)) (f b) :=
have h₁ : ∀ h : eq a a, eq (eq.rec_on h (f a)) (f a), from
  assume h : eq a a, eq.refl (eq.rec_on h (f a)),
have h₂ : ∀ h : eq a b, eq (eq.rec_on h (f a)) (f b), from
  eq.rec_on H h₁,
show eq (eq.rec_on H (f a)) (f b), from
  h₂ H
```

Another option is to define a *heterogeneous equality* `heq` that can equate terms of different types, so that we can write `heq (f a) (f b)` instead of `eq (eq.rec_on H (f a)) (f b)`. It is straightforward to define such an equality in Lean:

```
inductive heq {A : Type} (a : A) : Π {B : Type}, B → Prop :=
refl : heq a a
```

Moreover, given `a b : A`, we can prove `heq a b → eq a b` using proof irrelevance. This theorem is called `heq.to_eq` in the Lean standard library. We can now state and prove `hcongr` using heterogeneous equality. Note the proof is also more compact and easier to understand.

```
theorem hcongr {A : Type} {B : A → Type} {a b : A} (f : Π x : A, B x)
              (H : eq a b) : heq (f a) (f b) :=
eq.rec_on H (heq.refl (f a))
```

Heterogeneous equality, which gives elements of different types the illusion that they can be considered equal, is sometimes called *John Major equality*. (The name is a bit of political humor, due to Conor McBride.)

Automatically Generated Constructions

In the previous sections, we have seen that whenever we declare an inductive datatype `I`, the Lean kernel automatically declares its constructors (aka introduction rules), and generates and declares the eliminator/recursor `I.rec`. The eliminator expresses a principle of definition by recursion, as well as the principle of proof by induction. The kernel also associates a *computational rule* which determines how these definitions are eliminated when terms and proofs are normalized.

Consider, for example, the natural numbers. Given the motive `C : nat → Type`, and minor premises `fz : C zero` and `fs : Π (n : nat), C n → C (succ n)`, we have the following two computational rules: `nat.rec fz fs zero` reduces to `fz`, and `nat.rec fz fs (succ a)` reduces to `fs a (nat.rec fz fs a)`.

```
open nat

variable C  : nat → Type
variable fz : C zero
variable fs : Π (n : nat), C n → C (succ n)

eval nat.rec fz fs zero
-- nat.rec_on is defined from nat.rec
eval nat.rec_on zero fz fs

example : nat.rec fz fs zero = fz :=
rfl

variable a : nat

eval nat.rec fz fs (succ a)
eval nat.rec_on (succ a) fz fs

example (a : nat) : nat.rec fz fs (succ a) = fs a (nat.rec fz fs a) :=
rfl
```

The source code that validates an inductive declaration and generates the eliminator/recursor and computational rules is part of the Lean kernel. The kernel is also known as the *trusted code base*, because a bug in the kernel may compromise the soundness of the whole system.

When you define an inductive datatype, Lean automatically generates a number of useful definitions. We have already seen some of them: `rec_on`, `induction_on`, and `cases_on`. The module `M` that generates these definitions is *not* part of the trusted code base. A bug in `M` does not compromise the soundness of the whole system, since the kernel will catch such errors when type checking any incorrectly generated definition produced by `M`.

As described before, `rec_on` just uses its arguments in a more convenient order than `rec`. In `rec_on`, the major premise is provided before the minor premises. Constructions using `rec_on` are often easier to read and understand than the equivalent ones using `rec`.

```
open nat

print definition nat.rec_on

definition rec_on {C : nat → Type} (n : nat)
                  (fz : C zero) (fs : Π a, C a → C (succ a)) : C n :=
nat.rec fz fs n
```

Moreover, `induction_on` is just a special case of `rec_on` where the motive `C` is a proposition. Finally, `cases_on` is a special case of `rec_on` where the inductive/recursive hypotheses are omitted in the minor premises. For example, in `nat.cases_on` the minor premise `fs` has type `Π (n : nat), C (succ n)` instead of `Π (n : nat), C n → C (succ n)`. Note that the inductive/recursive hypothesis `C n` has been omitted.

```
open nat

print definition nat.induction_on
print definition nat.cases_on

definition induction_on {C : nat → Prop} (n : nat)
                        (fz : C zero) (fs : Π a, C a → C (succ a)) : C n :=
nat.rec_on n fz fs

definition cases_on {C : nat → Prop} (n : nat)
                    (fz : C zero) (fs : Π a, C (succ a)) : C n :=
nat.rec_on n fz (fun (a : nat) (r : C a), fs a)
```

For any inductive datatype that is not a proposition, we can show that its constructors are injective and disjoint. For example, on `nat`, we can show that `succ a = succ b → a = b` (injectivity), and `succ a ≠ zero` (disjointness). Both proofs can be performed using the automatically generated definition `nat.no_confusion`. More generally, for any inductive datatype `I` that is not a proposition, Lean automatically generates a definition of `I.no_confusion`. Given a motive `C` and an equality `h : c₁ t = c₂ s`, where c_1 and c_2 are two distinct `I` constructors, `I.no_confusion` constructs an inhabitant of `C`. This is essentially the *principle of explosion*, that is, the fact that anything follows from a contradiction. On the other hand, given a proof of `c t = c s` with the same constructor on both sides and a proof of `t = s → C`, `I.no_confusion` returns an inhabitant of `C`.

Let us illustrate by considering the constructions for the type `nat`. The type of `no_confusion` is based on the auxiliary definition `no_confusion_type`:

```
open nat

check @nat.no_confusion
-- Π {P : Type} {v1 v2 : ℕ}, v1 = v2 → nat.no_confusion_type P v1 v2

check nat.no_confusion_type
-- Type → ℕ → ℕ → Type
```

Note that the motive is an implicit argument in `no_confusion`. The constructions work as follows:

```
variable C : Type
variables a b : nat

eval nat.no_confusion_type C zero     (succ a)
-- C
eval nat.no_confusion_type C (succ a) zero
-- C
eval nat.no_confusion_type C zero     zero
-- C → C
eval nat.no_confusion_type C (succ a) (succ b)
-- (a = b → C) → C
```

In other words, from a proof of `zero = succ a` or `succ a = 0`, we obtain an element of any type `C` at will. On the other hand, a proof of `zero = zero` provides no help in constructing an element of type `C`, whereas a proof of `succ a = succ b` reduces the task of constructing an element of type `C` to the task of constructing such an element under the additional hypothesis `a = b`.

It is not hard to prove that constructors are injective and disjoint using `no_confusion`. In the following example, we prove these two properties for `nat` and leave as exercise the equivalent proofs for trees.

```
open nat

theorem succ_ne_zero (a : nat) (h : succ a = zero) : false :=
nat.no_confusion h

theorem succ.inj (a b : nat) (h : succ a = succ b) : a = b :=
nat.no_confusion h (fun e : a = b, e)

inductive tree (A : Type) : Type :=
| leaf : A → tree A
| node : tree A → tree A → tree A
```

```
open tree

variable {A : Type}

theorem leaf_ne_node {a : A} {l r : tree A}
                     (h : leaf a = node l r) : false :=
sorry

theorem leaf_inj {a b : A} (h : leaf a = leaf b) : a = b :=
sorry

theorem node_inj_left {l1 r1 l2 r2 : tree A}
                      (h : node l1 r1 = node l2 r2) : l1 = l2 :=
sorry

theorem node_inj_right {l1 r1 l2 r2 : tree A}
                       (h : node l1 r1 = node l2 r2) : r1 = r2 :=
sorry
```

If a constructor contains dependent arguments (such as `sigma.mk`), the generated `no_confusion` uses heterogeneous equality to equate arguments of different types:

```
variables (A : Type) (B : A → Type)
variables (a1 a2 : A) (b1 : B a1) (b2 : B a2)
variable  (C : Type)

-- Remark: b1 and b2 have different types

eval sigma.no_confusion_type C (sigma.mk a1 b1) (sigma.mk a2 b2)
-- (a1 = a2 → b1 == b2 → C) → C
```

Lean also generates the predicate transformer `below` and the recursor `brec_on`. It is unlikely that you will ever need to use these constructions directly; they are auxiliary definitions used by the recursive equation compiler we will describe in the next chapter, and we will not discuss them further here.

Universe Levels

Since an inductive type lives in `Type.{i}` for some `i`, it is reasonable to ask *which* universe levels `i` can be instantiated to. The goal of this section is to explain the relevant constraints.

In the standard library, there are two cases, depending on whether the inductive type is specified to land in `Prop`. Let us first consider the case where the inductive type is not specified to land in `Prop`, which is the only case that arises in the homotopy type theory instantiation of the kernel. Recall that each constructor `c` in the definition of a family `C` of inductive types is of the form

```
c : Π (a : A) (b : B[a]), C a p[a,b]
```

where `a` is a sequence of datatype 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. Then the universe level `i` of `C` is constrained to satisfy the following:

> For each constructor `c` as above, and each `Bk[a]` in the sequence `B[a]`, if `Bk[a] : Type.{j}`, we have `i` ≥ `j`.

In other words, the universe level `i` 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 `C` 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`. The exception we alluded to in the discussion of equality above is this: we are allowed to eliminate to an arbitrary `Type` when there is only one constructor, and each constructor argument is either in `Prop` or an index. This exception, which makes it possible to treat ordinary equality and heterogeneous equality as inductive types, can be justified by the fact that the elimination rule cannot take advantage of any "hidden" information.

Because inductive types can be polymorphic over universe levels, whether an inductive definition lands in `Prop` could, in principle, depend on how the universe levels are instantiated. To simplify the generation of the recursors, Lean adopts a convention that rules out this ambiguity: if you do not specify that the inductive type is an element of `Prop`, Lean requires the universe level to be at least one. Hence, a type specified by single inductive definition is either always in `Prop` or never in `Prop`. For example, if `A` and `B` are elements of `Prop`, `A × B` is assumed to have universe level at least one, representing a datatype rather than a proposition. The analogous definition of `A × B`, where `A` and `B` are restricted to `Prop` and the resulting type is declared to be an element of `Prop` instead of `Type`, is exactly the definition of `A ∧ B`.

7

Induction and Recursion

Other than the type universes and Pi types, inductively defined types provide the only means of defining new types in the Calculus of Inductive Constructions. We have also seen that, fundamentally, 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 for these types.

Working with induction and recursion is therefore fundamental to working in the Calculus of Inductive Constructions. For that reason Lean provides more natural ways of defining recursive functions, performing pattern matching, and writing inductive proofs. Behind the scenes, these are "compiled" down to recursors, using some of the auxiliary definitions described in Section Automatically Generated Constructions. Thus, the function definition package, which performs this reduction, is not part of the trusted code base.

Pattern Matching

The `cases_on` recursor can be used to define functions and prove theorems by cases. But complicated definitions may use several nested `cases_on` applications, and may be hard to read and understand. Pattern matching provides a more convenient and standard way of defining functions and proving theorems. Lean supports a very general form of pattern matching called *dependent pattern matching*.

A pattern-matching definition is of the following form:

```
definition [name] [parameters] : [domain] → [codomain]
| [name] [patterns_1] := [value_1]
...
| [name] [patterns_n] := [value_n]
```

The parameters are fixed, and each assignment defines the value of the function for a different case specified by the given pattern. As a first example, we define the function `sub2` for natural numbers:

```
open nat

definition sub2 : nat → nat
| sub2 0     := 0
| sub2 1     := 0
| sub2 (a+2) := a

example : sub2 5 = 3 := rfl
```

The default compilation method guarantees that the pattern matching equations hold definitionally.

```
example : sub2 0 = 0 := rfl

example : sub2 1 = 0 := rfl

example (a : nat) : sub2 (a + 2) = a := rfl
```

We can use the command `print definition` to inspect how our definition was compiled into recursors.

```
print definition sub2
```

We will say a term is a *constructor application* if it is of the form `c a_1 ... a_n` where `c` is the constructor of some inductive datatype. Note that in the definition `sub2`, the terms `1` and `a+2` are not constructor applications. However, the compiler normalizes them at compilation time, and obtains the constructor applications `succ zero` and `succ (succ a)` respectively. This normalization step is just a convenience that allows us to write definitions resembling the ones found in textbooks. There is no magic here: the compiler simply uses the kernel's ordinary evaluation mechanism. If we had written `2+a`, the definition would be rejected since `2+a` does not normalize into a constructor application.

In the next example, we use pattern-matching to define Boolean negation `bnot`, and proving `bnot (bnot b) = b`.

```
open bool

definition bnot : bool → bool
| bnot tt := ff
| bnot ff := tt

theorem bnot_bnot : ∀ (b : bool), bnot (bnot b) = b
| bnot_bnot tt := rfl    -- proof that bnot (bnot tt) = tt
| bnot_bnot ff := rfl    -- proof that bnot (bnot ff) = ff
```

As described in Chapter Inductive Types, Lean inductive datatypes can be parametric. The following example defines the `tail` function using pattern matching. The argument `A : Type` is a parameter and occurs before the colon to indicate it does not participate in the pattern matching. Lean allows parameters to occur after `:`, but it cannot pattern match on them.

```
import data.list
open list

definition tail {A : Type} : list A → list A
| tail nil      := nil
| tail (h :: t) := t

-- Parameter A may occur after ':'
definition tail2 : Π {A : Type}, list A → list A
| tail2 (@nil A) := (@nil A)
| tail2 (h :: t) := t

-- @ is allowed on the left-hand-side
definition tail3 : Π {A : Type}, list A → list A
| @tail3 A nil      := nil
| @tail3 A (h :: t) := t

-- A is explicit parameter
definition tail4 : Π (A : Type), list A → list A
| tail4 A nil      := nil
| tail4 A (h :: t) := t
```

Structural Recursion and Induction

The function definition package supports structural recursion, that is, recursive applications where one of the arguments is a subterm of the corresponding term on the left-hand-side. Later, we describe how to compile recursive equations using well-founded recursion. The main advantage of the default compilation method is that the recursive equations hold definitionally.

Here are some examples from the last chapter, written in the new style:

```
definition add : nat → nat → nat
| add m zero     := m
| add m (succ n) := succ (add m n)

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_add zero     := rfl
| zero_add (succ n) := eq.subst (zero_add n) rfl
```

```
definition mul : nat → nat → nat
| mul n zero     := zero
| mul n (succ m) := mul n m + m
```

The "definition" of `zero_add` makes it clear that proof by induction is really a form of induction in Lean.

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 be written as follows:

```
definition add (m : nat) : nat → nat
| add zero     := m
| add (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 below.

A more interesting example of structural recursion is given by the Fibonacci function `fib`. The subsequent theorem, `fib_pos`, combines pattern matching, recursive equations, and calculational proof.

```
import data.nat
open nat algebra

definition fib : nat → nat
| fib 0     := 1
| fib 1     := 1
| fib (a+2) := fib (a+1) + fib a

-- the defining equations hold definitionally
example : fib 0 = 1 := rfl
example : fib 1 = 1 := rfl
example (a : nat) : fib (a+2) = fib (a+1) + fib a := rfl

-- fib is always positive
theorem fib_pos : ∀ n, 0 < fib n
| fib_pos 0     := show 0 < 1, from zero_lt_succ 0
| fib_pos 1     := show 0 < 1, from zero_lt_succ 0
| fib_pos (a+2) := show 0 < fib (a+1) + fib a, from calc
    0 = 0 + 0              : rfl
  ... < fib (a+1) + 0      : add_lt_add_right (fib_pos (a+1)) 0
  ... < fib (a+1) + fib a : add_lt_add_left  (fib_pos a)     (fib (a+1))
```

Another classic example is the list `append` function.

```
import data.list
open list

definition append {A : Type} : list A → list A → list A
| append nil    l := l
| append (h::t) l := h :: append t l

example : append [(1 : ℕ), 2, 3] [4, 5] = [1, 2, 3, 4, 5] := rfl
```

Dependent Pattern-Matching

All the examples we have seen so far can be easily written using `cases_on` and `rec_on`. However, this is not the case with indexed inductive families, such as `vector A n`. A lot of boilerplate code needs to be written 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 A (succ n)` and deletes the first element. A first thought might be to use the `cases_on` function:

```
open nat

inductive vector (A : Type) : nat → Type :=
| nil {} : vector A zero
| cons   : Π {n}, A → vector A n → vector A (succ n)

open vector
notation h :: t := cons h t

check @vector.cases_on
-- Π {A : Type}
--   {C : Π (a : ℕ), vector A a → Type}
--   {a : ℕ}
--   (n : vector A a),
--   (e1 : C 0 nil)
--   (e2 : Π {n : ℕ} (a : A) (a_1 : vector A n), C (succ n) (cons a a_1)),
--   C a n
```

But what value should we return in the `nil` case? Something funny is going on: if `v` has type `vector A (succ n)`, it *can't* be nil, but it is not clear how to tell that to `cases_on`.

One standard solution is to define an auxiliary function:

```
definition tail_aux {A : Type} {n m : nat} (v : vector A m) :
    m = succ n → vector A n :=
vector.cases_on v
  (assume H : 0 = succ n, nat.no_confusion H)
  (take m (a : A) w : vector A m,
    assume H : succ m = succ n,
      have H1 : m = n, from succ.inj H,
      eq.rec_on H1 w)
```

```
definition tail {A : Type} {n : nat} (v : vector A (succ n)) : vector A n :=
tail_aux v rfl
```

In the nil case, m is instantiated to 0, and no_confusion (discussed in Section Automatically Generated Constructions) 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 = succ n in tail_aux is used to "communicate" the relationship between n and the index associated with the minor premise. Moreover, the zero = succ n 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 function definition package generates all the boilerplate code automatically for us.

Here are a number of examples:

```
definition head {A : Type} : Π {n}, vector A (succ n) → A
| head (h :: t) := h

definition tail {A : Type} : Π {n}, vector A (succ n) → vector A n
| tail (h :: t) := t

theorem eta {A : Type} : ∀ {n} (v : vector A (succ n)), head v :: tail v = v
| eta (h::t) := rfl

definition map {A B C : Type} (f : A → B → C)
               : Π {n : nat}, vector A n → vector B n → vector C n
| map nil     nil     := nil
| map (a::va) (b::vb) := f a b :: map va vb

definition zip {A B : Type} : Π {n}, vector A n → vector B n → vector (A × B) n
| zip nil nil         := nil
| zip (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
/-
definition map : Π {A : Type} {B : Type} {C : Type},
  (A → B → C) → (Π {n : ℕ}, vector A n → vector B n → vector C n)
λ (A : Type) (B : Type) (C : Type) (f : A → B → C) {n : ℕ}
(a : vector A n) (a_1 : vector B n),
  nat.brec_on n
    (λ {n : ℕ} (b : nat.below n) (a : vector A n) (a_1 : vector B n),
       nat.cases_on n
         (λ (b : nat.below 0) (a : vector A 0) (a_1 : vector B 0),
            (λ (t_1 : ℕ) (a_2 : vector A t_1),
               vector.cases_on a_2
```

```
                (λ (H_1 : 0 = 0) (H_2 : a == nil),
                   (λ (t_1 : ℕ) (a_1_1 : vector B t_1),
                     vector.cases_on a_1_1
                       (λ (H_1 : 0 = 0) (H_2 : a_1 == nil), nil)
                       (λ (n : ℕ) (a : B) (a_2 : vector B n)
                        (H_1 : 0 = succ n),
                          nat.no_confusion H_1))
                    0
                    a_1
                    (eq.refl 0)
-/
```

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.

The name of the function being defined can be omitted from the left-hand side of pattern matching equations. This feature is particularly useful when the function name is long or there are many cases. When the name is omitted, Lean will silently include @f in the left-hand-side of every pattern matching equation, where f is the name of the function being defined. Here is an example:

```
variables {A B : Type}
definition unzip : Π {n : nat}, vector (A × B) n → vector A n × vector B n
| zero     nil         := (nil, nil)
| (succ n) ((a, b)::v) :=
  match unzip v with
    (va, vb) := (a :: va, b :: vb)
  end

example : unzip (((1 : ℕ), (10 : ℕ)) :: (2, 20) :: nil) =
            (1 :: 2 :: nil, 10 :: 20 :: nil) :=
rfl
```

Variations on Pattern Matching

We say that a set of recursive equations *overlaps* when there is an input that more than one left-hand-side can match. In the following definition the input 0 0 matches the left-hand-side of the first two equations. Should the function return 1 or 2?

```
definition f : nat → nat → nat
| f 0     y     := 1
| f x     0     := 2
| f (x+1) (y+1) := 3
```

Overlapping patterns are often used to succinctly express complex patterns in data, and they are allowed in Lean. Lean handles the ambiguity by using the first applicable equation. In the example above, the following equations hold definitionally:

```
variables (a b : nat)

example : f 0     0     = 1 := rfl
example : f 0     (a+1) = 1 := rfl
example : f (a+1) 0     = 2 := rfl
example : f (a+1) (b+1) = 3 := rfl
```

Lean also supports *wildcard patterns*, also known as *anonymous variables*. They are used to create patterns where we don't care about the value of a specific argument. In the function `f` defined above, the values of `x` and `y` are not used in the right-hand-side. Here is the same example using wildcards:

```
open nat
definition f : nat → nat → nat
| f 0  _  := 1
| f _  0  := 2
| f _  _  := 3
variables (a b : nat)
example : f 0     0     = 1 := rfl
example : f 0     (a+1) = 1 := rfl
example : f (a+1) 0     = 2 := rfl
example : f (a+1) (b+1) = 3 := rfl
```

Some functional 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, discussed in Chapter Type Classes. Roughly, an element of `inhabited A` is simply a witness to the fact that there is an element of `A`; in Chapter Type Classes, we will see that Lean can be instructed that suitable base types are inhabited, and can automatically infer that other constructed types are inhabited on that basis. On this basis, the standard library provides an arbitrary element, `arbitrary A`, of any inhabited type.

We can also use the type `option A` 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.

```
open nat option

definition f1 : nat → nat → nat
| f1 0  _  := 1
| f1 _  0  := 2
| f1 _  _  := arbitrary nat   -- the "incomplete" case

variables (a b : nat)

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
```

```
definition f2 : nat → nat → option nat
| f2 0  _  := some 1
| f2 _  0  := some 2
| f2 _  _  := 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
```

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 inaccesible by writing `⌞t⌟` (the brackets are entered as `\cll` and `\clr`, for "corner-lower-left" and "corner-lower-right") or `?(t)`.

The following example can be found in [3]. 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 not 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.

```
variables {A B : Type}
inductive image_of (f : A → B) : B → Type :=
imf : Π a, image_of f (f a)

open image_of

definition inverse : Π f : A → B, Π b, image_of f b → A
| inverse f ⌞f a⌟ (imf _ _) := a
```

Inaccessible terms can also be used to reduce the complexity of the generated definition. Dependent pattern matching is compiled using the `cases_on` and `no_confusion` constructions. The number of instances of `cases_on` introduced by the compiler can be reduced by marking parts that only report specialization. In the next example, we define the type of finite ordinals `finord n`, a type with `n` inhabitants. We also define the function `to_nat` that maps an element of `finord n` to an elmeent of `nat`. If we do not mark `n+1` as inaccessible, the compiler will generate a definition containing two `cases_on` expressions. We encourage you to replace `⌞n+1⌟` with `(n+1)` in the next example and inspect the generated definition using `print definition to_nat`.

```
open nat

inductive finord : nat → Type :=
| fz : Π n, finord (succ n)
| fs : Π {n}, finord n → finord (succ n)

open finord

definition to_nat : Π {n : nat}, finord n → nat
| @to_nat ⌞n+1⌟ (fz n) := zero
| @to_nat ⌞n+1⌟ (fs f) := succ (to_nat f)
```

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.

```
definition is_not_zero (a : nat) : bool :=
match a with
| zero   := ff
| succ _ := tt
end

-- We can use recursive equations and match
variable {A : Type}
variable p : A → bool

definition filter : list A → list A
| filter nil      := nil
| filter (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
```

You can also use pattern matching in a local `have` expression:

```
import data.nat logic
open bool nat

definition mult : nat → nat → nat :=
have plus : nat → nat → nat
| 0        b := b
```

```
| (succ a) b := succ (plus a b),
have mult : nat → nat → nat
| 0        b := 0
| (succ a) b := plus (mult a b) b,
mult
```

Other Examples

In some definitions, we have to help the compiler by providing some implicit arguments explicitly in the left-hand-side of recursive equations. In such cases, if we don't provide the implicit arguments, the elaborator is unable to solve some placeholders (i.e.~meta-variables) in the nested match expression.

```
variables {A B : Type}
definition unzip : Π {n : nat}, vector (A × B) n → vector A n × vector B n
| @unzip zero     nil         := (nil, nil)
| @unzip (succ n) ((a, b)::v) :=
  match unzip v with
    (va, vb) := (a :: va, b :: vb)
  end

example : unzip (((1 : ℕ), (10 : ℕ)) :: (2, 20) :: nil) =
  (1 :: 2 :: nil, 10 :: 20 :: nil) :=
rfl
```

Next, we define the function `diag` which extracts the diagonal of a square matrix `vector (vector A n) n`. Note that, this function is defined by structural induction. However, the term `map tail v` is not a subterm of `((a :: va) :: v)`. Could you explain what is going on?

```
variables {A B : Type}

definition tail : Π {n}, vector A (succ n) → vector A n
| tail (h :: t) := t

definition map (f : A → B)
               : Π {n : nat}, vector A n → vector B n
| map nil      := nil
| map (a::va)  := f a :: map va

definition diag : Π {n : nat}, vector (vector A n) n → vector A n
| diag nil                := nil
| diag ((a :: va) :: v) := a :: diag (map tail v)
```

Well-Founded Recursion

[TODO: write this section.]

8

Building Theories and Proofs

In this chapter, we return to a discussion of some of the pragmatic features of Lean that support the development of structured theories and proofs.

More on Coercions

In Section Coercions, we discussed coercions briefly. The goal of this section is to provide a more precise account.

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 `A`, we can view any element `l : list A` as an element of `set A`, namely, the set of elements occurring in the list. The corresponding coercion is defined on the "family" of types `list A`, parameterized by `A`.

In fact, 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 type theory terminology, an element `F : Π x1 : A1, ..., xn : An, Type` is called a *family of types*. For every sequence of arguments `a1 : A1, ..., an : An`, `F a1 ... an` is a type, so we think of `F` as being a family parameterized by these arguments. A coercion of the first kind is of the form

```
c : Π x1 : A1, ..., xn : An, y : F x1 ... xn, G b1 ... bm
```

where `G` is another family of types, and the terms `b1 ... bn` depend on `x1, ..., xn, y`. This allows us to write `f t` where `t` is of type `F a1 ... an` but `f` expects an argument of type `G y1 ... ym`, for some `y1 ... ym`. For example, if `F` is `list : Π A : Type, Type`, `G` is `set Π A : Type, Type`, then a coercion `c : Π A : Type, list A → set A` allows us to pass an argument of type `list T` for some `T` any time an element of type `set T` is expected. These are the types of coercions we considered in Section Coercions.

Let us now consider the second kind of coercion. By the *class of sorts*, we mean the collection of universes `Type.{i}`. A coercion of the second kind is of the form

```
c : Π x1 : A1, ..., xn : An, F x1 ... xn → Type
```

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. We will see in a later chapter that this is very useful when defining algebraic structures in which one component, the carrier of the structure, is a `Type`. For example, we can define a semigroup as follows:

```
structure Semigroup : Type :=
(carrier : Type)
(mul : carrier → carrier → carrier)
(mul_assoc : ∀ a b c : carrier, mul (mul a b) c = mul a (mul b c))

notation a `*` b := Semigroup.mul _ a b
```

In other words, a semigroup consists of a type, `carrier`, and a multiplication, `mul`, with the property that the multiplication is associative. The `notation` command allows us to write `a * b` instead of `Semigroup.mul S a b` whenever we have `a b : carrier S`; 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`:

```
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 : Semigroup.carrier S`:

```
attribute Semigroup.carrier [coercion]

example (S : Semigroup) (a b : S) : a * b * a = a * (b * a) :=
!Semigroup.mul_assoc
```

It is the coercion that makes it possible to write `(a b : S)`.

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:

```
structure morphism (S1 S2 : Semigroup) : Type :=
(mor : S1 → S2)
(resp_mul : ∀ a b : S1, mor (a * b) = (mor a) * (mor b))
```

In other words, a morphism from `S1` to `S2` 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:

```
check morphism.mor    -- morphism ?S1 ?S2 → ?S1 → ?S2
```

As a result, it is a prime candidate for the third type of coercion.

```
attribute morphism.mor [coercion]

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 : morphism.resp_mul f
            ... = f a * f a * f a : morphism.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.

Remember that you can create a coercion whose scope is limited to the current namespace or section using the `local` modifier:

```
local attribute morphism.mor [coercion]
```

You can also declare a persistent coercion by assigning the attribute when you define the function initially, as described in Section Coercions. Coercions that are defined in a namespace "live" in that namespace, and are made active when the namespace is opened. If you want a coercion to be active as soon as a module is imported, be sure to declare it at the "top level," i.e. outside any namespace.

Remember also that you can instruct Lean's pretty-printer to show coercions with `set_option`, and you can print all the coercions in the environment using `print coercions`:

```
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 : morphism.resp_mul f
            ... = f a * f a * f a : morphism.resp_mul f

set_option pp.coercions true
check test

print coercions
```

Lean will also chain coercions as necessary. You can think of the coercion declarations as forming a directed graph where the nodes are families of types and the edges are the coercions between them. More precisely, each node is either a family of types, or the class of sorts, of the class of function types. The latter two are sinks in the graph. Internally, Lean automatically computes the transitive closure of this graph, in which the "paths" correspond to chains of coercions.

More on Implicit Arguments

In Section Implicit Arguments, we discussed implicit arguments. For example, if a term `t` has type Π `{x : A}, P x`, the variable `x` is *implicit* in `t`, which 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.

Dual to the `@` symbol is the exclamation mark, `!`, which essentially makes explicit arguments implicit by inserting underscores for them. Look at the terms that result from the following definitions to see this in action:

```
definition foo (n m k l : ℕ) : (n - m) * (k + l) = (k + l) * (n - m) := !mul.comm

print foo
-- definition foo : ∀ (n m k l : ℕ), (n - m) * (k + l) = (k + l) * (n - m)
-- λ (n m k l : ℕ), mul.comm (n - m) (k + l)

definition foo2 (n m k l : ℕ) : (n + k) + l = (k + l) + n := !add.assoc ⬝ !add.comm

print foo2
```

```
-- definition foo2 : ∀ (n : ℕ), ℕ → (∀ (k l : ℕ), n + k + l = k + l + n)
-- λ (n m k l : ℕ), add.assoc n k l ⬝ add.comm n (k + l)

definition foo3 (l : ℕ) (H : ∀ (n : ℕ), l + 2 ≠ 2 * (n + 1)) (n : ℕ) : l ≠ 2 * n :=
assume K : l = 2 * n,
absurd (show l + 2 = 2 * n + 2, from K ▶ rfl) !H

print foo3
-- definition foo3 : ∀ (l : ℕ),
--   (∀ (n : ℕ), l + 2 ≠ 2 * (n + 1)) → (∀ (n : ℕ), l ≠ 2 * n)
-- λ (l : ℕ) (H : ∀ (n : ℕ), l + 2 ≠ 2 * (n + 1)) (n : ℕ)
-- (K : l = 2 * n),
--   absurd (show l + 2 = 2 * n + 2, from K ▶ rfl) (H n)
```

In the first two examples, the exclamation mark indicates that the arguments to `mul.comm`, `add.assoc`, and `add.comm` should be made implicit, saving us the trouble of having to write lots of underscores. Note, by the way, that in the last example we use a neat trick: to show `l + 2 = 2 * n + 2` we take the reflexivity proof `rfl : l + 2 = l + 2` and then substitute `2 * n` for the second `l`.

More precisely, if `t` is of a function type, the expression `!t` makes all the arguments implicit up until the first argument that cannot be inferred from later arguments or the return type. This is usually what you want; for example, when applying a theorem, the arguments can often be inferred from context, but the hypothesis need to be provided explicitly.

In the following example, we declare `P` and `p` without implicit arguments, and then use the exclamation mark to make them implicit after the fact.

```
variables (P : Π (n m : ℕ) (v : vector bool n) (w : vector bool m), Type)
          (p : Π (n m : ℕ) (v : vector bool n) (w : vector bool m), P n m v w)
          (n m : ℕ) (v : vector bool n) (w : vector bool m)

set_option pp.metavar_args false
eval (!p : P n m v w)     -- p n m v w
eval (!p : P n n v v)     -- p n n v v
check !p                  -- p ?n ?m ?v ?w : P ?n ?m ?v ?w

eval (!P v w : Type)      -- P n m v w
eval (!p : !P w v)        -- p m n w v
```

Notice that we set `pp.metavar_args` to simplify the output. In the first `eval`, the expression `!p` inserts underscores for all explicit arguments of `p`, because the values of all of the placeholders in `p _ _ _ _` can be inferred from its type `P n m v w`. The same is true in the second example. In the subsequent `check` statement, the arguments of `p` are inserted, but cannot be inferred. Hence there are still metavariables in the output.

For `P` things are different: if we know that the type of `P _ _ _ _` is `Type`, we don't have enough information to assign values to the holes. However, we can fill the first two holes if we are given the last two arguments. Thus `!P v w` is interpreted as `P _ _ v w`, and from this we can infer that the holes must be `n` and `m`, respectively.

Here are some more examples of this behavior.

```
check @add_lt_add_right

definition foo (n m k : ℕ) (H : n < m) : n + k < m + k := !(add_lt_add_right H)

example {n m k l : ℕ} (H : n < m) (K : m + l < k + l) : n < k + l :=
calc
    n ≤ n + l : !le_add_right
  ... < m + l : !foo H
  ... < k + l : K
```

In the following example we show that a reflexive euclidean relation is both symmetric and transitive. Notice that we set the variable `R` to be an explicit argument of `reflexive`, `symmetric`, `transitive` and `euclidean`. However, for the theorems it is more convenient to make `R` implicit. We can do this with the command `variable {R}`, which makes `R` implicit from that point on.

```
variables {A : Type} (R : A → A → Prop)

definition reflexive  : Prop := ∀ (a : A), R a a
definition symmetric  : Prop := ∀ {a b : A}, R a b → R b a
definition transitive : Prop := ∀ {a b c : A}, R a b → R b c → R a c
definition euclidean  : Prop := ∀ {a b c : A}, R a b → R a c → R b c

variable {R}

theorem th1 (refl : reflexive R) (eucl : euclidean R) : symmetric R :=
take a b : A, assume (H : R a b),
show R b a, from eucl H !refl

theorem th2 (symm : symmetric R) (eucl : euclidean R) : transitive R :=
take (a b c : A), assume (H : R a b) (K : R b c),
have H' : R b a, from symm H,
show R a c, from eucl H' K

-- ERROR:
/-
  theorem th3 (refl : reflexive R) (eucl : euclidean R) : transitive R :=
  th2 (th1 refl eucl) eucl
-/

theorem th3 (refl : reflexive R) (eucl : euclidean R) : transitive R :=
@th2 _ _ (@th1 _ _ @refl @eucl) @eucl
```

However, when we want to combine `th1` and `th2` into `th3` we notice something funny. If we just write the proof `th2 (th1 refl eucl) eucl` we get an error. The reason is that `eucl` has type `∀ {a b c : A}, R a b → R a c → R b c`, hence `eucl` is interpreted as `@eucl _ _ _`. Similarly, the types of `th1` and `th2` start with a quantification over implicit arguments, hence they are interpreted as `th1 _ _` and `th2 _ _ _`, respectively. We can solve this by writing `@eucl`, `@th1` and `@th2`, but this is very inconvenient.

A better solution is to use a weaker form of implicit argument, marked with the binders ⦃ and ⦄, or, equivalently, `{{` and `}}`. The first two can be inserted by typing `\{{` and `\}}`, respectively.

```
definition symmetric  : Prop := ∀ ⦃a b : A⦄, R a b → R b a
definition transitive : Prop := ∀ ⦃a b c : A⦄, R a b → R b c → R a c
definition euclidean  : Prop := ∀ ⦃a b c : A⦄, R a b → R a c → R b c
```

Arguments in these binders are still implicit, but they are not added to a term `t` until `t` is applied to something. In other words, given an expression `t` of function type, if the next argument to `t` is a strong implicit argument, marked with `{` and `}`, that implicit argument is asserted aggressively; if the next argument to `t` is a weaker implicit argument, marked with ⦃ and ⦄, the implicit argument is not inserted unless the term is applied to something else. With `H : symmetric R`, this is what we want. Because we now have `H : ∀ ⦃a b : A⦄, R a b → R b a`, the expression `H` is interpreted as `@H`, but `H p` is interpreted as `@H _ _ p`. This allows us to prove `th3` in the expected way.

```
theorem th3 (refl : reflexive R) (eucl : euclidean R) : transitive R :=
th2 (th1 refl eucl) eucl
```

There is a third kind of implicit argument, used for type classes, and denoted with square brackets, `[` amd `]`. We will explain these kinds of arguments in Chapter Type Classes.

Elaboration and Unification

When you enter an expression like `λ x y z, f (x + y) z` for Lean to process, 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.

The process of taking a partially-specified expression and inferring what is left implicit is known as *elaboration*. Lean's elaboration algorithm is powerful, but at the same time, subtle and complex. Working in a system of dependent type theory requires knowing what sorts of information the elaborator can reliably infer, as well as knowing how to respond to error messages that are raised when the elaborator fails. To that end, it is helpful to have a general idea of how Lean's elaborator works.

When Lean is parsing an expression, it first enters a preprocessing phase. First, Lean inserts "holes" for implicit arguments. If term `t` has type `Π {x : A}, P x`, then `t` is replaced by `@t _` everywhere. Then, the holes — either the ones inserted in the previous step or the ones explicitly written by the user — in a term are instantiated by *metavariables* `?M1, ?M2, ?M3, ...`. Each overloaded notation is associated with a list of choices, that is, the possible interpretations. Similarly, Lean tries to detect the points where a coercion may need to be inserted in an application `s t`, to make the inferred type of `t` match the argument type of `s`. These become choice points too. If one possible outcome of the elaboration procedure is that no coercion is needed, then one of the choices on the list is the identity.

After preprocessing, Lean extracts a list of constraints that need to be solved in order for the term to have a valid type. Each application term `s t` gives rise to a constraint `T1 = T2`, where `t` has type `T1` and `s` has type `Π x : T2, T3`. Notice that the expressions `T1` and `T2` will often contain metavariables; they may even be metavariables themselves. Moreover, a definition of the form `definition foo : T := t` or a theorem of the form `theorem bar : T := t` generates the constraint that the inferred type of `t` should be `T`.

The elaborator now has a straightforward task: find expressions to substitute for all the metavariables so that all of the constraints are simultaneously satisfied. An assignment of terms to metavariables is known as a *substitution*, and the general task of finding a substitution that makes two expressions coincide is known as a *unification problem*. (If only one of the expressions contains metavariables, the task is a special case known as a *matching problem*.)

Some constraints are straightforwardly handled. If `f` and `g` are distinct constants, it is clear that there is no way to unify the terms `f s_1 ... s_m` and `g t_1 ... t_n`. On the other hand, one can unify `f s_1 ... s_m` and `f t_1 ... t_m` by unifying `s_1` with `t_1`, `s_2` with `t_2`, and so on. If `?M` is a metavariable, one can unify `?M` with any term `t` simply by assigning `t` to `?M`. These are all aspects of *first-order* unification, and such constraints are solved first.

In contrast, *higher-order* unification is much more tricky. Consider, for example, the expressions `?M a b` and `f (g a) b b`. All of the following assignments to `?M` are among the possible solutions:

- `λ x y, f (g x) y y`
- `λ x y, f (g x) y b`
- `λ x y, f (g a) b y`
- `λ x y, f (g a) b b`

Such problems arise in many ways. For example:

- When you use `induction_on x` for an inductively defined type, Lean has to infer the relevant induction predicate.
- When you write `eq.subst e p` with an equation `e : a = b` to convert a proposition `P a` to a proposition `P b`, Lean has to infer the relevant predicate.

- When you write `sigma.mk a b` to build an element of `Σ x : A, B x` from an element `a : A` and an element `B : B a`, Lean has to infer the relevant `B`. (And notice that there is an ambiguity; `sigma.mk a b` could also denote an element of `Σ x : A, B a`, which is essentially the same as `A × B a`.)

In cases like this, Lean has to perform a backtracking search to find a suitable value of a higher-order metavariable. It is known that even second-order unification is generally undecidable. The algorithm that Lean uses is not complete (which means that it can fail to find a solution even if one exists) and potentially nonterminating. Nonetheless, it performs quite well in ordinary situations.

Moreover, the elaborator performs a global backtracking search over all the nondeterministic choice points introduced by overloads and coercions. In other words, the elaborator starts by trying to solve the equations with the first choice on each list. Each time the procedure fails, it analyzes the failure, and determines the next viable choice to try.

To complicate matters even further, sometimes the elaborator has to reduce terms using the internal computation rules of the formal system. For example, if it happens to be the case that `f` is defined to be `λ x, g x x`, we can unify expressions `f ?M` and `g a a` by assigning `?M` to `a`. In general, any number of computation steps may be needed to unify terms. It is computationally infeasible to try all possible reductions in the search, so, once again, Lean's elaborator relies on an incomplete strategy.

The interaction of computation with higher-order unification is particularly knotty. For the most part, Lean avoids performing computational reduction when trying to solve higher-order constraints. You can override this, however, by marking some symbols with the `reducible` attribute, as described in Section 8.4.

The elaborator relies on additional tricks and gadgets to solve a list of constraints and instantiate metavariables. Below we will see that users can specify that some parts of terms should be filled in by *tactics*, which can, in turn, invoke arbitrary automated procedures. In the next chapter, we will discuss the mechanism of `class inference`, which can be configured to execute a prolog-like search for appropriate instantiations of an implicit argument. These can be used to help the elaborator find implicit facts on the fly, such as the fact that a particular set is finite, as well as implicit data, such as a default element of a type, or the appropriate multiplication in an algebraic structure.

It is important to keep in mind that all these mechanisms interact. The elaborator processes its list of constraints, trying to solve the easier ones first, postponing others until more information is available, and branching and backtracking at choice points. Even small proofs can generate hundreds or thousands of constraints. The elaboration process continues until the elaborator fails to solve a constraint and has exhausted all its backtracking options, or until all the constraints are solved. In the first case, it returns an error message which tries to provide the user with helpful information as to where and why it failed. In the second case, the type checker is asked to confirm that the assignment that the elaborator has found does indeed make the term type check. If all the metavariables in the original expression have been assigned, the result is a fully elaborated, type-correct expression. Otherwise, Lean flags the sources of the remaining metavariables as "placeholders" or "goals" that could not be filled.

Reducible Definitions

Transparent identifiers can be declared to be *reducible* or *irreducible* or *semireducible*. By default, a definition is *semireducible*. This status provides hints that govern the way the elaborator tries to solve higher-order unification problems. As with other attributes, the status of an identifier with respect to reducibility has no bearing on type checking at all, which is to say, once a fully elaborated term is type correct, marking one of the constants it contains to be reducible does not change the correctness. The type checker in the kernel of Lean ignores such attributes, and there is no problem marking a constant reducible at one point, and then irreducible later on, or vice-versa.

The purpose of the annotation is to help Lean's unification procedure decide which declarations should be unfolded. The higher-order unification procedure has to perform case analysis, implementing a backtracking search. At various stages, the procedure has to decide whether a definition `C` should be unfolded or not.

- An *irreducible* definition will never be unfolded during higher-order unification (but can still be unfolded in other situations, for example during type checking).
- A *reducible* definition will be always eligible for unfolding.
- A definition which is *semireducible* can be unfolded during *simple* decisions and won't be unfolded during *complex* decisions. An unfolding decision is *simple* if the unfolding does not require the procedure to consider an extra case split. It is *complex* if the unfolding produces at least one extra case, and consequently increases the search space.

You can assign the `reducible` attribute when a symbol is defined:

```
definition pr1 [reducible] (A : Type) (a b : A) : A := a
```

The assignment persists to other modules. You can achieve the same result with the `attribute` command:

```
definition pr2 (A : Type) (a b : A) : A := b

-- mark pr2 as reducible
attribute pr2 [reducible]

-- mark pr2 as irreducible
attribute pr2 [irreducible]
```

The `local` modifier can be used to instruct Lean to limit the scope to the current namespace or section.

```
definition pr2 (A : Type) (a b : A) : A := b

local attribute pr2 [irreducible]
```

When reducibility hints are declared in a namespace, their scope is restricted to the namespace. In other words, even if you import the module in which the attributes are declared, they do not take effect until the namespace is opened. As with coercions, if you want a reducibility attribute to be set whenever a module is imported, be sure to declare it at the top level. See also Section 8.7 below for more information on how to import only the reducibility attributes, without exposing other aspects of the namespace.

Finally, we can go back to *semireducible* using the `attribute` command:

```
-- pr2 is semireducible
definition pr2 (A : Type) (a b : A) : A := b

-- mark pr2 as reducible
attribute pr2 [reducible]
-- ...
-- make it semireducible again
attribute pr2 [semireducible]
```

Helping the Elaborator

Because proof terms and expressions in dependent type theory can become quite complex, working in dependent type theory effectively involves relying on the system to fill in details automatically. When the elaborator fails to elaborate a term, there are two possibilities. One possibility is that there is an error in the term, and no solution is possible. In that case, your goal, as the user, is to find the error and correct it. The second possibility is that the term has a valid elaboration, but the elaborator failed to find it. In that case, you have to help the elaborator along by providing information. This section provides some guidance in both situations.

If the error message is not sufficient to allow you to identify the problem, a first strategy is to ask Lean's pretty printer to show more information, as discussed in Section Setting Options, using some or all of the following options:

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

The following option subsumes all of those settings:

```
set_option pp.all true
```

Sometimes, the elaborator will fail with the message that the unifier has exceeded its maximum number of steps. As we noted in the last section, some elaboration problems can lead to nonterminating behavior, and so Lean simply gives up after it has reached a pre-set maximum. You can change this with the `set_option` command:

```
set_option unifier.max_steps 100000
```

This can sometimes help you determine whether there is an error in the term or whether the elaboration problem has simply grown too complex. In the latter case, there are steps you can take to cut down the complexity.

To start with, Lean provides a mechanism to break large elaboration problems down into simpler ones, with a `proof ... qed` block. Here is the sample proof from Section Examples of Propositional Validities, with additional `proof ... qed` annotations:

```
example (p q r : Prop) : p ∧ (q ∨ r) ↔ (p ∧ q) ∨ (p ∧ r) :=
iff.intro
  (assume H : p ∧ (q ∨ r),
    show (p ∧ q) ∨ (p ∧ r), from
    proof
      have Hp : p, from and.elim_left H,
      or.elim (and.elim_right H)
        (assume Hq : q,
          show (p ∧ q) ∨ (p ∧ r), from or.inl (and.intro Hp Hq))
        (assume Hr : r,
          show (p ∧ q) ∨ (p ∧ r), from or.inr (and.intro Hp Hr))
    qed)
  (assume H : (p ∧ q) ∨ (p ∧ r),
    show p ∧ (q ∨ r), from
    proof
      or.elim H
        (assume Hpq : p ∧ q,
          have Hp : p, from and.elim_left Hpq,
          have Hq : q, from and.elim_right Hpq,
          show p ∧ (q ∨ r), from and.intro Hp (or.inl Hq))
        (assume Hpr : p ∧ r,
          have Hp : p, from and.elim_left Hpr,
          have Hr : r, from and.elim_right Hpr,
          show p ∧ (q ∨ r), from and.intro Hp (or.inr Hr))
    qed)
```

Writing `proof t qed` as a subterm of a larger term breaks up the elaboration problem as follows: first, the elaborator tries to elaborate the surrounding term, independent of `t`. If it succeeds, that solution is used to constrain the type of `t`, and the elaborator processes that term independently. The net result is that a big elaboration problem gets broken down into smaller elaboration problems. This "localizes" the elaboration procedure, which has both positive and negative effects. A disadvantage is that information is insulated, so that the solution to one problem cannot inform the solution to another. The key advantage is that it can simplify the elaborator's task. For example, backtracking points within a `proof ... qed` do not become backtracking points for the outside term; the elaborator either succeeds or fails to elaborate each independently. As another benefit, error messages are often improved; an error that ultimately stems from an incorrect choice of an overload in one subterm is not "blamed" on another part of the term.

In principle, one can write `proof t qed` for any term `t`, but it is used most effectively following a `have` or `show`, as in the example above. This is because `have` and `show` specify the intended type of the `proof ... qed` block, reducing any ambiguity about the subproblem the elaborator needs to solve.

The use of `proof ... qed` blocks with `have` and `show` illustrates two general strategies that can help the elaborator: first, breaking large problems into smaller problems, and, second, providing additional information. The first strategy can also be achieved by breaking a large definition into smaller definitions, or breaking a theorem with a large proof into auxiliary lemmas. Even breaking up long terms internal to a proof using auxiliary `have` statements can help locate the source of an error.

The second strategy, providing additional information, can be achieved by using `have`, `show`, `(t : T)` notation, and `#<namespace>` (see Section Coercions) to indicate expected types. More directly, it often helps to specify the implicit arguments. When Lean cannot solve for the value of a metavariable corresponding to an implicit argument, you can always use `@` to provide that argument explicitly. Doing so will either help the elaborator solve the elaboration problem, or help you find an error in the term that is blocking the intended solution.

In Lean, tactics not only allow us to invoke arbitrary automated procedures, but also provide an alternative approach to construct proofs and terms. For many users, this is one of the most effective mechanisms to help the elaborator. A tactic can be viewed as a "recipe", a sequence of commands or instructions, that describes how to build a proof. This recipe may be as detailed as we want. A tactic `T` can be embedded into proof terms by writing `by T` or `begin T end`. These annotations instruct Lean that tactic `T` should be invoked to construct the term in the given location. As with `proof ... qed`, the elaborator tries to elaborate the surrounding terms before executing `T`. In fact, the expression `proof t qed` is just syntactic sugar for `by exact t`, which invokes the `exact` tactic. We will discuss tactics in Chapter Tactic-Style Proofs.

If you are running Lean using Emacs, you can "profile" the elaborator and type checker, to find out where they are spending all their time. Type `M-x lean-execute` to run an independent Lean process manually and add the option `--profile`. The output buffer will then report the times required by the elaborator and type checker, for each definition and theorem processed. If you ever find the system slowing down while processing a file, this can help you locate the source of the problem.

Sections

Lean provides various sectioning mechanisms that help structure a theory. We saw in Section Variables and Sections that the `section` command makes it possible not only to group together elements of a theory that go together, but also to declare variables that are inserted as arguments to theorems and definitions, as necessary. In fact, Lean has two ways of introducing local elements into the sections, namely, as `variables` or as `parameters`.

Remember that the point of the variable command is to declare variables for use in theorems, as in the following example:

```
import standard
open nat algebra

section
  variables x y : ℕ

  definition double := x + x

  check double y
  check double (2 * x)

  theorem t1 : double x = 2 * x :=
  calc
    double x = x + x         : rfl
         ... = 1 * x + x     : one_mul
         ... = 1 * x + 1 * x : one_mul
         ... = (1 + 1) * x   : right_distrib
         ... = 2 * x         : rfl

  check t1 y
  check t1 (2 * x)

  theorem t2 : double (2 * x) = 4 * x :=
  calc
    double (2 * x) = 2 * (2 * x) : t1
               ... = 2 * 2 * x   : mul.assoc
               ... = 4 * x       : rfl
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. To ask Lean to include a variable in every definition in a section, use the `include` command. This is often useful with type classes, and is discussed in Section Instances in Sections in the next chapter.

Notice that 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.

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

```
import standard

section
  parameters {A : Type} (R : A → A → Type)
  hypothesis transR : ∀ {x y z}, R x y → R y z → R x z

  variables {a b c d e : A}

  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
```

Here, `hypothesis` functions as a synonym for parameter, so that `A`, `R`, and `transR` are all parameters in the section. This means that, as before, they are inserted as arguments to definitions and theorems as needed. But there is a difference: within the section, `t1` is an abbreviation for `@t1 A R transR`, which is to say, these arguments are fixed until the section is closed. 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. If you want to apply your theorems to arbitrary binary relations within the section, make `R` a variable.

Notice that Lean is consistent when it comes to providing alternative syntax for `Prop`-valued variants of declarations:

Type	Prop
constant	axiom
variable	premise
parameter	hypothesis
take	assume

Lean also allows you to use `conjecture` in place of `hypothesis`.

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
open int eq.ops algebra

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, from (sub_self a)⁻¹ ▶ !dvd_zero

  theorem mod_symm (H : a ≡ b) : b ≡ a :=
  have H1 : m ∣ -(b - a), from iff.mpr !dvd_neg_iff_dvd H,
  have H2 : m ∣ a - b, from neg_sub b a ▶ H1,
  H2

  theorem mod_trans (H1 : a ≡ b) (H2 : b ≡ c) : a ≡ c :=
  have H1 : m ∣ (c - b) + (b - a), from !dvd_add H2 H1,
  have H2 : m ∣ c - a, from eq.subst
    (calc
      (c - b) + (b - a) = c - b + b - a  : add.assoc
                    ... = c + -b + b - a : rfl
                    ... = c - a          : neg_add_cancel_right)
    H1,
  H2
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
```

More on Namespaces

Recall from Section Namespaces that namespaces not only package shorter names for theorems and identifiers, but also things like notation, coercions, classes, rewrite rules, and so on. You can ask Lean to display a list of these "metaclasses":

```
print metaclasses
```

These can be opened independently using modifiers to the `open` command:

```
import data.nat

open [declaration] nat
open [notation] nat
open [coercion] nat
open [class] nat
open [abbreviation] nat
```

For example, `open [coercion] nat` makes the coercions in the namespace `nat` available (and nothing else). You can multiple metaclasses on one line:

```
import data.nat

open [declaration] [notation] [coercion] nat
```

You can also open a namespace while *excluding* certain metaclasses. For example,

```
import data.nat

open - [notation] [coercion] nat
```

imports all metaclasses but `[notation]` and `[coercion]`. You can limit the scope of an `open` command by putting it in a section. For example, here we temporarily import notation from nat:

```
import data.nat

section
  open [notation] nat

  /- ... -/
end
```

You can also import only certain theorems by providing an explicit list in parentheses:

```
import data.nat

open nat (add sub_sub zero_div)

check add
check sub_sub
check zero_div
```

The `open` command above imports all metaobjects from `nat`, but limits the shortened identifiers to the ones listed. If you want to import *only* the shortened identifiers, use the following:

```
import data.nat
open [declaration] nat (add sub_sub zero_div)
```

When you open a section, you can rename identifiers on the fly:

```
import data.nat
open nat (renaming add -> plus)

check plus
```

Or you can *exclude* a list of items from being imported:

```
import data.nat
open nat (hiding add)
```

Within a namespace, you can declare certain identifiers to be `protected`. This means that when the namespace is opened, the short version of these names are not made available:

```
namespace foo
  protected definition bar (A : Type) (x : A) := x
end foo

open foo
check foo.bar  -- "check bar" yields an error
```

In the Lean library, common names are protected to avoid clashes. For example, we want to write `nat.rec_on`, `int.rec_on`, and `list.rec_on`, even when all of these namespaces are open, to avoid ambiguity and overloading. You can always define a local abbreviation to use the shorter name:

```
import data.list
open list
local abbreviation induction_on := @list.induction_on
check induction_on
```

Alternatively, you can "unprotect" the definition by renaming it when you open the namespace:

```
import data.list
open list (renaming induction_on → induction_on)
check induction_on
```

As yet a third alternative, you obtain an alias for the shorter name by opening the namespace for that identifier only:

```
import data.list
open list (induction_on)
check induction_on
```

You may find that at times you want to cobble together a namespace, with notation, rewrite rules, or whatever, from existing namespaces. Lean provides an `export` command for that. The `export` command supports the same options and modifiers as the `open` command: when you export to a namespace, aliases for all the items you export become part of the new namespace. For example, below we define a new namespace, `my_namespace`, which includes items from `bool`, `nat`, and `list`.

```
import standard

namespace my_namespace
  export bool (hiding measurable)
  export nat
  export list
end my_namespace

check my_namespace.band
check my_namespace.zero
check my_namespace.append

open my_namespace

check band
check zero
check append
```

This makes it possible for you to define nicely prepackaged configurations for those who will use your theories later on.

Sometimes it is useful to hide auxiliary definitions and theorems from the outside world, for example, so that they do not clutter up the namespace. The `private` keyword allows you to do this. The name of a `private` definition is only visible in the module/file where it was declared.

```
import data.nat
open nat

private definition inc (x : nat) := x + 1
private theorem inc_eq_succ (x : nat) : succ x = inc x :=
  rfl
```

In this example, the definition `inc` and theorem `inc_eq_succ` are not visible or accessible in modules that import this one.

9

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. 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.

Type Classes and Instances

The basic idea is simple. In Section More on Coercions, we saw that any family types can serve as the source or target of a coercion. In much the same way, any family of types can be marked as a *type class*. Then we can 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.

Here is a somewhat frivolous example:

```
import data.nat
open nat

attribute nat [class]

definition nat_one [instance] : ℕ := 1

definition foo [x : ℕ] : nat := x

check @foo
eval foo

example : foo = 1 := rfl
```

Here we declare `nat` to be a class with a "canonical" instance `1`. Then we declare `foo` to be, essentially, the identity function on the natural numbers, but we mark the argument implicit, and indicate that it should be inferred by type class inference. When we write `foo`, the preprocessor interprets it as `foo ?x`, where `?x` is an implicit argument. But when the elaborator gets hold of the expression, it sees that `?x : ℕ` is supposed to be solved by type class inference. It looks for a suitable element of the class, and it finds the instance `one`. Thus, when we evaluate `foo`, we simply get `1`.

It is tempting to think of `foo` as defined to be equal to `1`, but that is misleading. Every time we write `foo`, the elaborator searches for a value. If we declare other instances of the class, that can change the value that is assigned to the implicit argument. This can result in seemingly paradoxical behavior. For example, we might continue the development above as follows:

```
definition nat_two [instance] : ℕ := 2

eval foo

example : foo ≠ 1 := dec_trivial
```

Now the "same" expression `foo` evaluates to `2`. Whereas before we could prove `foo = 1`, now we can prove `foo ≠ 1`, because the inferred implicit argument has changed. When searching for a suitable instance of a type class, the elaborator tries the most recent instance declaration first, by default. We will see below, however, that it is possible to give individual instances higher or lower priority. The proof `dec_trivial` will be explained below.

As with `coercion` and other attributes, you can assign the `class` or `instance` attributes in a definition, or after the fact, with an `attribute` command. As usual, the assignments `attribute foo [class]` and `attribute foo [instance]` are only in effect in the current namespace, but the assignments persist on import. To limit the scope of an assignment to the current file, use the `local attribute` variant.

The reason the example is frivolous is that there is rarely a need to "infer" a natural number; we can just hard-code the choice of `1` or `2` into the definition of `foo`. Type classes become useful when they depend on parameters, in which case, the value that is inferred depends on these parameters.

Let us work through 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 `A` is a type, `∃ x : A, x = x` is true only if `A` 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 `A` when `l` is of type `list A`; but then we are faced with the problem that `head l` needs to return an "arbitrary" element of `A` in the case where `l` is the empty list, `nil`.

For purposes like this, 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. We will carry out a similar development in the examples that follow, using a namespace `hide` 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:

```
inductive inhabited [class] (A : Type) : Type :=
mk : A → inhabited A
```

An element of the class `inhabited A` is simply an expression of the form `inhabited.mk a`, for some element `a : A`. The eliminator for the inductive type will allow us to "extract" such an element of `A` from an element of `inhabited A`.

The second step of the program is to populate the class with some instances:

```
definition Prop.is_inhabited [instance] : inhabited Prop :=
inhabited.mk true

definition bool.is_inhabited [instance] : inhabited bool :=
inhabited.mk bool.tt

definition nat.is_inhabited [instance] : inhabited nat :=
inhabited.mk nat.zero

definition unit.is_inhabited [instance] : inhabited unit :=
inhabited.mk unit.star
```

This arranges things so that when type class inference is asked to infer an element `?M : Prop`, it can find the element `true` to assign to `?M`, and similarly for the elements `tt`, `zero`, and `star` of the types `bool`, `nat`, and `unit`, respectively.

The final step of the program is to define a function that infers an element `H : inhabited A` and puts it to good use. The following function simply extracts the corresponding element `a : A`:

```
definition default (A : Type) [H : inhabited A] : A :=
inhabited.rec (λ a, a) H
```

This has the effect that given a type expression `A`, whenever we write `default A`, we are really writing `default A ?H`, leaving the elaborator to find a suitable value for the metavariable `?H`. When the elaborator succeeds in finding such a value, it has effectively produced an element of type `A`, 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 A`, we are asking the elaborator to synthesize an element of type `A`.

Notice that we can "see" the value that is synthesized with `eval`:

```
eval default Prop     -- true
eval default nat      -- nat.zero
eval default bool     -- bool.tt
eval default unit     -- unit.star
```

We can also codify these choices as theorems:

```
example : default Prop = true := rfl
example : default nat = nat.zero := rfl
example : default bool = bool.tt := rfl
example : default unit = unit.star := rfl
```

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 A` instead of `default A`. 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.

Chaining Instances

If that were the extent of type class inference, it would not be all the impressive; it would be simply a mechanism of storing a list of instances for the elaborator to find in a lookup table. What makes type class inference powerful is that one can *chain* instances. That is, an instance declaration can in turn depend on an implicit instance of a type class. This causes class inference to chain through instances recursively, backtracking when necessary, in a Prolog-like search.

For example, the following definition shows that if two types `A` and `B` are inhabited, then so is their product:

```
definition prod.is_inhabited [instance] {A B : Type} [H1 : inhabited A]
  [H2 : inhabited B] : inhabited (prod A B) :=
inhabited.mk ((default A, default B))
```

With this added to the earlier instance declarations, type class instance can infer, for example, a default element of `nat × bool × unit`:

```
open prod

check default (nat × bool × unit)
eval default (nat × bool × unit)
```

Given the expression `default (nat × bool × unit)`, the elaborator is called on to infer an implicit argument `?M : inhabited (nat × bool × unit)`. The instance `inhabited_product` reduces this to inferring `?M1 : inhabited nat` and `?M2 : inhabited (bool × unit)`. The first one is solved by the instance `nat.is_inhabited`. The second invokes another application of `inhabited_product`, and so on, until the system has inferred the value `(nat.zero, bool.tt, unit.star)`.

Similarly, we can inhabit function spaces with suitable constant functions:

```
definition inhabited_fun [instance] (A : Type) {B : Type} [H : inhabited B] :
  inhabited (A → B) :=
inhabited.rec_on H (λ b, inhabited.mk (λ a, b))

check default (nat → nat × bool × unit)
eval default (nat → nat × bool × unit)
```

In this case, type class inference finds the default element `λ (a : nat), (nat.zero, bool.tt, unit.star)`.

As an exercise, try defining default instances for other types, such as sum types and the list type.

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. Nonetheless, as we will see, the implementation of the type class allows for a smooth transition between constructive and classical logic.

In the standard library, `decidable` is defined formally as follows:

```
inductive decidable [class] (p : Prop) : Type :=
| inl :  p → decidable p
| inr : ¬p → decidable p
```

Logically speaking, having an element `t : decidable p` is stronger than having an element `t : p ∨ ¬p`; it enables us to define values of an arbitrary type depending on the truth value of `p`. For example, for the expression `if p then a else b` to make sense, we need to know that `p` is decidable. That expression is syntactic sugar for `ite p a b`, where `ite` is defined as follows:

```
definition ite (c : Prop) [H : decidable c] {A : Type} (t e : A) : A :=
decidable.rec_on H (λ Hc, t) (λ Hnc, e)
```

The standard library also contains a variant of `ite` called `dite`, the dependent if-then-else expression. It is defined as follows:

```
definition dite (c : Prop) [H : decidable c] {A : Type} (t : c → A) (e : ¬ c → A) : A :=
decidable.rec_on H (λ Hc : c, t Hc) (λ Hnc : ¬ c, e Hnc)
```

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)`.

In the standard library, we cannot prove that every proposition is decidable. But we can prove that *certain* propositions are decidable. For example, we can prove that basic operations like equality and comparisons on the natural numbers and the integers are decidable. Moreover, decidability is preserved under propositional connectives:

```
check @decidable_and
-- Π {p q : Prop} [Hp : decidable p] [Hq : decidable q], decidable (p ∧ q)

check @decidable_or
check @decidable_not
check @decidable_implies
```

Thus we can carry out definitions by cases on decidable predicates on the natural numbers:

```
import standard

open nat

definition 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. When you import the classical axioms, then, `decidable p` has an instance for every `p`, and the elaborator infers that value quickly. Thus all theorems in the standard library that rely on decidability assumptions are freely available in the classical library.

This explains the "proof" `dec_trivial` in Section Type Classes and Instances above. The expression `dec_trivial` is actually defined in the module `init.logic` to be notation for the expression `of_is_true trivial`, where `of_is_true` infers the decidability of the theorem you are trying to prove, extracts the corresponding decision procedure, and confirms that it evaluates to `true`.

Overloading with Type Classes

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 `A → A → A` for some type `A`. We can use type classes to infer an appropriate addition function for suitable types `A`. We will see in the next section that this is especially useful for developing algebraic hierarchies of structures in a formal setting.

We can declare a type class `has_add A` as follows:

```
import standard

namespace hide

inductive has_add [class] (A : Type) : Type :=
mk : (A → A → A) → has_add A

definition add {A : Type} [s : has_add A] :=
has_add.rec (λ x, x) s

notation a `+` b := add a b

end hide
```

The class `has_add A` is supposed to be inhabited exactly when there is an appropriate addition function for `A`. The `add` function is designed to find an instance of `has_add A` for the given type, A, 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 the declare instances for nat, int, and bool:

```
definition has_add_nat [instance] : has_add nat :=
has_add.mk nat.add

definition has_add_int [instance] : has_add int :=
has_add.mk int.add

definition has_add_bool [instance] : has_add bool :=
has_add.mk bool.bor

open [coercion] nat int
open bool

set_option pp.notation false
check (2 : nat) + 2    -- nat
check (2 : int) + 2    -- int
check tt + ff          -- bool
```

In the example above, we expose the coercions in namespaces nat and int, so that we can use numerals. If we opened these namespace outright, the symbol + would be ad-hoc overloaded. This would result in an ambiguity as to which addition we have in mind when we write a + b for a b : nat. The ambiguity is benign, however, since the new interpretation of + for nat is definitionally equal to the usual one. Setting the option to turn off notation while pretty-printing shows us that it is the new add function that is inferred in each case. Thus we are relying on type class overloading to disambiguate the meaning of the expression, rather than ad-hoc overloading.

As with inhabited and decidable, 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 A and B, we can define addition on A × B pointwise:

```
definition has_add_prod [instance] {A B : Type} [sA : has_add A] [sB : has_add B] :
  has_add (A × B) :=
has_add.mk (take p q, (add (prod.pr1 p) (prod.pr1 q), add (prod.pr2 p) (prod.pr2 q)))

open nat

check (1, 2) + (3, 4)    -- ℕ × ℕ
eval (1, 2) + (3, 4)     -- (4, 6)
```

We can similarly define pointwise addition of functions:

```
definition has_add_fun [instance] {A B : Type} [sB : has_add B] :
  has_add (A → B) :=
```

```
has_add.mk (λ f g, λ x, f x + g x)

open nat

check (λ x : nat, (1 : nat)) + (λ x, (2 : nat))   -- ℕ → ℕ
eval (λ x : nat, (1 : nat)) + (λ x, (2 : nat))    -- λ (x : ℕ), 3
```

As an exercise, try defining instances of `has_add` for lists and vectors, and show that they have the work as expected.

Managing Type Class Inference

Recall from Section Displaying Information that you can ask Lean for information about the classes and instances that are currently in scope:

```
print classes
print instances inhabited
```

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 add this to your file in Emacs mode and use `C-c C-x` to run an independent Lean process on your file, 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 the Emacs Lean mode, 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.priority` in both the standard and hott libraries. 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:

```
open nat

inductive foo [class] :=
mk : nat → nat → foo

definition foo.a [p : foo] : nat := foo.rec_on p (λ a b, a)

definition i1 [instance] [priority std.priority.default+10] : foo :=
foo.mk 1 1

definition i2 [instance] : foo :=
foo.mk 2 2

example : foo.a = 1 := rfl

definition i3 [instance] [priority std.priority.default+20] : foo :=
foo.mk 3 3

example : foo.a = 3 := rfl

attribute i3 [instance] [priority 500]

example : foo.a = 1 := rfl

attribute i1 [instance] [priority std.priority.default-10]

example : foo.a = 2 := rfl
```

Instances in Sections

We can easily introduces instances of type classes in a section or context using variables and parameters. Recall that variables are only included in declarations when they are explicitly mentioned. Instances of type classes are rarely explicitly mentioned in definitions, so to make sure that an instance of a type class is included in every definition and theorem, we use the `include` command.

```
section
  variables {A : Type} [H : has_add A] (a b : A)
  include H

  definition foo : a + b = a + b := rfl
  check @foo
end
```

Note that the `include` command includes a variable in every definition and theorem in that section. If we want to declare a definition or theorem which does not use the instance, we can use the `omit` command:

```
section
  variables {A : Type} [H : has_add A] (a b : A)
  include H
  definition foo1 : a + b = a + b := rfl
  omit H
  definition foo2 : a  = a := rfl -- H is not an argument of foo2
  include H
  definition foo3 : a + a = a + a := rfl

  check @foo1
  check @foo2
  check @foo3
end
```

Bounded Quantification

A "bounded universal quantifier" is one that is of the form `∀ x : nat, x < n → P x`. As a final illustration of the power of type class inference, we show that a proposition of this form is decidable assuming `P` is, and that type class inference can make use of that fact.

First, we define `ball n P` as shorthand for ∀ x : nat, x < n → P x.

```
-- ∀ x : nat, x < 0 → P x
definition ball_zero (P : nat → Prop) : ball zero P :=
λ x Hlt, absurd Hlt !not_lt_zero

variables {n : nat} {P : nat → Prop}

-- (∀ x : nat, x < succ n → P x) implies (∀ x : nat, x < n → P x)
definition ball_of_ball_succ (H : ball (succ n) P) : ball n P  :=
λ x Hlt, H x (lt.step Hlt)

-- (∀ x : nat, x < n → P x) and (P n) implies (∀ x : nat, x < succ n → P x)
definition ball_succ_of_ball (H₁ : ball n P) (H₂ : P n) : ball (succ n) P :=
λ (x : nat) (Hlt : x < succ n), or.elim (eq_or_lt_of_le (le_of_lt_succ Hlt))
  (λ he  : x = n, eq.rec_on (eq.rec_on he rfl) H₂)
  (λ hlt : x < n, H₁ x hlt)

-- (¬ P n) implies ¬ (∀ x : nat, x < succ n → P x)
definition not_ball_of_not (H₁ : ¬ P n) : ¬ ball (succ n) P :=
λ (H : ball (succ n) P), absurd (H n (lt.base n)) H₁

-- ¬ (∀ x : nat, x < n → P x) implies ¬ (∀ x : nat, x < succ n → P x)
definition not_ball_succ_of_not_ball (H₁ : ¬ ball n P) : ¬ ball (succ n) P :=
λ (H : ball (succ n) P), absurd (ball_of_ball_succ H) H₁
```

Finally, assuming `P` is a decidable predicate, we prove `∀ x : nat, x < n → P x` by induction on `n`.

```
definition dec_ball [instance] (H : decidable_pred P) : Π (n : nat), decidable (ball n P)
| dec_ball 0     := inl (ball_zero P)
```

```
| dec_ball (a+1) :=
  match dec_ball a with
  | inl iH  :=
      match H a with
      | inl Pa  := inl (ball_succ_of_ball iH Pa)
      | inr nPa := inr (not_ball_of_not nPa)
      end
  | inr niH := inr (not_ball_succ_of_not_ball niH)
  end
```

Now we can use `dec_trivial` to prove simple theorems by "evaluation."

```
example : ∀ x : nat, x ≤ 4 → x ≠ 6 :=
dec_trivial

example : ¬ ∀ x : nat, x ≤ 5 → ∀ y, y < x → y * y ≠ x :=
dec_trivial
```

We can also use the bounded quantifier to define a computable function. In this example, the expression `is_constant_range f n` returns `tt` if and only if the function `f` has the same value for every `i` such that `0 ≤ i < n`.

```
open bool
definition is_constant_range (f : nat → nat) (n : nat) : bool :=
if ∀ i, i < n → f i = f 0 then tt else ff

example : is_constant_range (λ i, zero) 10 = tt :=
rfl
```

As an exercise, we encourage you to show that `∃ x : nat, x < n ∧ P x` is also decidable.

```
import data.nat
open nat decidable algebra

definition bex (n : nat) (P : nat → Prop) : Prop :=
∃ x : nat, x < n ∧ P x

definition not_bex_zero (P : nat → Prop) : ¬ bex 0 P :=
sorry

variables {n : nat} {P : nat → Prop}

definition bex_succ (H : bex n P) : bex (succ n) P :=
sorry

definition bex_succ_of_pred  (H : P n) : bex (succ n) P :=
sorry

definition not_bex_succ (H₁ : ¬ bex n P) (H₂ : ¬ P n) : ¬ bex (succ n) P :=
sorry
```

```
definition dec_bex [instance] (H : decidable_pred P) : Π (n : nat), decidable (bex n P) :=
sorry
```

10

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.

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> : Type :=
  <constructor> :: <fields>
```

Most parts are optional. Here is an example:

```
structure point (A : Type) :=
mk :: (x : A) (y : A)
```

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 recursor
check point.induction_on -- then recursor to Prop
check point.destruct     -- an alias for point.rec_on
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`.

```
eval point.x (point.mk (10 : ℕ) 20)
eval point.y (point.mk (10 : ℕ) 20)

open point

example (A : Type) (a b : A) : x (mk a b) = a :=
rfl

example (A : Type) (a b : A) : y (mk a b) = b :=
rfl
```

If the constructor is not provided, then a constructor is named `mk` by default.

```
structure prod (A : Type) (B : Type) :=
(pr1 : A) (pr2 : B)

check prod.mk
```

The keyword `record` is an alias for `structure`.

```
record point (A : Type) :=
mk :: (x : A) (y : A)
```

You can provide universe levels explicitly. The annotations in the next example force the parameters `A` and `B` to be types from the same universe, and set the return type to also be in the same universe.

```
structure prod.{u} (A : Type.{u}) (B : Type.{u}) : Type.{max 1 u} :=
(pr1 : A) (pr2 : B)

set_option pp.universes true
check prod.mk
```

The `set_option` command above instructs Lean to display the universe levels.

We use `max 1 l` as the resultant universe level to ensure the universe level is never `0` even when the parameter `A` and `B` are propositions. Recall that in Lean, `Type.{0}` is `Prop`, which is impredicative and proof irrelevant.

Objects

We have been using constructors to create elements of a structure (or record) 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-type> (, <field-name> := <expr>)* |}
or
⦃ <structure-type> (, <field-name> := <expr>)* ⦄
```

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 (A : Type) :=
mk :: (x : A) (y : A)

check {| point, x := (10 : ℕ), y := 20 |}   -- point ℕ
check {| point, y := (20 : ℕ), x := 10 |}
check ⦃ point, x := (10 : ℕ), y := 20 ⦄

example : {| point, x := (10 : ℕ), y := 20 |} = {| point, y := 20, x := 10 |} :=
rfl
```

Note that `point` is a parametric type, but we did not provide its parameters. Here, in each case, Lean infers that we are constructing an object of type `point` ℕ from the fact that one of the components is specified to be of type ℕ. Of course, the parameters can be explicitly provided with the type if needed.

```
check {⦃ point ℕ, x := 10, y := 20 ⦄
```

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

check {| my_struct, a := 10, b := true |}
```

The notation for defining record objects can also be used in pattern-matching expressions.

```
open nat

structure big :=
(field1 : nat) (field2 : nat)
(field3 : nat) (field4 : nat)
(field5 : nat) (field6 : nat)

definition b : big := big.mk 1 2 3 4 5 6

definition v3 : nat :=
  match b with
   {| big, field3 := v |} := v
  end

example : v3 = 3 := rfl
```

Record update is another common operation. It consists in creating a new record object by modifying the value of one or more fields. Lean provides a variation of the notation described above for record updates.

```
{| <structure-type> (, <field-name> := <expr>)* (, <record-obj>)* |}
or
⦃ <structure-type> (, <field-name> := <expr>)* (, <record-obj>)* ⦄
```

The semantics is simple: record objects `<record-obj>` provide the values for the unspecified fields. 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.

```
open nat

structure point (A : Type) :=
mk :: (x : A) (y : A)

definition p1 : point nat := {| point, x := 10, y := 20 |}
definition p2 : point nat := {| point, x := 1, p1 |}
definition p3 : point nat := {| point, y := 1, p1 |}

example : point.y p1 = point.y p2 :=
rfl

example : point.x p1 = point.x p3 :=
rfl
```

Inheritance

We can *extend* existing structures by adding new fields. This feature allow us to simulate a form of *inheritance*.

```
structure point (A : Type) :=
mk :: (x : A) (y : A)

inductive color :=
red | green | blue

structure color_point (A : Type) extends point A :=
mk :: (c : color)
```

The type `color_point` inherits all the fields from `point` and declares a new one `c : color`. Lean automatically generates a coercion from `color_point` to `point`, so that a `color_point` can be provided wherever a `point` is expected.

```
definition x_plus_y (p : point num) :=
point.x p + point.y p

definition green_point : color_point num :=
{| color_point, x := 10, y := 20, c := color.green |}

eval x_plus_y green_point    -- 30

-- display implicit coercions
set_option pp.coercions true

check x_plus_y green_point    -- num

example : green_point = point.mk 10 20 :=
rfl

check color_point.to_point    -- color_point ?A → point ?A
```

The coercions are named `to_<parent structure>`. Lean always defines functions that map the child structure to its parents, but we can ask Lean not to mark these functions as coercions by using the `private` keyword.

```
structure color_point (A : Type) extends private point A :=
mk :: (c : color)

variable f : point ℕ → bool

check f (color_point.to_point (@color_point.mk ℕ 1 2 color.red))
```

For private parent structures, we have to use the coercions explicitly. If we remove `color_point.to_point` from the above `check` command, we get a type error.

We can "rename" fields inherited from parent structures using the `renaming` clause.

```
structure prod (A : Type) (B : Type) :=
pair :: (pr1 : A) (pr2 : B)

-- Rename fields pr1 and pr2 to x and y respectively.
structure point3 (A : Type) extends prod A A renaming pr1→x pr2→y :=
mk :: (z : A)

check point3.x
check point3.y
check point3.z

example : point3.mk (10 : ℕ) 20 30 = prod.pair 10 20 :=
rfl
```

In the next example, we define a structure using multiple inheritance, and then define an object using objects of the parent structures.

```
import data.nat
open nat

structure point (A : Type) :=
(x : A) (y : A) (z : A)

structure rgb_val :=
(red : nat) (green : nat) (blue : nat)

structure red_green_point (A : Type) extends point A, rgb_val :=
(no_blue : blue = 0)

definition p : point nat := {| point, x := 10, y := 10, z := 20 |}
definition r : rgb_val := {| rgb_val, red := 200, green := 50, blue := 0 |}
definition rgp : red_green_point nat := {| red_green_point, p, r, no_blue := rfl |}

example : point.x rgp = 10 := rfl
example : rgb_val.red rgp = 200 := rfl
```

Structures as Classes

Any structure can be tagged as a *class*. This makes it a suitable target for the class-instance resolution procedures that were described in the previous chapter. Declaring a structure as a class also has the effect that the structure argument in each projection is tagged as an implicit argument to be inferred by type class resolution.

For example, in the definition of the `has_mul` structure below, the projection `has_mul.mul` has an implicit argument `[s : has_mul A]`. This means that when we write `has_mul.mul a b` with `a b : A`, type class resolution will search for a suitable instance of `has_mul A`, a multiplication structure associated with `A`. As a result, we can define the binary notation `a * b`, leaving the structure implicit.

```
namespace hide

structure has_mul [class] (A : Type) :=
mk :: (mul : A → A → A)

check @has_mul.mul    -- Π {A : Type} [c : has_mul A], A → A → A

infixl `*`   := has_mul.mul

section
  variables (A : Type) (s : has_mul A) (a b : A)
  check a * b
end

end hide
```

In the last `check` command, the structure `s` in the local context is used to synthesize the implicit argument in `a * b`.

When a structure is marked as a class, the functions mapping a child structure to its parents are also marked as instances unless the `private` modifier is used. As a result, whenever an instance of the parent structure is required, and instance of the child structure can be provided. In the following example, we use this mechanism to "reuse" the notation defined for the parent structure, `has_mul`, with the child structure, `semigroup`.

```
namespace hide

structure has_mul [class] (A : Type) :=
mk :: (mul : A → A → A)

infixl `*`   := has_mul.mul

structure semigroup [class] (A : Type) extends has_mul A :=
mk :: (assoc : ∀ a b c, mul (mul a b) c = mul a (mul b c))

section
  variables (A : Type) (s : semigroup A) (a b : A)
```

```
  check a * b
end

end hide
```

Once again, the structure `s` in the local context is used to synthesize the implicit argument in `a * b`. We can see what is going by asking Lean to display implicit arguments, coercions, and disable notation.

```
section
  variables (A : Type) (s : semigroup A) (a b : A)

  set_option pp.implicit true
  set_option pp.notation false

  check a * b -- @has_mul.mul A (@semigroup.to_has_mul A s) a b : A
end
```

Here is a fragment of the algebraic hierarchy defined using this mechanism. In Lean, you can also inherit from multiple structures. Moreover, fields with the same name are merged. If the types do not match an error is generated. The "merge" can be avoided by using the `renaming` clause.

```
namespace hide

structure has_mul [class] (A : Type) :=
mk :: (mul : A → A → A)

structure has_one [class] (A : Type) :=
mk :: (one : A)

structure has_inv [class] (A : Type) :=
mk :: (inv : A → A)

infixl `*`   := has_mul.mul
postfix `⁻¹` := has_inv.inv
notation 1   := has_one.one

structure semigroup [class] (A : Type) extends has_mul A :=
mk :: (assoc : ∀ a b c, mul (mul a b) c = mul a (mul b c))

structure comm_semigroup [class] (A : Type) extends semigroup A :=
mk :: (comm : ∀ a b, mul a b = mul b a)

structure monoid [class] (A : Type) extends semigroup A, has_one A :=
mk :: (right_id : ∀ a, mul a one = a) (left_id : ∀ a, mul one a = a)

structure comm_monoid [class] (A : Type) extends monoid A, comm_semigroup A

print prefix comm_monoid

end hide
```

Notice that we can suppress `:=` and `::` when we are not declaring any new fields, as is the case for the structure `comm_monoid`. The `print prefix` command shows that the common fields of `monoid` and `comm_semigroup` have been merged.

The `renaming` clause allow us to perform non-trivial merge operations such as combining an abelian group with a monoid to obtain a ring.

```
structure group [class] (A : Type) extends monoid A, has_inv A :=
(is_inv : ∀ a, mul a (inv a) = one)

structure abelian_group [class] (A : Type) extends group A renaming mul→add, comm_monoid A

structure ring [class] (A : Type)
  extends abelian_group A renaming
    assoc→add.assoc
    comm→add.comm
    one→zero
    right_id→add.right_id
    left_id→add.left_id
    inv→uminus
    is_inv→uminus_is_inv,
  monoid A renaming
    assoc→mul.assoc
    right_id→mul.right_id
    left_id→mul.left_id
:=
(dist_left  : ∀ a b c, mul a (add b c) = add (mul a b) (mul a c))
(dist_right : ∀ a b c, mul (add a b) c = add (mul a c) (mul b c))
```

11

Tactic-Style Proofs

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. One important difference is that term-style proofs are elaborated globally, and information gathered from one part of a term can be used to fill in implicit information in another part of the term. In contrast, tactics apply locally, and are narrowly focused on a single subgoal in the proof.

Entering the Tactic Mode

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

```
theorem test (p q : Prop) (Hp : p) (Hq : q) : p ∧ q ∧ p :=
sorry
```

We can write this goal as follows:

```
p : Prop, q : Prop, Hp : p, Hq : q ⊢ p ∧ q ∧ p
```

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

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

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

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

```
p : Prop,
q : Prop,
Hp : p,
Hq : q
⊢ p

⊢ q ∧ p
```

For brevity, Lean only displays the context for the first goal, which is the one addressed by the next tactic command. The first goal is met with the command `exact Hp`. The `exact` command is just a variant of `apply` which signals that the expression given should fill the goal exactly. It is good form to use it in a tactic proof, since its failure signals that something has gone wrong; but otherwise `apply` would work just as well.

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

```
reveal test
print test
```

You can write a tactic script incrementally. If you run Lean on an incomplete tactic proof bracketed by `begin` and `end`, the system reports all the unsolved goals that remain. If you are running Lean with its Emacs interface, you can see this information by putting your cursor on the `end` symbol, which should be underlined. In the Emacs interface, there is another extremely useful trick: if you put your cursor on a line of a tactic proof and press "C-c C-g", Lean will show you the goal that remains at the end of the line.

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

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

Unsurprisingly, it produces exactly the same proof term.

```
reveal test
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
```

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

```
theorem test (p q : Prop) (Hp : p) (Hq : q) : p ∧ q ∧ p :=
by apply (and.intro Hp); exact (and.intro Hq Hp)
```

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

Basic Tactics

In addition to `apply` and `exact`, another useful tactic is `intro`, which introduces a hypothesis. What follows is an example of an identity from propositional logic that we proved in Section Examples of Propositional Validities, but now prove using tactics. We adopt the following convention regarding indentation: whenever a tactic introduces one or more additional subgoals, we indent another two spaces, until the additional subgoals are deleted.

```
example (p q r : Prop) : p ∧ (q ∨ r) ↔ (p ∧ q) ∨ (p ∧ r) :=
begin
  apply iff.intro,
    intro H,
    apply (or.elim (and.elim_right H)),
      intro Hq,
      apply or.intro_left,
      apply and.intro,
        exact (and.elim_left H),
      exact Hq,
    intro Hr,
    apply or.intro_right,
    apply and.intro,
    exact (and.elim_left H),
    exact Hr,
  intro H,
  apply (or.elim H),
    intro Hpq,
    apply and.intro,
      exact (and.elim_left Hpq),
    apply or.intro_left,
    exact (and.elim_right Hpq),
  intro Hpr,
  apply and.intro,
    exact (and.elim_left Hpr),
  apply or.intro_right,
  exact (and.elim_right Hpr)
end
```

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

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

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

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

```
example : ∀ a b c : nat, a = b → a = c → c = b :=
begin
  intros [a, b, c, H1, H2],
  exact eq.trans (eq.symm H2) H1
end
```

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

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

```
example (H1 : x = y) (H2 : y = z) (H3 : z = w) : x = w :=
begin
  apply (eq.trans H1),
  apply (eq.trans H2),
  assumption   -- applied H3
end
```

It will unify metavariables in the conclusion if necessary:

```
example (H1 : x = y) (H2 : y = z) (H3 : z = w) : x = w :=
begin
  apply eq.trans,
  assumption,     -- solves x = ?b with H1
  apply eq.trans,
  assumption,     -- solves ?b = w with H2
  assumption      -- solves z = w with H3
end
```

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

```
example : ∀ a b c : nat, a = b → a = c → c = b :=
begin
  intros,
  apply eq.trans,
  apply eq.symm,
  assumption,
  assumption
end
```

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

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

There is variant of `apply` called `fapply` that is more aggressive in creating new subgoals for arguments. Here is an example of how it is used:

```
import data.nat
open nat

example : ∃ a : ℕ, a = a :=
begin
  fapply exists.intro,
  exact nat.zero,
  apply rfl
end
```

The command `fapply exists.intro` creates two goals. The first is to provide a natural number, `a`, and the second is to prove that `a = a`. Notice that the second goal depends on the first; solving the first goal instantiates a metavariable in the second.

Notice also that we could not write `exact 0` in the proof above, because `0` is a numeral that is coerced to a natural number. In the context of a tactic proof, expressions are elaborated "locally," before being sent to the tactic command. When the tactic command is being processed, Lean does not have enough information to determine that `0` needs to be coerced. We can get around that by stating the type explicitly:

```
example : ∃ a : ℕ, a = a :=
begin
  fapply exists.intro,
  exact (0 : ℕ),
  apply rfl
end
```

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

```
import data.nat
open nat

variables x y z : ℕ

example : x = x :=
begin
  generalize x, -- goal is x : ℕ ⊢ ∀ (x : ℕ), x = x
  intro y,      -- goal is x y : ℕ ⊢ y = y
  apply rfl
end

example (H : x = y) : y = x :=
begin
  generalize H, -- goal is x y : ℕ, H : x = y ⊢ y = x
  intro H1,     -- goal is x y : ℕ, H H1 : x = y ⊢ y = x
  apply (eq.symm H1)
end
```

In the first example above, the `generalize` tactic generalizes the conclusion over the variable `x`, turning the goal into a ∀. In the second, it generalizes the goal over the hypothesis H, putting the antecedent explicitly into the goal. We generalize any term, not just variables:

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

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

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

The `revert` tactic is a combination of `generalize` and `clear`:

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

example (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
  apply (eq.symm H1)
end
```

Like `intro`, the tactics `generalize`, `clear`, and `revert` have plural forms. For example, we could have written above:

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

Structuring Tactic Proofs

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`, `obtains`, and so on. Conversely, when writing an arbitrary Lean term, you can always invoke the tactic mode by inserting a `begin...end` block. In the next example, we use `show` within a tactic block to fulfill a goal by providing an explicit term.

```
example (p q r : Prop) : p ∧ (q ∨ r) ↔ (p ∧ q) ∨ (p ∧ r) :=
begin
  apply iff.intro,
    intro H,
    apply (or.elim (and.elim_right H)),
      intro Hq,
      show (p ∧ q) ∨ (p ∧ r),
        from or.inl (and.intro (and.elim_left H) Hq),
    intro Hr,
    show (p ∧ q) ∨ (p ∧ r),
      from or.inr (and.intro (and.elim_left H) Hr),
  intro H,
  apply (or.elim H),
    intro Hpq,
    show p ∧ (q ∨ r), from
      and.intro
        (and.elim_left Hpq)
        (or.inl (and.elim_right Hpq)),
  intro Hpr,
  show p ∧ (q ∨ r), from
    and.intro
      (and.elim_left Hpr)
      (or.inr (and.elim_right Hpr))
end
```

You can also 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,
    apply (or.elim (and.elim_right H)),
      intro Hq,
      show (p ∧ q) ∨ (p ∧ r),
        from or.inl (and.intro (and.elim_left H) Hq),
    intro Hr,
    show (p ∧ q) ∨ (p ∧ r),
      from or.inr (and.intro (and.elim_left H) Hr),
  end,
```

```
  begin
    intro H,
    apply (or.elim H),
    begin
      intro Hpq,
      show p ∧ (q ∨ r), from
        and.intro
          (and.elim_left Hpq)
          (or.inl (and.elim_right Hpq)),
    end,
    begin
      intro Hpr,
      show p ∧ (q ∨ r), from
        and.intro
          (and.elim_left Hpr)
          (or.inr (and.elim_right Hpr))
    end
  end
end
```

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,
    apply (or.elim (and.elim_right H)),
    { intro Hq,
      apply or.intro_left,
      apply and.intro,
      { exact (and.elim_left H) },
      { exact Hq }},
    { intro Hr,
      apply or.intro_right,
      apply and.intro,
      { exact (and.elim_left H)},
      { exact Hr }}},
  { intro H,
    apply (or.elim H),
    { intro Hpq,
      apply and.intro,
      { exact (and.elim_left Hpq) },
      { apply or.intro_left,
        exact (and.elim_right Hpq) }},
    { intro Hpr,
      apply and.intro,
      { exact (and.elim_left Hpr)},
      { apply or.intro_right,
          exact (and.elim_right Hpr) }}}
end
```

Here we have adopted the convention that whenever a tactic increases the number of goals to be solved, the tactics that solve each subsequent goal are enclosed in braces. This may not increase readability much, but it does help clarify the structure of the proof.

There is a `have` construct for tactic-style proofs that is similar to the one for term-style proofs. In the proof below, the first `have` creates the subgoal `Hp : p`. The `from` clause solves it, and after that `Hp` is available to subsequent tactics. The example illustrates that you can also use another `begin...end` block, or a `by` clause, to prove a subgoal introduced by `have`.

```
variables p q : Prop

example : p ∧ q ↔ q ∧ p :=
begin
  apply iff.intro,
  begin
    intro H,
    have Hp : p, from and.left H,
    have Hq : q, from and.right H,
    apply and.intro,
    repeat assumption
  end,
  begin
    intro H,
    have Hp : p,
      begin
        apply and.right,
        apply H
      end,
    have Hq : q, by apply and.left; exact H,
    apply (and.intro Hp Hq)
  end
end
```

Cases and Pattern Matching

The `cases` tactic works on elements of an inductively defined type. It does what the name suggests: it decomposes an element of an inductive type according to each of the possible constructors, and leaves a goal for each case. Note that the following example also uses the `revert` tactic to move the hypothesis into the conclusion of the goal.

```
import data.nat
open nat

example (x : ℕ) (H : x ≠ 0) : succ (pred x) = x :=
begin
  revert H,
  cases x,
  -- first goal: ⊢ 0 ≠ 0 → succ (pred 0) = 0
  { intro H1,
    apply (absurd rfl H1)},
  -- second goal: ⊢ succ a ≠ 0 → succ (pred (succ a)) = succ a
  { intro H1,
```

```
    apply rfl}
end
```

The name of the cases tactic is particularly well suited to use with disjunctions:

```
example (a b : Prop) : a ∨ b → b ∨ a :=
begin
  intro H,
  cases H with [Ha, Hb],
  { exact or.inr Ha },
  { exact or.inl Hb }
end
```

In the next example, we rely on the decidability of equality for the natural numbers to carry out another proof by cases:

```
import data.nat
open nat

check nat.sub_self

example (m n : nat) : m - n = 0 ∨ m ≠ n :=
begin
  cases (decidable.em (m = n)) with [Heq, Hne],
  { apply eq.subst Heq,
    exact or.inl (nat.sub_self m)},
  { apply or.inr Hne }
end
```

The cases tactic can also be used to extract the arguments of a constructor, even for an inductive type like and, for which there is only one constructor.

```
example (p q : Prop) : p ∧ q → q ∧ p :=
begin
  intro H,
  cases H with [H1, H2],
  apply and.intro,
  exact H2,
  exact H1
end
```

Here the with clause names the two arguments to the constructor. If you omit it, Lean will choose a name for you. If there are multiple constructors with arguments, you can provide cases with a list of all the names, arranged sequentially:

```
import data.nat
open nat

inductive foo : Type :=
| bar1 : ℕ → ℕ → foo
```

```
| bar2 : ℕ → ℕ → ℕ → foo

definition silly (x : foo) : ℕ :=
begin
  cases x with [a, b, c, d, e],
  exact b,    -- a, b, c are in the context
  exact e     -- d, e    are in the context
end
```

You can also use pattern matching in a tactic block. With

```
example (p q r : Prop) : p ∧ q ↔ q ∧ p :=
begin
  apply iff.intro,
  { intro H,
    match H with
    |  and.intro H₁ H₂ := by apply and.intro; repeat assumption
    end },
  { intro H,
    match H with
    | and.intro H₁ H₂ := by apply and.intro; repeat assumption
    end },
end
```

With pattern matching, the first and third examples in this section could be written as follows:

```
example (x : ℕ) (H : x ≠ 0) : succ (pred x) = x :=
begin
  revert H,
  match x with
  | 0      := by intro H1; exact (absurd rfl H1)
  | succ y := by intro H1; apply rfl
  end
end

definition silly (x : foo) : ℕ :=
begin
  match x with
  | foo.bar1 a b   := b
  | foo.bar2 c d e := e
  end
end
```

The Rewrite Tactic

The `rewrite` tactic provide a basic mechanism for applying substitutions to goals and hypotheses, providing a convenient and efficient way of working with equality. This tactic is loosely based on the rewrite tactic available in the proof language SSReflect.

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

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

example (H₁ : f 0 = 0) (H₂ : k = 0) : f k = 0 :=
begin
  rewrite H₂, -- replace k with 0
  rewrite H₁  -- replace f 0 with 0
end
```

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

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

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

example (H₁ : f 0 = 0) (H₂ : k = 0) : f k = 0 :=
begin
  rewrite [H₂, H₁]
end
```

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

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

example (H₁ : a = b) (H₂ : f a = 0) : f b = 0 :=
begin
  rewrite [-H₁, H₂]
end
```

In this example, the term `-H`$_1$ instructs the `rewriter` to replace `b` with `a`.

The notation `*t` instructs the rewriter to apply the rewrite `t` zero or more times, while the notation `+t` instructs the rewriter to use it at least once. Note that rewriting with `*t` never fails.

```
import data.nat
open nat algebra
```

```
example (x y : nat) : (x + y) * (x + y) = x * x + y * x + x * y + y * y :=
by rewrite [*left_distrib, *right_distrib, -add.assoc]
```

To avoid non-termination, the `rewriter` tactic has a limit on the maximum number of iterations performed by rewriting steps of the form `*t` and `+t`. For example, without this limit, the tactic `rewrite *add.comm` would make Lean diverge on any goal that contains a sub-term of the form `t + s` since commutativity would be always applicable. The limit can be modified by setting the option `rewriter.max_iter`.

The notation `rewrite n t`, where `n`, is a positive number indicates that `t` must be applied exactly `n` times. Similarly, `rewrite n>t` is notation for at most `n` times.

A pattern `p` can be optionally provided to a rewriting step `t` using the notation `{p}t` . It allows us to specify where the rewrite should be applied. This feature is particularly useful for rewrite rules such as commutativity `a + b = b + a` which may be applied to many different sub-terms. A pattern may contain placeholders. In the following example, the pattern `b + _` instructs the `rewrite` tactic to apply commutativity to the first term that matches `b + _`, where `_` can be matched with an arbitrary term.

```
example (a b c : nat) : a + b + c = a + c + b :=
begin
  rewrite [add.assoc, {b + _}add.comm, -add.assoc]
end
```

In the example above, the first step rewrites `a + b + c` to `a + (b + c)`. Then, `{b + _}add.comm` applies commutativity to the term `b + c`. Without the pattern `{b + _}`, the tactic would instead rewrite `a + (b + c)` to `(b + c) + a`. Finally, `-add.assoc` applies associativity in the “reverse direction” rewriting `a + (c + b)` to `a + c + b`.

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

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

example (H : a + 0 = 0) : f a = f 0 :=
begin
  rewrite [add_zero at H, H]
end
```

The first step, `add_zero at H`, rewrites the hypothesis `(H : a + 0 = 0)` to `a = 0`. Then the new hypothesis `(H : a = 0)` is used to rewrite the goal to `f 0 = f 0`.

Multiple hypotheses can be specified in the same `at` clause.

```
variables (a b : nat)

example (H₁ : a + 0 = 0) (H₂ : b + 0 = 0) : a + b = 0 :=
begin
```

```
  rewrite add_zero at (H₁, H₂),
  rewrite [H₁, H₂]
end
```

You may also use `t at *` to indicate that all hypotheses and the goal should be rewritten using `t`. The tactic step fails if none of them can be rewritten. The notation `t at * ⊢` applies `t` to all hypotheses. You can enter the symbol ⊢ by typing `\|-`.

```
variables (a b : nat)

example (H₁ : a + 0 = 0) (H₂ : b + 0 = 0) : a + b + 0 = 0 :=
begin
  rewrite add_zero at *,
  rewrite [H₁, H₂]
end
```

The step `add_zero at *` rewrites the hypotheses H_1, H_2 and the main goal using the `add_zero (x : nat) : x + 0 = x`, producing `a = 0`, `b = 0` and `a + b = 0` respectively.

The `rewrite` tactic is not restricted to propositions. In the following example, we use `rewrite H at v` to rewrite the hypothesis `v : vector A n` to `v : vector A 0`.

```
import data.examples.vector
open nat

variables {A : Type} {n : nat}
example (H : n = 0) (v : vector A n) : vector A 0 :=
begin
  rewrite H at v,
  exact v
end
```

Given a rewrite `(t : l = r)`, the tactic `rewrite t` by default locates a sub-term `s` which matches the left-hand-side `l`, and then replaces all occurrences of `s` with the corresponding right-hand-side. The notation `at {i_1, ..., i_k}` can be used to restrict which occurrences of the sub-term `s` are replaced. For example, `rewrite t at {1, 3}` specifies that only the first and third occurrences should be replaced.

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

example (H₁ : a = b) (H₂ : f b a b = 0) : f a a a = 0 :=
by rewrite [H₁ at {1, 3}, H₂]
```

Similarly, `rewrite t at H {1, 3}` specifies that `t` must be applied to hypothesis `H` and only the first and third occurrences must be replaced. You can also specify which occurrences should not be replaced using the notation `rewrite t at -{i_1, ..., i_k}`. Here is the previous example using this feature.

```
example (H1 : a = b) (H2 : f b a b = 0) : f a a a = 0 :=
by rewrite [H1 at -{2}, H2]
```

So far, we have used theorems and hypotheses as rewriting rules. In these cases, the term `t` is just an identifier. The notation `rewrite (t)` can be used to provide an arbitrary term `t` as a rewriting rule.

```
open algebra

variables {A : Type} [s : group A]
include s

theorem inv_eq_of_mul_eq_one {a b : A} (H : a * b = 1) : a⁻¹ = b :=
by rewrite [-(mul_one a⁻¹), -H, inv_mul_cancel_left]
```

In the example above, the term `mul_one a⁻¹` has type `a⁻¹ * 1 = a⁻¹`. Thus, the rewrite step `-(mul_one a⁻¹)` replaces `a⁻¹` with `a⁻¹ * 1`.

Calculational proofs and the rewrite tactic can be used together.

```
example (a b c : nat) (H1 : a = b) (H2 : b = c + 1) : a ≠ 0 :=
calc
  a     = succ c : by rewrite [H1, H2, add_one]
    ... ≠ 0      : succ_ne_zero c
```

The `rewrite` tactic also supports reduction steps: `↑f`, `▶*`, `↓t`, and `▶t`. The step `↑f` unfolds `f` and performs beta/iota reduction and simplify projections. This step fails if there is no `f` to be unfolded. The step `▶*` is similar to `↑f`, but does not take a constant to unfold as argument, therefore it never fails. The fold step `↓t` unfolds the head symbol of `t`, then search for the result in the goal (or a given hypothesis), and replaces any match with `t`. Finally, `▶t` tries to reduce the goal (or a given hypothesis) to `t`, and fails if it is not convertible to `t`. (The up arrow is entered with `\u`, the down arrow is entered with `\d`, and the right triangle is entered with `\t`. You can also use the ASCII alternatives `^f`, `>*`, `<d t`, and `> t` for `↑f`, `▶*`, `↓t`, and `▶t`, respectively.)

```
definition double (x : nat) := x + x

variable f : nat → nat

example (x y : nat) (H1 : double x = 0) (H3 : f 0 = 0) : f (x + x) = 0 :=
by rewrite [↑double at H1, H1, H3]
```

The step `↑double at H1` unfolds `double` in the hypothesis `H1`. The notation `rewrite ↑[f_1, ..., f_n]` is shorthand for `rewrite [↑f_1, ..., ↑f_n]`

The tactic `esimp` is a shorthand for `rewrite ▶*`. Here are two simple examples:

```
open sigma nat

example (x y : nat) (H : (fun (a : nat), pr1 ⟨a, y⟩) x = 0) : x = 0 :=
begin
  esimp at H,
  exact H
end

example (x y : nat) (H : x = 0) : (fun (a : nat), pr1 ⟨a, y⟩) x = 0 :=
begin
  esimp,
  exact H
end
```

Here is an example where the fold step is used to replace `a + 1` with `f a` in the main goal.

```
open nat

definition foo [irreducible] (x : nat) := x + 1

example (a b : nat) (H : foo a = b) : a + 1 = b :=
begin
  rewrite ↓foo a,
  exact H
end
```

Here is another example: given any type `A`, we show that the `list A` append operation `s ++ t` is associative.

```
import data.list
open list
variable {A : Type}

theorem append_assoc : ∀ (s t u : list A), s ++ t ++ u = s ++ (t ++ u)
| append_assoc nil t u      := by apply rfl
| append_assoc (a :: l) t u :=
  begin
    rewrite ▶ a :: (l ++ t ++ u) = _,
    rewrite append_assoc
  end
```

We discharge the inductive cases using the `rewrite` tactic. The base case is solved by applying reflexivity, because `nil ++ t ++ u` and `nil ++ (t ++ u)` are definitionally equal. In the inductive step, we first reduce the goal `a :: s ++ t ++ u = a :: s ++ (t ++ u)` to `a :: (s ++ t ++ u) = a :: s ++ (t ++ u)` by applying the reduction step `▶ a :: (l ++ t ++ u) = _`. The idea is to expose the term `l ++ t ++ u`, which can be rewritten using the inductive hypothesis `append_assoc (s t u : list A) : s ++ t ++ u = s ++ (t ++ u)`. Notice that we used a placeholder `_` in the right-hand-side of this reduction step; this placeholder is unified with the right-hand-side of the main goal. As a result, we do not have the copy the right-hand side of the goal.

The rewrite tactic supports type classes. In the following example we use theorems from the mul_zero_class and add_monoid classes in an example for the comm_ring class. The rewrite is acceptable because every comm_ring (commutative ring) is an instance of the classes mul_zero_class and add_monoid.

```
import algebra.ring
open algebra

example {A : Type} [s : comm_ring A] (a b c : A) : a * 0 + 0 * b + c * 0 + 0 * a = 0 :=
begin
  rewrite [+mul_zero, +zero_mul, +add_zero]
end
```

There are two variants of rewrite, namely krewrite and xrewrite, that are more aggressive about matching patterns. krewrite will unfold definitions as long as the head symbol matches, for example, when trying to match a pattern f p with an expression f t. In contrast, xrewrite will unfold all definitions that are not marked irreducible. Both are computationally expensive and should be used sparingly. krewrite is often useful when matching patterns requires unfolding projections in an algebraic structure.

12

Axioms and Computation

We have seen that the version of the Calculus of Inductive Constructions that has been implemented in Lean includes Pi types, and inductive types, and a nested hierarchy of universes with an impredicative, proof-irrelevant `Prop` at the bottom. In this chapter, we consider extensions of 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, denoting a natural number will reduce to a numeral.

To support classical reasoning, Lean's standard library defines one choice axiom, which is justified on a set-theoretic interpretation of type theory. In the standard library, the law of the excluded middle is a consequence of this axiom. The library also imports two semi-constructive (or mildly classical) axioms, propositional extensionality and quotients. These are used, for example, to develop theories of sets and finite sets. Even some computationally inclined users may also wish to use the law of the excluded middle to reason about computation. Below we will describe the effects that these axioms have on computation aspects of the system.

However, the classical choice axiom (also known as the Hilbert operator) is entirely inimical to a computational interpretation of the system, which 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 depend on this construction lose their computational content, and in Lean we are required to mark such definitions as `noncomputable` to flag that fact.

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 Pi 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 the 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 : A` for any type `A`, 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, ignore intermediate typing constraints, and reduces terms until they reach a normal form. Current plans for Lean include the development of a fast evaluator along these lines.

Having adopted a proof-irrelevant `Prop`, one might consider it legitimate to use, for example, the law of the excluded middle, governing propositions. Of course, this, too, can block computation, but it does not block fast evaluation as described above. From a constructive point of view, the most objectionable classical axioms are "choice axioms" that allow us to extract "data" from any existential proposition, completely erasing the distinction between the proof-irrelevant and data-relevant parts of the theory. These are discussed in Section Choice Axioms below.

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 the effect that equivalent propositions can be substituted for one another in any context:

```
section
  open eq.ops
  variables a b c d e : Prop
  variable P : Prop → Prop

  example (H : a ↔ b) : (c ∧ a ∧ d → e) ↔ (c ∧ b ∧ d → e) :=
  propext H ▶ !iff.refl

  example (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.

Function Extensionality

Similar to propositional extensionality, function extensionality asserts that any two functions of type `Π x : A, B x` that agree on all their inputs are equal.

```
check @funext
-- ∀ {A : Type} {B : A → Type} {f₁ f₂ : Π x : A, B x}, (∀ x, 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 `X : Type` we define the `set X := X → Prop` to denote the type of subsets of `X`, essentially identifying subsets with predicates. By combining `funext` and `propext`, we obtain an extensional theory of such sets:

```
definition set (X : Type) := X → Prop

namespace set

variable {X : Type}

definition mem [reducible] (x : X) (a : set X) := a x
notation e ∈ a := mem e a

theorem setext {a b : set X} (H : ∀ x, x ∈ a ↔ x ∈ b) : a = b :=
funext (take x, propext (H x))

end set
```

We can then proceed to define the empty set and set intersection, for example, and prove set identities:

```
definition empty [reducible] : set X := λ x, false
notation `∅` := empty

definition inter [reducible] (a b : set X) : set X := λ x, x ∈ a ∧ x ∈ b
notation a ∩ b := inter a b

theorem inter_self (a : set X) : a ∩ a = a :=
```

```
setext (take x, !and_self)

theorem inter_empty (a : set X) : a ∩ ∅ = ∅ :=
setext (take x, !and_false)

theorem empty_inter (a : set X) : ∅ ∩ a = ∅ :=
setext (take x, !false_and)

theorem inter.comm (a b : set X) : a ∩ b = b ∩ a :=
setext (take x, !and.comm)
```

The following is an example of how function extensionality blocks computation inside the Lean kernel.

```
import data.nat
open nat algebra

definition f₁ (x : ℕ) := x
definition f₂ (x : ℕ) := 0 + x

theorem feq : f₁ = f₂ := funext (take x, eq.subst !zero_add rfl)
check eq.rec (0 : ℕ) feq    -- ℕ
eval eq.rec (0 : ℕ) feq     -- eq.rec 0 feq
```

First, we show that the two functions f_1 and f_2 are equal using function extensionality, and then we "cast" `0` of type ℕ by replacing f_1 by f_2 in the type. Of course, the cast is vacuous, because ℕ does not depend on f_1. 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.

In the next section, we will exhibit a similar example with the quotient construction. Current research programs, including work on *observational type theory* and *cubical type theory*, aim to extend type theory in ways that permit reductions for casts involving function extensionality, quotients, and more. But the solutions are not so clear cut, and the rules of Lean's underlying calculus do not sanction such reductions.

In a sense, however, a cast does not change the "meaning" of an expression. Rather, it is a mechanism to reason about the expression's type. Given an appropriate semantics, it then makes sense to reduce terms in ways that preserve their meaning, ignoring the intermediate bookkeeping needed to make the reductions type correct. In that case, adding new axioms in `Prop` does not matter; by proof irrelevance, an expression in `Prop` carries no information, and can be safely ignored by the reduction procedures.

Quotients

Let A be any type, and let R be an equivalence relation on A. It is mathematically common to form the "quotient" A / R, that is, the type of elements of A "modulo" R. Set theoretically, one can view A / R as the set of equivalence classes of A modulo R. If f : A → B is any function that respects the equivalence relation in the sense that for every x y : A, R x y implies f x = f y, then f "lifts" to a function f' : A / R → B defined on each equivalence class [x] by f' [x] = f x. Lean's standard library extends the Calculus of Inductive Constructions with additional constants that perform exactly these constructions, and installs this last equation as a definitional reduction rule.

First, it is useful to define the notion of a *setoid*, which is simply a type with an associated equivalence relation:

```
structure setoid [class] (A : Type) :=
(r : A → A → Prop) (iseqv : equivalence r)

namespace setoid
  infix `≈` := setoid.r

  variable {A : Type}
  variable [s : setoid A]
  include s

  theorem refl (a : A) : a ≈ a :=
  and.elim_left (@setoid.iseqv A s) a

  theorem symm {a b : A} : a ≈ b → b ≈ a :=
  λ H, and.elim_left (and.elim_right (@setoid.iseqv A s)) a b H

  theorem trans {a b c : A} : a ≈ b → b ≈ c → a ≈ c :=
  λ H₁ H₂, and.elim_right (and.elim_right (@setoid.iseqv A s)) a b c H₁ H₂
end setoid
```

Given a type A, a relation R on A, and a proof p that R is an equivalence relation, we can define setoid.mk p as an instance of the setoid class. Lean's type class inference mechanism then allows us to use the generic notation ≈ for R, and to use the generic theorems setoid.refl, setoid.symm, setoid.trans to reason about R.

The quotient package consists of the following constructors:

```
open setoid
constant quot.{l}   : Π {A : Type.{l}}, setoid A → Type.{l}

namespace quot
  constant mk        : Π {A : Type}   [s : setoid A], A → quot s
  notation `⟦`:max a `⟧`:0 := mk a

  constant sound     : Π {A : Type}   [s : setoid A] {a b : A}, a ≈ b → ⟦a⟧ = ⟦b⟧
  constant lift      : Π {A B : Type} [s : setoid A] (f : A → B), (∀ a b, a ≈ b → f a = f b) → quot s → B
  constant ind       : ∀ {A : Type}   [s : setoid A] {B : quot s → Prop}, (∀ a, B ⟦a⟧) → ∀ q, B q
end quot
```

For any type `A` with associated equivalence relation `R`, first we declare a setoid instance `s` to associate `R` as "the" equivalence relation on `A`. Once we do that, `quot s` denotes the quotient type `A / R`, and given `a : A`, `⟦a⟧` denotes the "equivalence class" of `a`. The meaning of constants `sound`, `lift`, and `ind` are given by their types. In particular, `lift` is the function which lifts a function `f : A → B` that respects the equivalence relation to the function `lift f : quot s → B` which lifts `f` to `A / R`. After declaring the constants associated with the quotient type, the library file then calls an internal function, `init_quotient`, which installs the reduction that simplifies `lift f ⟦a⟧` to `f a`.

To illustrate the use of quotients, let us define the type of ordered pairs. In the standard library, `A × B` represents the Cartesian product of the types `A` and `B`. We can view it as the type of pairs `(a, b)` where `a : A` and `b : B`. We can use quotient types to define the type of unordered pairs of type `A`. We can use the notation `{a₁, a₂}` to represent the unordered pair containing `a₁` and `a₂`. Moreover, we want to be able to prove the equality `{a₁, a₂} = {a₂, a₁}`. We start this construction by defining a relation `p ~ q` on `A × A`.

```
import data.prod
open prod prod.ops quot

private definition eqv {A : Type} (p₁ p₂ : A × A) : Prop :=
(p₁.1 = p₂.1 ∧ p₁.2 = p₂.2) ∨ (p₁.1 = p₂.2 ∧ p₁.2 = p₂.1)

infix `~` := eqv
```

To make the proofs more compact, we open the namespaces `eq` and `or`. Thus, we can write `symm`, `trans`, `inl` and `inr` instead of `eq.symm`, `eq.trans`, `or.inl` and `or.inr` respectively. We also define the notation `⟨H₁, H₂⟩` for `(and.intro H₁ H₂)`.

```
open eq or

local notation `⟨` H₁ `,` H₂ `⟩` := and.intro H₁ H₂
```

The next step is to prove that `eqv` is an equivalence relation, which is to say, it is reflexive, symmetric and transitive. We can prove these three facts in a convenient and readable way by using dependent pattern matching to perform case-analysis and break the hypotheses into pieces that are then reassembled to produce the conclusion.

```
private theorem eqv.refl {A : Type} : ∀ p : A × A, p ~ p :=
take p, inl ⟨rfl, rfl⟩

private theorem eqv.symm {A : Type} : ∀ p₁ p₂ : A × A, 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 {A : Type} : ∀ p₁ p₂ p₃ : A × A, 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⟩
| (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 (A : Type) : equivalence (@eqv A) :=
mk_equivalence (@eqv A) (@eqv.refl A) (@eqv.symm A) (@eqv.trans A)
```

Now that we have proved that `eqv` is an equivalence relation, we can construct a `setoid (A × A)`, and use it to define the type `uprod A` of unordered pairs. Moreover, we define the unordered pair `{a₁, a₂}` as `⟦(a₁, a₂)⟧`.

```
definition uprod.setoid [instance] (A : Type) : setoid (A × A) :=
setoid.mk (@eqv A) (is_equivalence A)

definition uprod (A : Type) : Type :=
quot (uprod.setoid A)

namespace uprod
  definition mk {A : Type} (a₁ a₂ : A) : uprod A :=
  ⟦(a₁, a₂)⟧

  notation `{` a₁ `,` a₂ `}` := mk a₁ a₂
end uprod
```

Now, we can easily prove that `{a₁, a₂} = {a₂, a₁}` using the `quot.sound` since `(a₁, a₂) ~ (a₂, a₁)`.

```
  theorem mk_eq_mk {A : Type} (a₁ a₂ : A) : {a₁, a₂} = {a₂, a₁} :=
  quot.sound (inr ⟨rfl, rfl⟩)
```

To complete the example, given `a : A` and `u : uprod A`, 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`, in the lemma `mem_respects`. This is an idiom that is used extensively in the Lean standard library.

```
  private definition mem_fn {A : Type} (a : A) : A × A → Prop
  | (a₁, a₂) := a = a₁ ∨ a = a₂

  -- auxiliary lemma for proving mem_respects
  private lemma mem_swap {A : Type} {a : A} : ∀ {p : A × A}, mem_fn a p = mem_fn a (swap p)
```

```
| (a₁, a₂) := propext (iff.intro
    (λ l : a = a₁ ∨ a = a₂, 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 {A : Type} : ∀ {p₁ p₂ : A × A} (a : A),  p₁ ~ p₂ → mem_fn a p₁ = mem_fn a p₂
| (a₁, a₂) (b₁, b₂) a (inl ⟨a₁b₁, a₂b₂⟩) :=
  begin esimp at a₁b₁, esimp at a₂b₂, rewrite [a₁b₁, a₂b₂] end
| (a₁, a₂) (b₁, b₂) a (inr ⟨a₁b₂, a₂b₁⟩) :=
  begin esimp at a₁b₂, esimp at a₂b₁, rewrite [a₁b₂, a₂b₁], apply mem_swap end

definition mem {A : Type} (a : A) (u : uprod A) : Prop :=
quot.lift_on u (λ p, mem_fn a p) (λ p₁ p₂ e, mem_respects a e)

infix `∈` := mem

theorem mem_mk_left {A : Type} (a b : A) : a ∈ {a, b} :=
inl rfl

theorem mem_mk_right {A : Type} (a b : A) : b ∈ {a, b} :=
inr rfl

theorem mem_or_mem_of_mem_mk {A : Type} {a b c : A} : c ∈ {a, b} → c = a ∨ c = b :=
λ h, h
```

The quotient construction can be used to derive function extensionality, and we have seen that the latter blocks computation. The following provides another example of the same phenomenon, similar to the one we discussed in the last section.

```
import data.finset
open finset quot list nat

definition s₁ : finset nat := to_finset [1, 2]
definition s₂ : finset nat := to_finset [2, 1]

theorem seq : s₁ = s₂ := dec_trivial
check eq.rec (0 : ℕ) seq
eval eq.rec (0 : ℕ) seq
```

Choice Axioms

The following axiom is used to support classical reasoning in Lean:

```
axiom strong_indefinite_description {A : Type} (P : A → Prop) (H : nonempty A) :
  { x | (∃ y : A, P y) → P x}
```

This asserts that given any predicate `P` on a nonempty type `A`, we can (magically) produce an element `x` with the property that if any element of `A` satisfies `P`, then `x` does. In the presence of classical logic, we could prove this from the slightly weaker axiom:

```
axiom indefinite_description {A : Type} {P : A → Prop} (H : ∃ x, P x) :
  {x : A | P x}
```

This says that knowing that there is an element of A satisfying P is enough to produce one. This axiom essentially undoes the separation of data from propositions, because it allows us to extract a piece of data — an element of A satisfying P — from the proposition that such an element exists.

The axiom `strong_indefinite_description` is imported when you import `logic.choice`. Separating the x asserted to exist by the axiom from the property it satisfies allows us to define the Hilbert epsilon function:

```
noncomputable definition epsilon {A : Type} [H : nonempty A] (P : A → Prop) : A :=
let u : {x | (∃ y, P y) → P x} :=
  strong_indefinite_description P H in
elt_of u

theorem epsilon_spec_aux {A : Type} (H : nonempty A) (P : A → Prop) (Hex : ∃ y, P y) :
    P (@epsilon A H P) :=
let u : {x | (∃ y, P y) → P x} :=
  strong_indefinite_description P H in
has_property u Hex

theorem epsilon_spec {A : Type} {P : A → Prop} (Hex : ∃ y, P y) :
    P (@epsilon A (nonempty_of_exists Hex) P) :=
epsilon_spec_aux (nonempty_of_exists Hex) P Hex
```

Assuming the type A is nonempty, `epsilon P` returns an element of A, with the property that if any element of A satisfies P, `epsilon P` does. Notice that the definition is preceded by the keyword `noncomputable`, to signal the fact that expressions depending on this definition will not compute to canonical normal forms, even under the more liberal evaluation scheme described above.

Just as `indefinite_description` is a weaker version of `strong_indefinite_description`, the `some` operator is a weaker version of the `epsilon` operator. It is sometimes easier to use. Assuming `H : ∃ x, P x` is a proof that some element of A satisfies P, `some H` denotes such an element.

```
noncomputable definition some {A : Type} {P : A → Prop} (H : ∃ x, P x) : A :=
@epsilon A (nonempty_of_exists H) P

theorem some_spec {A : Type} {P : A → Prop} (H : ∃ x, P x) : P (some H) :=
epsilon_spec H
```

Excluded Middle

The law of the excluded middle is the following

```
check @em
-- ∀ (a : Prop), a ∨ ¬a
```

We can prove it using the choice axiom described in the previous section. This is a consequence of Diaconescu's theorem which 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 `strong_indefinite_description` (Hilbert's choice), `propext` (propositional extensionality) and `funext` (function extensionality). The standard library contains this proof, which we reproduce here.

First, we import the necessary axioms, fix a parameter, `p`, and define two predicates `U` and `V`:

```
import logic.eq
open classical eq.ops

section
parameter  p : Prop

definition U (x : Prop) : Prop := x = true ∨ p
definition V (x : Prop) : Prop := x = false ∨ p
```

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 `epsilon` to choose an element from each of `U` and `V`:

```
noncomputable definition u := epsilon U
noncomputable definition v := epsilon V

lemma u_def : U u :=
epsilon_spec (exists.intro true (or.inl rfl))

lemma v_def : V v :=
epsilon_spec (exists.intro false (or.inl rfl))
```

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 Hvf⁻¹ ▸ Hut⁻¹ ▸ 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 (take x : Prop,
    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 H' : epsilon U = epsilon V, from Hpred ▶ rfl,
show u = v, from H'
```

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 Classical Logic.

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))))
```

Propositional Decidability

Taken together, the law of the excluded middle and the axiom of indefinite description imply that every proposition is decidable. The following is the contained in `logic.choice`:

```
noncomputable definition decidable_inhabited [instance] (a : Prop) : inhabited (decidable a) :=
inhabited_of_nonempty
  (or.elim (em a)
    (assume Ha, nonempty.intro (inl Ha))
    (assume Hna, nonempty.intro (inr Hna)))

noncomputable definition prop_decidable [instance] (a : Prop) : decidable a :=
arbitrary (decidable a)
```

The definition decidable_inhabited uses the law of the excluded middle to show that decidable a is inhabited for any a. It is marked as an instance, and is silently used for for synthesizing the implicit argument in arbitrary (decidable a).

As an example, we use some to prove that if f : A → B is injective and A is inhabited, then f has a left inverse. To define the left inverse linv, we use the "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, the strong_indefinite_description 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. From a classical point of view, linv is a function. From a constructive point of view, it is unacceptable; since there is no way to implement such a function in general, the construction is not informative.

```
open classical function

noncomputable definition linv {A B : Type} [h : inhabited A] (f : A → B) : B → A :=
λ b : B, if ex : (∃ a : A, f a = b) then some ex else arbitrary A

theorem has_left_inverse_of_injective {A B : Type} {f : A → B}
        : inhabited A → injective f → ∃ g, g ∘ f = id :=
assume h   : inhabited A,
assume inj : ∀ a₁ a₂, f a₁ = f a₂ → a₁ = a₂,
have is_linv  : (linv f) ∘ f = id, from
  funext (λ a,
    assert ex  : ∃ a₁ : A, f a₁ = f a,   from exists.intro a rfl,
    have   feq : f (some ex) = f a,      from !some_spec,
    calc linv f (f a) = some ex   :  dif_pos ex
                ...    = a         :  inj _ _ feq),
exists.intro (linv f) is_linv
```

Constructive Choice

In the standard library, we say a type A is encodable if there are functions f : A → nat and g : nat → option A such that for all a : A, g (f a) = some a. Here is the precise definition:

```
structure encodable [class] (A : Type) :=
(encode : A → nat) (decode : nat → option A) (encodek : ∀ a, decode (encode a) = some a)
```

The standard library shows that indefinite_description axiom is actually a theorem for any encodable type A and decidable predicate p : A → Prop. It provides the following definition and theorem, which are concrete realizations of some and some_spec, respectively.

```
check @choose
-- choose : Π {A : Type} {p : A → Prop} [c : encodable A] [d : decidable_pred p], (∃ (x : A), p x) → A
```

```
check @choose_spec
-- choose_spec : ∀ {A : Type} {p : A → Prop} [c : encodable A] [d : decidable_pred p] (ex : ∃ (x : A), p x), p (choose ex)
```

The construction is straightforward: it finds `a : A` satisfying `p` by enumerating the elements of `A` and testing whether they satisfy `p` or not. We can show that this search always terminates because we have the assumption `∃ (x : A), p x`.

We can use this to provide a constructive version of the theorem `has_left_inverse_of_injective`. We remark this is not the only possible version. The constructive version contains more hypotheses than the classical version. In Bishop's terminology, it avoids "pseudo-generality." Considering the classical construction, it is clear that once we have `choose`, we can construct the left inverse as long as we can decide whether `b` is in the image of a function `f : A → B`.

```
import data.encodable
open encodable function

section
  parameters {A B : Type}
  parameter  (f : A → B)
  parameter  [inhA : inhabited A]
  parameter  [dex  : ∀ b, decidable (∃ a, f a = b)]
  parameter  [encA : encodable A]
  parameter  [deqB : decidable_eq B]
  include inhA dex encA deqB

  definition finv : B → A :=
  λ b : B, if ex : (∃ a, f a = b) then choose ex else arbitrary A

  theorem has_left_inverse_of_injective : injective f → has_left_inverse f :=
  assume inj : ∀ a₁ a₂, f a₁ = f a₂ → a₁ = a₂,
  have is_linv : ∀ a, finv (f a) = a, from
    (take a,
      assert ex  : ∃ a₁, f a₁ = f a, from exists.intro a rfl,
      have   feq : f (choose ex) = f a, from !choose_spec,
      calc finv (f a) = choose ex :  dif_pos ex
                 ...    = a         :  inj _ _ feq),
  exists.intro finv is_linv
end
```

The argument is essentially the same as the classical one; we have simply replaced the classical `some` with the constructive choice function `choose`, and added three extra hypotheses: `dex`, `encA` and `deqB`. The first one makes sure we can decide whether a value `b` is in the image of `f` or not, and the last two are needed to use `choose`.

The standard library contains many `encodable` types and shows that many types have decidable equality. The hypothesis `dex` can be satisfied in many cases. For example, it is trivially satisfied if `f` is surjective. It is also satisfied whenever `A` is finite.

```
section
  parameters {A B : Type} (f : A → B)

  definition decidable_in_image_of_surjective : surjective f → ∀ b, decidable (∃ a, f a = b) :=
  assume s : surjective f, take b,
  decidable.inl (s b)

  definition decidable_in_image_of_fintype_of_deceq [instance]
             [finA : fintype A] [deqB : decidable_eq B] : ∀ b, decidable (∃ a, f a = b) :=
  take b, decidable_exists_finite
end
```

Tracking used axioms

The Lean standard library contains only 3 axioms: `quot.sound`, `propext` and `strong_indefinite_description`. Most of the library depends only on the first two. The command `print axioms` displays all axioms that have been asserted/imported into the current logical context. Similarly, the command `print axioms decl_name` prints all axioms the declaration `decl_name` depends on.

IMPORTANT: in the Lean web version, we erase the proof of most theorems. The idea is to reduce the size of the file that must be downloaded to run Lean on your web browser. So, the result of the `print axioms` commands is not precise on the web version. Please use the Lean native application if you are interested in using these commands.

```
print axioms
print axioms nat.add
print axioms finset.union
print axioms set.empty_union
print axioms classical.some
```

13

More Tactics

We have seen that tactics provide a powerful language for describing and constructing proofs. Care is required: a proof that is a long string of tactic applications can be very hard to read and maintain. But when combined with the various structuring mechanisms that Lean's proof language has to offer, they provide efficient means for filling in the details of a proof. The goal of this chapter is to add some additional tactics to your repertoire.

[This chapter is still under construction.]

Induction

Just as the `cases` tactic performs proof by cases on an element of an inductively defined type, the `induction` tactic performs a proof by induction. As with the `cases` tactic, the `with` clause allows you to name the variables and hypotheses that are introduced. Also as with the `cases` tactic, the `induction` tactic will revert any hypotheses that depend on the induction variable and then reintroduce them for you automatically. The following examples prove the commutativity of addition on the natural numbers, using only the defining equations for addition (in particular, the property `add_succ`, which asserts that `x + succ y = succ (x + y)` for every `x` and `y`).

```
open nat

theorem zero_add (x : ℕ) : 0 + x = x :=
begin
  induction x with x ih,
    {exact rfl},
  rewrite [add_succ, ih]
end
```

```
theorem succ_add (x y : ℕ) : succ x + y = succ (x + y) :=
begin
  induction y with y ih,
    {exact rfl},
  rewrite [add_succ, ih]
end

theorem add.comm (x y : ℕ) : x + y = y + x :=
begin
  induction x with x ih,
    {show 0 + y = y + 0, by rewrite zero_add},
  show succ x + y = y + succ x,
    begin
      induction y with y ihy,
        {krewrite zero_add},
      rewrite [succ_add, ih]
    end
end
```

(For the use of `krewrite` here, see the end of Chapter Tactic-Style Proofs.)

The induction tactic can be used not only with the induction principles that are created automatically when an inductive type is defined, but also induction principles that prove on their own. For example, recall that the standard library defines the type `finset A` of finite sets of elements of any type `A`. Typically, we assume `A` has decidable equality, which means in particular that we can decide whether an element `a : A` is a member of a finite set `s`. Clearly, a property `P` holds for an arbitrary finite set when it holds for the empty set and when it is maintained for a finite set `s` after a new element `a`, that was not previously in `s`, is added to `s`. This is encapsulated by the following principle of induction:

```
theorem finset.induction {A : Type} [h : decidable_eq A] {P : finset A → Prop}
  (H₁ : P finset.empty)
  (H₂ : ∀ ⦃a : A⦄ {s : finset A}, a ∉ s → P s → P (insert a s))
  : (∀ s, P s)
```

To use this as an induction principle, one has to mark it with the attribute `[recursor 6]`, which tells the induction tactic that this is a user defined induction principle in which induction is carried out on the sixth argument. This is done in the standard library. Then, when induction is carried out on an element of `finset`, the induction tactic finds the relevant principle.

```
import data.finset data.nat
open finset nat

variables (A : Type) [deceqA : decidable_eq A]
include deceqA

theorem card_add_card (s₁ s₂ : finset A) : card s₁ + card s₂ = card (s₁ ∪ s₂) + card (s₁ ∩ s₂) :=
begin
  induction s₂ with a s₂ hs₂ ih,
```

```
    show card s₁ + card (∅:finset A) = card (s₁ ∪ ∅) + card (s₁ ∩ ∅),
      by rewrite [union_empty, card_empty, inter_empty],
    show card s₁ + card (insert a s₂) = card (s₁ ∪ (insert a s₂)) + card (s₁ ∩ (insert a s₂)),
      from sorry
end
```

The proof is carried out by induction on s_2. According to the with clause, the inductive step concerns the set insert a s_2 in place of s_2, hs_2 denotes the assuption $a \notin s_2$, and ih denotes the inductive hypothesis. (The full proof can be found in the library.) If necessary, we can specify the induction principle manually:

```
theorem card_add_card (s₁ s₂ : finset A) : card s₁ + card s₂ = card (s₁ ∪ s₂) + card (s₁ ∩ s₂) :=
begin
  induction s₂ using finset.induction with a s₂ hs₂ ih,
    show card s₁ + card (∅:finset A) = card (s₁ ∪ ∅) + card (s₁ ∩ ∅),
      by rewrite [union_empty, card_empty, inter_empty],
    show card s₁ + card (insert a s₂) = card (s₁ ∪ (insert a s₂)) + card (s₁ ∩ (insert a s₂)),
      from sorry
end
```

Other Tactics

The tactic subst substitutes a variable defined in the context, and clears both the variable and the hypothesis. The tactic substvars substitutes all the variables in the context.

```
import data.nat
open nat

variables a b c d : ℕ

example (Ha : a = b + c) : c + a = c + (b + c) :=
by subst a

example (Ha : a = b + c) (Hd : d = b) : a + d = b + c + d :=
by subst [a, d]

example (Ha : a = b + c) (Hd : d = b) : a + d = b + c + d :=
by substvars

example (Ha : a = b + c) (Hd : b = d) : a + d = d + c + d :=
by substvars

example (Hd : b = d) (Ha : a = b + c) : a + d = d + c + d :=
by substvars
```

A number of tactics are designed to help construct elements of inductive types. For example constructor <i> constructs an element of an inductive type by applying the ith constructor; constructor alone applies the first constructor that succeeds. The tactic `split` can only be applied to inductive types with only one constructor, and is then equivalent to `constructor 1`. Similarly, `left` and `right` are designed for use with inductive types with two constructors, and are then equivalent to `constructor 1` and `constructor 2`, respectively. Here are prototypical examples:

```
variables p q : Prop

example (Hp : p) (Hq : q) : p ∧ q :=
by split; exact Hp; exact Hq

example (Hp : p) (Hq : q) : p ∧ q :=
by split; repeat assumption

example (Hp : p) : p ∨ q :=
by constructor; assumption

example (Hq : q) : p ∨ q :=
by constructor; assumption

example (Hp : p) : p ∨ q :=
by constructor 1; assumption

example (Hq : q) : p ∨ q :=
by constructor 2; assumption

example (Hp : p) : p ∨ q :=
by left; assumption

example (Hq : q) : p ∨ q :=
by right; assumption
```

The tactic `existsi` is similar to `constructor 1`, but it allows us to provide an argument, as is commonly done with when introducing an element of an `exists` or `sigma` type.

```
import data.nat
open nat

example : ∃ x : ℕ, x > 2 :=
by existsi 3; exact dec_trivial

example (B : ℕ → Type) (b : B 2) : Σ x : ℕ, B x :=
by existsi 2; assumption
```

The `injection` tactic makes use of the fact that constructors to an inductive type are injective:

```
import data.nat
open nat

example (x y : ℕ) (H : succ x = succ y) : x = y :=
by injection H with H'; exact H'
```

```
example (x y : ℕ) (H : succ x = succ y) : x = y :=
by injection H; assumption
```

The first version gives the name the consequence of applying injectivity to the hypothesis `H`. The second version lets Lean choose the name.

The tactics `reflexivity`, `symmetry`, and `transitivity` work not just for equality, but also for any relation with a corresponding theorem marked with the attribute `refl`, `symm`, or `trans`, respectively. Here is an example of their use:

```
variables (A : Type) (a b c d : A)

example (H₁ : a = b) (H₂ : c = b) (H₃ : c = d) : a = d :=
by transitivity b; assumption; transitivity c; symmetry; assumption; assumption
```

The `contradiction` tactic closes a goal when contradictory hypotheses have been derived:

```
variables p q : Prop

example (Hp : p) (Hnp : ¬ p) : q :=
by contradiction
```

Similarly, `exfalso` and `trivial` implement "ex falso quodlibet" and the introduction rule for `true`, respectively.

Combinators

Combinators are used to combine tactics. The most basic one is the `and_then` combinator, written with a semicolon (`;`), which applies tactics successively.

The `par` combinator, written with a vertical bar (`|`), tries one tactic and then the other, using the first one that succeeds. The `repeat` tactic applies a tactic repeatedly. Here is an example of these in use:

```
example (a b c d : Prop) : a ∧ b ∧ c ∧ d ↔ d ∧ c ∧ b ∧ a :=
begin
  apply iff.intro,
  repeat (intro H; repeat (cases H with [H', H] | apply and.intro | assumption))
end
```

Here is another one:

```
import data.set
open set function eq.ops

variables {X Y Z : Type}
```

```
lemma image_comp (f : Y → X) (g : X → Y) (a : set X) : (f ∘ g) ' a = f ' (g ' a) :=
set.ext (take z,
  iff.intro
    (assume Hz,
      obtain x Hx₁ Hx₂, from Hz,
      by repeat (apply mem_image | assumption | reflexivity))
    (assume Hz,
      obtain y [x Hz₁ Hz₂] Hy₂, from Hz,
      by repeat (apply mem_image | assumption | esimp [comp] | rewrite Hz₂)))
```

Finally, some tactics can be used to "debug" a tactic proof by printing output to the screen when Lean is run from the command line. The command `trace` produces the given output, `state` shows the current goal, `now` fails if there are any current goals, and `check_expr t` displays the type of the expression in the context of the current goal.

```
open tactic

theorem tst {A B : Prop} (H1 : A) (H2 : B) : A :=
by (trace "first";  state; now  |
       trace "second"; state; fail |
       trace "third";  assumption)
```

Other tactics can be used to manipulate goals. For example, `rotate_left` or `rotate_right` followed by a number rotates through the goals. The tactic `rotate` is equivalent to `rotate_left`.

A

Quick Reference

Note that this quick reference guide describes Lean 2 only.

Displaying Information

```
check <expr>                 : check the type of an expression
eval <expr>                  : evaluate expression
print <id>                   : print information about <id>
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 or emacs mode
print prefix <namespace>     : display all declarations in the namespace
print coercions              : display all coercions
print coercions <source>     : display only the coercions from <source>
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
print metaclasses            : show kinds of metadata stored in a namespace
help commands                : display all available commands
help options                 : display all available options
```

Common Options

You can change an option by typing `set_option <option> <value>`. The `<option>` field supports TAB-completion. You can see an explanation of all options using `help options`.

```
pp.implicit        : display implicit arguments
pp.universes       : display universe variables
```

```
pp.coercions       : show coercions
pp.notation        : display output using defined notations
pp.abbreviations   : display output using defined abbreviations
pp.full_names      : use full names for identifiers
pp.all             : disable notations, implicit arguments, full names,
                     universe parameters and coercions
pp.beta            : beta reduce terms before displaying them
pp.max_depth       : maximum expression depth
pp.max_steps       : maximum steps for printing expression
pp.private_names   : show internal name assigned to private definitions and theorems
pp.metavar_args    : show arguments to metavariables
pp.numerals        : print output as numerals
```

Attributes

These can generally be declared with a `definition` or `theorem`, or using the `attribute` or `local attribute` commands.

Example: `local attribute nat.add nat.mul [reducible]`.

```
reducible       : unfold at any time during elaboration if necessary
quasireducible  : unfold during higher order unification,
                  but not during type class resolution
semireducible   : unfold when performance is not critical
irreducible     : avoid unfolding during elaboration
coercion        : use as a coercion between types
class           : type class declaration
instance        : type class instance
priority <num>  : add a priority to an instance or notation
parsing-only    : use notation only for input
unfold <num>    : if the argument at position <num> is marked with [constructor]
                  unfold this and that argument (for iota reduction)
constructor     : see unfold <num>
unfold-full     : unfold definition when fully applied
recursor        : user-defined recursor/eliminator, used for the induction tactic
recursor <num>  : user-defined non-dependent recursor/eliminator
                  where <num> is the position of the major premise
refl            : reflexivity lemma, used for calc-expressions, tactics and simplifier
symm            : symmetry lemma, used for calc-expressions, tactics and simplifier
trans           : transitivity lemma, used for calc-expressions, tactics and simplifier
subst           : substitution lemma, used for calc-expressions and simplifier
```

Proof Elements

Term Mode

```
take, assume     : syntactic sugar for lambda
let              : introduce local definitions
have             : introduce auxiliary fact (opaque, in the body)
assert           : like "have", but visible to tactics
show             : make result type explicit
```

```
suffices         : show that the goal follows from this fact
obtain ..., from : destruct structures such as exists, sigma, ...
match ... with   : introduce proof or definition by cases
proof ... qed    : introduce a proof or definition block, elaborated separately
```

The keywords `have` and `assert` can be anonymous, which is to say, they can be used without giving a label to the hypothesis. The corresponding element of the context can then be referred to using the keyword `this` until another anonymous element is introduced, or by enclosing the assertion in backticks. To avoid a syntactic ambiguity, the keyword `suppose` is used instead of `assume` to introduce an anonymous assumption.

One can also use anonymous binders (like `lambda`, `take`, `obtain`, etc.) by enclosing the type in backticks, as in λ `` `nat`, `nat` + 1``. This introduces a variable of the given type in the context with a hidden name.

Tactic Mode

At any point in a proof or definition you can switch to tactic mode and apply tactics to finish that part of the proof or definition.

```
begin ... end   : enter tactic mode, and blocking mechanism within tactic mode
{ ... }         : blocking mechanism within tactic mode
by ...          : enter tactic mode, can only execute a single tactic
begin+; by+     : same as =begin= and =by=, but make local results available
have            : as in term mode (enters term mode), but visible to tactics
show            : as in term mode (enters term mode)
match ... with  : as in term mode (enters term mode)
let             : introduce abbreviation (not visible in the context)
note            : introduce local fact (opaque, in the body)
```

Normally, entering tactic mode will make declarations in the local context given by "have"-expressions unavailable. The annotations `begin+` and `by+` make all these declarations available.

Sectioning Mechanisms

```
namespace <id> ... end <id>  : begin / end namespace
section ... end              : begin / end section
section <id> .... end <id>   : begin / end section

variable (var : type)        : introduce variable where needed
variable {var : type}        : introduce implicit variable where needed
variable {{var : type}}      : introduce implicit variable where needed,
                               which is not maximally inserted
variable [var : type]        : introduce class inference variable where needed
variable {var} (var) [var]   : change the bracket type of an existing variable
parameter                    : introduce variable, fixed within the section
```

```
include                       : include variable in subsequent definitions
omit                          : undo "include"
```

Tactics

We say a tactic is more "aggressive" when it uses a more expensive (and complete) unification algorithm, and/or unfolds more aggressively definitions.

General tactics

```
apply <expr>       : apply a theorem to the goal, create subgoals for non-dependent premises
fapply <expr>      : like apply, but create subgoals also for dependent premises that were
                     not assigned by unification procedure
eapply <expr>      : like apply, but used for applying recursor-like definitions
exact <expr>       : apply and close goal, or fail
rexact <expr>      : relaxed (and more expensive) version of exact
                     (this will fully elaborate <expr> before trying to match it to the goal)
refine <expr>      : like exact, but creates subgoals for unresolved subgoals

intro <ids>        : introduce multiple variables or hypotheses
intros <ids>       : same as intro <ids>
intro              : let Lean choose a name
intros             : introduce variables as long as the goal reduces to a function type
                     and let Lean choose the names

rename <id> <id>   : rename a variable or hypothesis
generalize <expr>  : generalize an expression
clear <ids>        : remove variables or hypotheses
revert <ids>       : move variables or hypotheses into the goal
assumption         : try to close a goal with something in the context
eassumption        : a more aggressive ("expensive") form of assumption
```

Equational reasoning

```
esimp                  : simplify expressions (by evaluation/normalization) in goal
esimp at <id>          : simplify hypothesis in context
esimp at *             : simplify everything
esimp [<ids>]          : unfold definitions and simplify expressions in goal
esimp [<ids>] at <id>  : unfold definitions and simplify hypothesis in context
esimp [<ids>] at *     : unfold definitions and simplify everything
unfold <id>            : similar to (esimp <id>)
fold <expr>            : unfolds <expr>, search for convertible term in the
                         goal, and replace it with <expr>

beta                   : beta reduce goal
whnf                   : put goal in weak head normal form
change <expr>          : change the goal to <expr> if it is convertible to <expr>

rewrite <rule>         : apply a rewrite rule (see below)
rewrite [<rules>]      : apply a sequence of rewrite rules (see below)
krewrite               : using keyed rewriting, matches any subterm
```

```
                      with the same head as the rewrite rule
xrewrite            : a more aggressive form of rewrite

subst <id>          : substitute a variable defined in the context, and clear hypothesis and
                      variable
substvars           : substitute all variables in the context
```

1. Rewrite rules

   You can combine rewrite rules from different groups in the following order, starting with the innermost:

```
e                : match left-hand-side of equation e to a goal subterm,
                   then replace every occurence with right-hand-side
{p}e             : apply e only where pattern p (which may contain placeholders) matches

n t              : apply t exactly n times
n>t              : apply t at most n times
*t               : apply t zero or more times (up to rewriter.max_iter)
+t               : apply t one or more times

-t               : apply t in reverse direction

↑id              : unfold id
↑[ids]           : unfold ids
↓id              : fold id
►expr            : reduce goal to expression expr
►*               : equivalent to esimp

t at {i, ...}    : apply t only at numbered occurences
t at -{i, ...}   : apply t only at all but the numbered occurences
t at H           : apply t at hypothesis H
t at H {i, ...}  : apply t only at numbered occurences in H
t at H -{i, ...} : apply t only at all but the numbered occurences in H
t at * ⊢         : apply t at all hypotheses
t at *           : apply t at the goal and all hypotheses
```

Induction and cases

```
cases <expr>                 : decompose an element of an inductive type
cases <expr> with <ids>      : name newly introduced variables as specified by <ids>
induction <expr> (with <ids>) : use induction
induction <expr> using <def> : use the definition <def> to apply induction
constructor                  : construct an element of an inductive type by applying the
                               first constructor that succeeds
constructor <i>              : construct an element of an inductive type by applying the
                               ith-constructor
fconstructor                 : construct an element of an inductive type by (fapply)ing the
                               first constructor that succeeds
fconstructor <i>             : construct an element of an inductive type by (fapply)ing the
                               ith-constructor
injection <id> (with <ids>)  : use injectivity of constructors at specified hypothesis
split                        : equivalent to (constructor 1), only applicable to inductive
```

```
                              datatypes with a single constructor (e.g. and introduction)
left                        : equivalent to (constructor 1), only applicable to inductive
                              datatypes with two constructors (e.g. left or introduction)
right                       : equivalent to (constructor 2), only applicable to inductive
                              datatypes with two constructors (e.g. right or introduction)
existsi <expr>              : similar to (constructor 1) but we can provide an argument,
                              useful for performing exists/sigma introduction
```

Special-purpose tactics

```
contradiction       : close contradictory goal
exfalso             : implements the "ex falso quodlibet" logical principle
congruence          : solve goals of the form (f a_1 ... a_n = f' b_1 ... b_n) by congruence
reflexivity         : reflexivity of equality (or any relation marked with attribute refl)
symmetry            : symmetry of equality (or any relation marked with attribute symm)
transitivity <expr> : transitivity of equality (or any relation marked with attribute trans)
trivial             : apply true introduction
```

Combinators

```
and_then <tac1> <tac2> (notation: <tac1> ; <tac2>)
                          : execute <tac1> and then execute <tac2>, backtracking when needed
                            (aka sequential composition)
or_else <tac1> <tac2> (notation: (<tac1> | <tac2>))
                          : execute <tac1> if it fails, execute <tac2>
<tac1>: <tac2>            : apply <tac1> and then apply <tac2> to all subgoals generated by <tac1>
par <tac1> <tac2>         : execute <tac1> and <tac2> in parallel
fixpoint (fun t, <tac>)   : fixpoint tactic, <tac> may refer to t
try <tac>                 : execute <tac>, if it fails do nothing
repeat <tac>              : repeat <tac> zero or more times (until it fails)
repeat1 <tac>             : like (repeat <tac>), but fails if <tac> does not succeed at least
                            once
at_most <num> <tac>       : like (repeat <tac>), but execute <tac> at most <num> times
do <num> <tac>            : execute <tac> exactly <num> times
determ <tac>              : discard all but the first proof state produced by <tac>
discard <tac> <num>       : discard the first <num> proof-states produced by <tac>
```

Goal management

```
focus_at <tac> <i>  : execute <tac> to the ith-goal, and fail if it is not solved
focus  <tac>        : equivalent to (focus_at <tac> 0)
rotate_left  <num>  : rotate goals to the left <num> times
rorate_right <num>  : rotate goals to the right <num> times
rotate <num>        : equivalent to (rotate_left <num>)
all_goals <tac>     : execute <tac> to all goals in the current proof state
fail                : tactic that always fails
id                  : tactic that does nothing and always succeeds
now                 : fail if there are unsolved goals
```

Information and debugging

```
state                          : display the current proof state
check_expr <expr>              : display the type of the given expression in the current goal
trace <string>                 : display the current string
with_options [<options>] <tac> : execute a single tactic with different options
                                 (<options> is a comma-separated list)
```

Emacs Lean-mode commands

Flycheck commands

```
C-c ! n    : next error
C-c ! p    : previous error
C-c ! l    : list errors
C-c C-x    : execute Lean (in stand-alone mode)
```

Lean-specific commands

```
C-c C-k    : show how to enter unicode symbol
C-c C-o    : set Lean options
C-c C-e    : execute Lean command
C-c C-r    : restart Lean process
C-c C-p    : print the definition of the identifier under the cursor
             in a new buffer
C-c C-g    : show the current goal at a line of a tactic proof, in a
             new buffer
C-c C-f    : fill a placeholder by the printed term in the minibuffer.
             Note: the elaborator might need more information
             to correctly infer the implicit arguments of this term
```

Unicode Symbols

This section lists some of the Unicode symbols that are used in the Lean library, their ASCII equivalents, and the keystrokes that can be used to enter them in the Emacs Lean mode.

Logical symbols

Unicode	Ascii	Emacs
true		
false		
¬	not	`\not`, `\neg`
∧	/\	`\and`
∨	\/	`\or`
→	->	`\to`, `\r`, `\implies`
↔	<->	`\iff`, `\lr`
∀	forall	`\all`
∃	exists	`\ex`
λ	fun	`\l`, `\fun`
≠	~=	`\ne`

Types

Π	Pi	`\Pi`
→	->	`\to`, `\r`, `\implies`
Σ	Sigma	`\S`, `\Sigma`
×	prod	`\times`
	sum	`\union`, `\u+`, `\uplus`
ℕ	nat	`\nat`
ℤ	int	`\int`
ℚ	rat	`\rat`
ℝ	real	`\real`

When you open the namespaces `prod` and `sum`, you can use `*` and `+` for the types `prod` and `sum` respectively. To avoid overwriting notation, these have to have the same precedence as the arithmetic operations. If you don't need to use notation for the arithmetic operations, you can obtain lower-precedence versions by opening the namespaces `low_precedence_times` and `low_precedence_plus` respectively.

Greek letters

Unicode	Emacs
α	`\alpha`
β	`\beta`
γ	`\gamma`
...	...

Equality proofs (`open eq.ops`)

Unicode	Ascii	Emacs
$^{-1}$	eq.symm	`\sy`, `\inv`, `\-1`
·	eq.trans	`\tr`
▶	eq.subst	`\t`

Symbols for the rewrite tactic

Unicode	Ascii	Emacs
↑	^	`\u`
↓	<d	`\d`

Brackets

Unicode	Ascii	Emacs
⌞t⌟	?(t)	`\cll t \clr`
⦃ t ⦄	{{t}}	`\{{ t \}}`
⟨ t ⟩		`\< t \>`
t		`\<< t \>>`

Set theory

Unicode	Ascii	Emacs
∈	mem	`\in`
∉		`\nin`
∩	inter	`\i`
∪	union	`\un`
⊆	subseteq	`\subeq`

Binary relations

Unicode	Ascii	Emacs
≤	<=	`\le`
≥	>=	`\ge`
∣	dvd	`|`
≡		`\equiv`
≈		`\eq`

Binary operations

Unicode	Ascii	Emacs
∘	comp	`\comp`

Bibliography

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

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

[3] 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.

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