Mathematics in Lean

Release 0.1

Jeremy Avigad
Kevin Buzzard
Robert Y. Lewis
Patrick Massot

Jun 12, 2022

CONTENTS

CHAPTER

ONE

INTRODUCTION

1.1 Getting Started

The goal of this book is to teach you to formalize mathematics using the Lean 3 interactive proof assistant. It assumes that you know some mathematics, but it does not require much. Although we will cover examples ranging from number theory to measure theory and analysis, we will focus on elementary aspects of those fields, in the hopes that if they are not familiar to you, you can pick thhem up as you go. We also don't presuppose any background in formalization. Formalization can be seen as a kind of computer programming: we will write mathematical definitions, theorems, and proofs in a regimented language, like a programming language, that Lean can understand. In return, Lean provides feedback and information, interprets expressions and guarantees that they are well-formed, and ultimately certifies the correctness of our proofs.

You can learn more about Lean from the Lean project page and the Lean community web pages. This tutorial is based on Lean's large and ever-growing library, *mathlib*. We also strongly recommend taking a look at the Lean Zulip online chat group if you haven't already. You'll find a lively and welcoming community of Lean enthusiasts there, happy to answer questions and offer moral support.

Although you can read a pdf or html version of this book online, it designed to be read interactively, running Lean from inside the VS Code editor. To get started:

1. Install Lean, VS Code, and mathlib following the instructions on the community web site.
2. In a terminal, type `leanproject get mathematics_in_lean` to set up a working directory for this tutorial.
3. Type `code mathematics_in_lean` to open that directory in `VS Code`.

Opening any Lean file will simultaneously open this book in a VS Code window. You can update to a newer version by tying `git pull` followed by `leanproject get-mathlib-cache` inside the `mathematics_in_lean` folder.

Alternatively, you can run Lean and VS Code in the cloud, using Gitpod. You can find instructions as to how to do that on the Mathematics in Lean project page on Github.

Each section in this book has an associated Lean file with examples and exercises. You can find them in the folder *src*, organized by chapter. We recommend making a copy of that folder so that you can experiment with the files as you go, while leaving the originals intact. The text will often include examples, like this one:

```
#eval "Hello, World!"
```

You should be able to find the corresponding example in the associated Lean file. If you click on the line, VS Code will show you Lean's feedback in the `Lean Goal` window, and if you hover your cursor over the `#eval` command VS Code will show you Lean's response to this command in a pop-up window. You are encouraged to edit the file and try examples of your own.

This book moreover provides lots of challenging exercises for you to try. Don't rush past these! Lean is about *doing* mathematics interactively, not just reading about it. Working through the exercises is central to the experience. You can always compare your solutions to the ones in the `solutions` folder associated with each section.

1.2 Overview

Put simply, Lean is a tool for building complex expressions in a formal language known as *dependent type theory*.

Every expression has a *type*, and you can use the *#check* command to print it. Some expressions have types like $\mathbb{N}$ or $\mathbb{N} \to \mathbb{N}$. These are mathematical objects.

```
#check 2 + 2

def f (x : ℕ) := x + 3

#check f
```

Some expressions have type *Prop*. These are mathematical statements.

```
#check 2 + 2 = 4

def fermat_last_theorem :=
  ∀ x y z n : ℕ, n > 2 ∧ x * y * z ≠ 0 → x^n + y^n ≠ z^n

#check fermat_last_theorem
```

Some expressions have a type, *P*, where *P* itself has type *Prop*. Such an expression is a proof of the proposition *P*.

```
theorem easy : 2 + 2 = 4 := rfl

#check easy

theorem hard : fermat_last_theorem := sorry

#check hard
```

If you manage to construct an expression of type *fermat_last_theorem* and Lean accepts it as a term of that type, you have done something very impressive. (Using `sorry` is cheating, and Lean knows it.) So now you know the game. All that is left to learn are the rules.

This book is complementary to a companion tutorial, Theorem Proving in Lean, which provides a more thorough introduction to the underlying logical framework and core syntax of Lean. *Theorem Proving in Lean* is for people who prefer to read a user manual cover to cover before using a new dishwasher. If you are the kind of person who prefers to hit the *start* button and figure out how to activate the potscrubber feature later, it makes more sense to start here and refer back to *Theorem Proving in Lean* as necessary.

Another thing that distinguishes *Mathematics in Lean* from *Theorem Proving in Lean* is that here we place a much greater emphasis on the use of *tactics*. Given that we are trying to build complex expressions, Lean offers two ways of going about it: we can write down the expressions themselves (that is, suitable text descriptions thereof), or we can provide Lean with *instructions* as to how to construct them. For example, the following expression represents a proof of the fact that if `n` is even then so is `m * n`:

```
example : ∀ m n : nat, even n → even (m * n) :=
assume m n ⟨k, (hk : n = 2 * k)⟩,
have hmn : m * n = 2 * (m * k),
```

(continues on next page)

(continued from previous page)

```
  by rw [hk, mul_left_comm],
show ∃ l, m * n = 2 * l,
  from ⟨_, hmn⟩
```

The *proof term* can be compressed to a single line:

```
example : ∀ m n : nat, even n → even (m * n) :=
λ m n ⟨k, hk⟩, ⟨m * k, by rw [hk, mul_left_comm]⟩
```

The following is, instead, a *tactic-style* proof of the same theorem:

```
example : ∀ m n : nat, even n → even (m * n) :=
begin
  -- say m and n are natural numbers, and assume n=2*k
  rintros m n ⟨k, hk⟩,
  -- We need to prove m*n is twice a natural. Let's show it's twice m*k.
  use m * k,
  -- substitute in for n
  rw hk,
  -- and now it's obvious
  ring
end
```

As you enter each line of such a proof in VS Code, Lean displays the *proof state* in a separate window, telling you what facts you have already established and what tasks remain to prove your theorem. You can replay the proof by stepping through the lines, since Lean will continue to show you the state of the proof at the point where the cursor is. In this example, you will then see that the first line of the proof introduces `m` and `n` (we could have renamed them at that point, if we wanted to), and also decomposes the hypothesis `even n` to a `k` and the assumption that `n = 2 * k`. The second line, `use m * k`, declares that we are going to show that `m * n` is even by showing `m * n = 2 * (m * k)`. The next line uses the `rewrite` tactic to replace `n` by `2 * k` in the goal, and the *ring* tactic solves the resulting goal `m * (2 * k) = 2 * (m * k)`.

The ability to build a proof in small steps with incremental feedback is extremely powerful. For that reason, tactic proofs are often easier and quicker to write than proof terms. There isn't a sharp distinction between the two: tactic proofs can be inserted in proof terms, as we did with the phrase `by rw [hk, mul_left_comm]` in the example above. We will also see that, conversely, it is often useful to insert a short proof term in the middle of a tactic proof. That said, in this book, our emphasis will be on the use of tactics.

In our example, the tactic proof can also be reduced to a one-liner:

```
example : ∀ m n : nat, even n → even (m * n) :=
by { rintros m n ⟨k, hk⟩, use m * k, rw hk, ring }
```

Here we have used tactics to carry out small proof steps. But they can also provide substantial automation, and justify longer calculations and bigger inferential steps. For example, we can invoke Lean's simplifier with specific rules for simplifying statements about parity to prove our theorem automatically.

```
example : ∀ m n : nat, even n → even (m * n) :=
by intros; simp * with parity_simps
```

Another big difference between the two introductions is that *Theorem Proving in Lean* depends only on core Lean and its built-in tactics, whereas *Mathematics in Lean* is built on top of Lean's powerful and ever-growing library, *mathlib*. As a result, we can show you how to use some of the mathematical objects and theorems in the library, and some of the very useful tactics. This book is not meant to be used as an overview of the library; the *community <https://leanprover-community.github.io/>_* web pages contain extensive documentation. Rather, our goal is to introduce you to the style of

thinking that underlies that formalization, so that you are comfortable browsing the library and finding things on your own.

Interactive theorem proving can be frustrating, and the learning curve is steep. But the Lean community is very welcoming to newcomers, and people are available on the Lean Zulip chat group round the clock to answer questions. We hope to see you there, and have no doubt that soon enough you, too, will be able to answer such questions and contribute to the development of *mathlib*.

So here is your mission, should you choose to accept it: dive in, try the exercises, come to Zulip with questions, and have fun. But be forewarned: interactive theorem proving will challenge you to think about mathematics and mathematical reasoning in fundamentally new ways. Your life may never be the same.

Acknowledgments. We are grateful to Gabriel Ebner for setting up the infrastructure for running this tutorial in VS Code. We are also grateful for help from Bryan Gin-ge Chen, Johan Commelin, Julian Külshammer, and Guilherme Silva. Our work has been partially supported by the Hoskinson Center for Formal Mathematics.

CHAPTER

TWO

BASICS

This chapter is designed to introduce you to the nuts and bolts of mathematical reasoning in Lean: calculating, applying lemmas and theorems, and reasoning about generic structures.

2.1 Calculating

We generally learn to carry out mathematical calculations without thinking of them as proofs. But when we justify each step in a calculation, as Lean requires us to do, the net result is a proof that the left-hand side of the calculation is equal to the right-hand side.

In Lean, stating a theorem is tantamount to stating a goal, namely, the goal of proving the theorem. Lean provides the `rewrite` tactic, abbreviated `rw`, to replace the left-hand side of an identity by the right-hand side in the goal. If `a`, `b`, and `c` are real numbers, `mul_assoc a b c` is the identity `a * b * c = a * (b * c)` and `mul_comm a b` is the identity `a * b = b * a`. Lean provides automation that generally eliminates the need to refer the facts like these explicitly, but they are useful for the purposes of illustration. In Lean, multiplication associates to the left, so the left-hand side of `mul_assoc` could also be written `(a * b) * c`. However, it is generally good style to be mindful of Lean's notational conventions and leave out parentheses when Lean does as well.

Let's try out `rw`.

```
example (a b c : ℝ) : (a * b) * c = b * (a * c) :=
begin
  rw mul_comm a b,
  rw mul_assoc b a c
end
```

The `import` line at the beginning of the example imports the theory of the real numbers from `mathlib`. For the sake of brevity, we generally suppress information like this when it is repeated from example to example. Clicking the `try it!` button displays the full example as it is meant to be processed and checked by Lean.

You are welcome to make changes to see what happens. You can type the ℝ character as `\R` or `\real` in VS Code. The symbol doesn't appear until you hit space or the tab key. If you hover over a symbol when reading a Lean file, VS Code will show you the syntax that can be used to enter it. If your keyboard does not have an easily accessible backslash, you can change the leading character by changing the `lean.input.leader` setting.

When a cursor is in the middle of a tactic proof, Lean reports on the current *proof state* in the *Lean infoview* window. As you move your cursor past each step of the proof, you can see the state change. A typical proof state in Lean might look as follows:

```
1 goal
x y : ℕ,
h₁ : prime x,
h₂ : ¬even x,
```

(continues on next page)

(continued from previous page)

```
h₃ : y > x
⊢ y ≥ 4
```

The lines before the one that begins with ⊢ denote the *context*: they are the objects and assumptions currently at play. In this example, these include two objects, `x` and `y`, each a natural number. They also include three assumptions, labelled `h₁`, `h₂`, and `h₃`. In Lean, everything in a context is labelled with an identifier. You can type these subscripted labels as `h\1`, `h\2`, and `h\3`, but any legal identifiers would do: you can use `h1`, `h2`, `h3` instead, or `foo`, `bar`, and `baz`. The last line represents the *goal*, that is, the fact to be proved. Sometimes people use *target* for the fact to be proved, and *goal* for the combination of the context and the target. In practice, the intended meaning is usually clear.

Try proving these identities, in each case replacing `sorry` by a tactic proof. With the `rw` tactic, you can use a left arrow (`\l`) to reverse an identity. For example, `rw ← mul_assoc a b c` replaces `a * (b * c)` by `a * b * c` in the current goal.

```
example (a b c : ℝ) : (c * b) * a = b * (a * c) :=
begin
  sorry
end

example (a b c : ℝ) : a * (b * c) = b * (a * c) :=
begin
  sorry
end
```

You can also use identities like `mul_assoc` and `mul_comm` without arguments. In this case, the rewrite tactic tries to match the left-hand side with an expression in the goal, using the first pattern it finds.

```
example (a b c : ℝ) : a * b * c = b * c * a :=
begin
  rw mul_assoc,
  rw mul_comm
end
```

You can also provide *partial* information. For example, `mul_comm a` matches any pattern of the form `a * ?` and rewrites it to `? * a`. Try doing the first of these examples without providing any arguments at all, and the second with only one argument.

```
example (a b c : ℝ) : a * (b * c) = b * (c * a) :=
begin
  sorry
end

example (a b c : ℝ) : a * (b * c) = b * (a * c) :=
begin
  sorry
end
```

You an also use `rw` with facts from the local context.

```
example (a b c d e f : ℝ) (h : a * b = c * d) (h' : e = f) :
  a * (b * e) = c * (d * f) :=
begin
  rw h',
  rw ←mul_assoc,
  rw h,
```

(continues on next page)

(continued from previous page)

```
  rw mul_assoc
end
```

Try these:

```
example (a b c d e f : ℝ) (h : b * c = e * f) :
  a * b * c * d = a * e * f * d :=
begin
  sorry
end

example (a b c d : ℝ) (hyp : c = b * a - d) (hyp' : d = a * b) : c = 0 :=
begin
  sorry
end
```

For the second one, you can use the theorem `sub_self`, where `sub_self a` is the identity `a - a = 0`.

We now introduce some useful features of Lean. First, multiple rewrite commands can be carried out with a single command, by listing the relevant identities within square brackets. Second, when a tactic proof is just a single command, we can replace the `begin ... end` block with a `by`.

```
example (a b c d e f : ℝ) (h : a * b = c * d) (h' : e = f) :
  a * (b * e) = c * (d * f) :=
by rw [h', ←mul_assoc, h, mul_assoc]
```

You still see the incremental progress by placing the cursor after a comma in any list of rewrites.

Another trick is that we can declare variables once and for all outside an example or theorem. When Lean sees them mentioned in the statement of the theorem, it includes them automatically.

```
variables a b c d e f g : ℝ

example (h : a * b = c * d) (h' : e = f) :
  a * (b * e) = c * (d * f) :=
by rw [h', ←mul_assoc, h, mul_assoc]
```

Inspection of the tactic state at the beginning of the above proof reveals that Lean indeed included the relevant variables, leaving out *g* that doesn't feature in the statement. We can delimit the scope of the declaration by putting it in a `section ... end` block. Finally, recall from the introduction that Lean provides us with a command to determine the type of an expression:

```
section
variables a b c : ℝ

#check a
#check a + b
#check (a : ℝ)
#check mul_comm a b
#check (mul_comm a b : a * b = b * a)
#check mul_assoc c a b
#check mul_comm a
#check mul_comm
#check @mul_comm

end
```

The `#check` command works for both objects and facts. In response to the command `#check a`, Lean reports that `a` has type $\mathbb{R}$. In response to the command `#check mul_comm a b`, Lean reports that `mul_comm a b` is a proof of the fact `a * b = b * a`. The command `#check (a : ℝ)` states our expectation that the type of `a` is $\mathbb{R}$, and Lean will raise an error if that is not the case. We will explain the output of the last three `#check` commands later, but in the meanwhile, you can take a look at them, and experiment with some `#check` commands of your own.

Let's try some more examples. The theorem `two_mul a` says that `2 * a = a + a`. The theorems `add_mul` and `mul_add` express the distributivity of multiplication over addition, and the theorem `add_assoc` expresses the associativity of addition. Use the `#check` command to see the precise statements.

```
example : (a + b) * (a + b) = a * a + 2 * (a * b) + b * b :=
begin
  rw [mul_add, add_mul, add_mul],
  rw [←add_assoc, add_assoc (a * a)],
  rw [mul_comm b a, ←two_mul]
end
```

Whereas it is possible to figure out what it going on in this proof by stepping through it in the editor, it is hard to read on its own. Lean provides a more structured way of writing proofs like this using the `calc` keyword.

```
example : (a + b) * (a + b) = a * a + 2 * (a * b) + b * b :=
calc
  (a + b) * (a + b)
      = a * a + b * a + (a * b + b * b) :
          by rw [mul_add, add_mul, add_mul]
  ... = a * a + (b * a + a * b) + b * b :
          by rw [←add_assoc, add_assoc (a * a)]
  ... = a * a + 2 * (a * b) + b * b     :
          by rw [mul_comm b a, ←two_mul]
```

Notice that there is no more `begin ... end` block: an expression that begins with `calc` is a *proof term*. A `calc` expression can also be used inside a tactic proof, but Lean interprets it as the instruction to use the resulting proof term to solve the goal.

The `calc` syntax is finicky: the dots and colons and justification have to be in the format indicated above. Lean ignores whitespace like spaces, tabs, and returns, so you have some flexibility to make the calculation look more attractive. One way to write a `calc` proof is to outline it first using the `sorry` tactic for justification, make sure Lean accepts the expression modulo these, and then justify the individual steps using tactics.

```
example : (a + b) * (a + b) = a * a + 2 * (a * b) + b * b :=
calc
  (a + b) * (a + b)
      = a * a + b * a + (a * b + b * b) :
    begin
      sorry
    end
  ... = a * a + (b * a + a * b) + b * b : by sorry
  ... = a * a + 2 * (a * b) + b * b     : by sorry
```

Try proving the following identity using both a pure `rw` proof and a more structured `calc` proof:

```
example : (a + b) * (c + d) = a * c + a * d + b * c + b * d :=
sorry
```

The following exercise is a little more challenging. You can use the theorems listed underneath.

```
example (a b : ℝ) : (a + b) * (a - b) = a^2 - b^2 :=
begin
```

(continues on next page)

(continued from previous page)

```
  sorry
end

#check pow_two a
#check mul_sub a b c
#check add_mul a b c
#check add_sub a b c
#check sub_sub a b c
#check add_zero a
```

We can also perform rewriting in an assumption in the context. For example, `rw mul_comm a b at hyp` replaces `a * b` by `b * a` in the assumption `hyp`.

```
example (a b c d : ℝ) (hyp : c = d * a + b) (hyp' : b = a * d) :
  c = 2 * a * d :=
begin
  rw hyp' at hyp,
  rw mul_comm d a at hyp,
  rw ← two_mul (a * d) at hyp,
  rw ← mul_assoc 2 a d at hyp,
  exact hyp
end
```

In the last step, the `exact` tactic can use `hyp` to solve the goal because at that point `hyp` matches the goal exactly.

We close this section by noting that `mathlib` provides a useful bit of automation with a `ring` tactic, which is designed to prove identities in any commutative ring.

```
example : (c * b) * a = b * (a * c) :=
by ring

example : (a + b) * (a + b) = a * a + 2 * (a * b) + b * b :=
by ring

example : (a + b) * (a - b) = a^2 - b^2 :=
by ring

example (hyp : c = d * a + b) (hyp' : b = a * d) :
  c = 2 * a * d :=
begin
  rw [hyp, hyp'],
  ring
end
```

The `ring` tactic is imported indirectly when we import `data.real.basic`, but we will see in the next section that it can be used for calculations on structures other than the real numbers. It can be imported explicitly with the command `import tactic`. We will see there are similar tactics for other common kind of algebraic structures.

2.2 Proving Identities in Algebraic Structures

Mathematically, a ring consists of a collection of objects, R, operations $+$ $\times$, and constants 0 and 1, and an operation $x \mapsto -x$ such that:

- R with $+$ is an *abelian group*, with 0 as the additive identity and negation as inverse.
- Multiplication is associative with identity 1, and multiplication distributes over addition.

In Lean, the collection of objects is represented as a *type*, `R`. The ring axioms are as follows:

```
variables (R : Type*) [ring R]

#check (add_assoc : ∀ a b c : R, a + b + c = a + (b + c))
#check (add_comm : ∀ a b : R, a + b = b + a)
#check (zero_add : ∀ a : R, 0 + a = a)
#check (add_left_neg : ∀ a : R, -a + a = 0)
#check (mul_assoc : ∀ a b c : R, a * b * c = a * (b * c))
#check (mul_one : ∀ a : R, a * 1 = a)
#check (one_mul : ∀ a : R, 1 * a = a)
#check (mul_add : ∀ a b c : R, a * (b + c) = a * b + a * c)
#check (add_mul : ∀ a b c : R, (a + b) * c = a * c + b * c)
```

You will learn more about the square brackets in the first line later, but for the time being, suffice it to say that the declaration gives us a type, `R`, and a ring structure on `R`. Lean then allows us to use generic ring notation with elements of `R`, and to make use of a library of theorems about rings.

The names of some of the theorems should look familiar: they are exactly the ones we used to calculate with the real numbers in the last section. Lean is good not only for proving things about concrete mathematical structures like the natural numbers and the integers, but also for proving things about abstract structures, characterized axiomatically, like rings. Moreover, Lean supports *generic reasoning* about both abstract and concrete structures, and can be trained to recognized appropriate instances. So any theorem about rings can be applied to concrete rings like the integers, $\mathbb{Z}$, the rational numbers, $\mathbb{Q}$, and the complex numbers $\mathbb{C}$. It can also be applied to any instance of an abstract structure that extends rings, such as any *ordered ring* or any *field*.

Not all important properties of the real numbers hold in an arbitrary ring, however. For example, multiplication on the real numbers is commutative, but that does not hold in general. If you have taken a course in linear algebra, you will recognize that, for every n, the n by n matrices of real numbers form a ring in which commutativity usually fails. If we declare `R` to be a *commutative* ring, in fact, all the theorems in the last section continue to hold when we replace $\mathbb{R}$ by `R`.

```
variables (R : Type*) [comm_ring R]
variables a b c d : R

example : (c * b) * a = b * (a * c) :=
by ring

example : (a + b) * (a + b) = a * a + 2 * (a * b) + b * b :=
by ring

example : (a + b) * (a - b) = a^2 - b^2 :=
by ring

example (hyp : c = d * a + b) (hyp' : b = a * d) :
  c = 2 * a * d :=
begin
  rw [hyp, hyp'],
  ring
end
```

We leave it to you to check that all the other proofs go through unchanged.

The goal of this section is to strengthen the skills you have developed in the last section and apply them to reasoning axiomatically about rings. We will start with the axioms listed above, and use them to derive other facts. Most of the facts we prove are already in `mathlib`. We will give the versions we prove the same names to help you learn the contents of the library as well as the naming conventions.

Lean provides an organizational mechanism similar to those used in programming languages: when a definition or theorem `foo` is introduced in a *namespace* `bar`, its full name is `bar.foo`. The command `open bar` later *opens* the namespace, which allows us to use the shorter name `foo`. To avoid errors due to name clashes, in the next example we put our versions of the library theorems in a new namespace called `my_ring`.

The next example shows that we do not need `add_zero` or `add_right_neg` as ring axioms, because they follow from the other axioms. import tactic

```
namespace my_ring
variables {R : Type*} [ring R]

theorem add_zero (a : R) : a + 0 = a :=
by rw [add_comm, zero_add]

theorem add_right_neg (a : R) : a + -a = 0 :=
by rw [add_comm, add_left_neg]

#check @my_ring.add_zero
#check @add_zero

end my_ring
```

The net effect is that we can temporarily reprove a theorem in the library, and then go on using the library version after that. But don't cheat! In the exercises that follow, take care to use only the general facts about rings that we have proved earlier in this section.

(If you are paying careful attention, you may have noticed that we changed the round brackets in `(R : Type*)` for curly brackets in `{R : Type*}`. This declares `R` to be an *implicit argument*. We will explain what this means in a moment, but don't worry about it in the meanwhile.)

Here is a useful theorem:

```
theorem neg_add_cancel_left (a b : R) : -a + (a + b) = b :=
by rw [←add_assoc, add_left_neg, zero_add]
```

Prove the companion version:

```
theorem add_neg_cancel_right (a b : R) : (a + b) + -b = a :=
sorry
```

Use these to prove the following:

```
theorem add_left_cancel {a b c : R} (h : a + b = a + c) : b = c :=
sorry

theorem add_right_cancel {a b c : R} (h : a + b = c + b) : a = c :=
sorry
```

With enough planning, you can do each of them with three rewrites.

We can now explain the use of the curly braces. Imagine you are in a situation where you have `a`, `b`, and `c` in your context, as well as a hypothesis `h : a + b = a + c`, and you would like to draw the conclusion `b = c`. In Lean, you can apply a theorem to hypotheses and facts just the same way that you can apply them to objects, so you might think

that `add_left_cancel a b c h` is a proof of the fact `b = c`. But notice that explicitly writing `a`, `b`, and `c` is redundant, because the hypothesis `h` makes it clear that those are the objects we have in mind. In this case, typing a few extra characters is not onerous, but if we wanted to apply `add_left_cancel` to more complicated expressions, writing them would be tedious. In cases like these, Lean allows us to mark arguments as *implicit*, meaning that they are supposed to be left out and inferred by other means, such as later arguments and hypotheses. The curly brackets in `{a b c : R}` do exactly that. So, given the statement of the theorem above, the correct expression is simply `add_left_cancel h`.

To illustrate, let us show that `a * 0 = 0` follows from the ring axioms.

```
theorem mul_zero (a : R) : a * 0 = 0 :=
begin
  have h : a * 0 + a * 0 = a * 0 + 0,
  { rw [←mul_add, add_zero, add_zero] },
  rw add_left_cancel h
end
```

We have used a new trick! If you step through the proof, you can see what is going on. The `have` tactic introduces a new goal, `a * 0 + a * 0 = a * 0 + 0`, with the same context as the original goal. In the next line, we could have omitted the curly brackets, which serve as an inner `begin ... end` pair. Using them promotes a modular style of proof: the part of the proof inside the brackets establishes the goal that was introduced by the `have`. After that, we are back to proving the original goal, except a new hypothesis `h` has been added: having proved it, we are now free to use it. At this point, the goal is exactly the result of `add_left_cancel h`. We could equally well have closed the proof with `apply add_left_cancel h` or `exact add_left_cancel h`.

Remember that multiplication is not assumed to be commutative, so the following theorem also requires some work.

```
theorem zero_mul (a : R) : 0 * a = 0 :=
sorry
```

By now, you should also be able replace each `sorry` in the next exercise with a proof, still using only facts about rings that we have established in this section.

```
theorem neg_eq_of_add_eq_zero {a b : R} (h : a + b = 0) : -a = b :=
sorry

theorem eq_neg_of_add_eq_zero {a b : R} (h : a + b = 0) : a = -b :=
sorry

theorem neg_zero : (-0 : R) = 0 :=
begin
  apply neg_eq_of_add_eq_zero,
  rw add_zero
end

theorem neg_neg (a : R) : -(-a) = a :=
sorry
```

We had to use the annotation `(-0 : R)` instead of `0` in the third theorem because without specifying `R` it is impossible for Lean to infer which `0` we have in mind, and by default it would be interpreted as a natural number.

In Lean, subtraction in a ring is provably equal to addition of the additive inverse.

```
example (a b : R) : a - b = a + -b :=
sub_eq_add_neg a b
```

On the real numbers, it is *defined* that way:

```
example (a b : ℝ) : a - b = a + -b :=
rfl

example (a b : ℝ) : a - b = a + -b :=
by reflexivity
```

The proof term `rfl` is short for `reflexivity`. Presenting it as a proof of `a - b = a + -b` forces Lean to unfold the definition and recognize both sides as being the same. The `reflexivity` tactic, which can be abbreviated as `refl`, does the same. This is an instance of what is known as a *definitional equality* in Lean's underlying logic. This means that not only can one rewrite with `sub_eq_add_neg` to replace `a - b = a + -b`, but in some contexts, when dealing with the real numbers, you can use the two sides of the equation interchangeably. For example, you now have enough information to prove the theorem `self_sub` from the last section:

```
theorem self_sub (a : R) : a - a = 0 :=
sorry
```

Show that you can prove this using `rw`, but if you replace the arbitrary ring `R` by the real numbers, you can also prove it using either `apply` or `exact`.

For another example of definitional equality, Lean knows that `1 + 1 = 2` holds in any ring. With a bit of effort, you can use that to prove the theorem `two_mul` from the last section:

```
lemma one_add_one_eq_two : 1 + 1 = (2 : R) :=
by refl

theorem two_mul (a : R) : 2 * a = a + a :=
sorry
```

We close this section by noting that some of the facts about addition and negation that we established above do not need the full strength of the ring axioms, or even commutativity of addition. The weaker notion of a *group* can be axiomatized as follows:

```
variables (A : Type*) [add_group A]

#check (add_assoc : ∀ a b c : A, a + b + c = a + (b + c))
#check (zero_add : ∀ a : A, 0 + a = a)
#check (add_left_neg : ∀ a : A, -a + a = 0)
```

It is conventional to use additive notation when the group operation is commutative, and multiplicative notation otherwise. So Lean defines a multiplicative version as well as the additive version (and also their abelian variants, `add_comm_group` and `comm_group`).

```
variables {G : Type*} [group G]

#check (mul_assoc : ∀ a b c : G, a * b * c = a * (b * c))
#check (one_mul : ∀ a : G, 1 * a = a)
#check (mul_left_inv : ∀ a : G, a⁻¹ * a = 1)
```

If you are feeling cocky, try proving the following facts about groups, using only these axioms. You will need to prove a number of helper lemmas along the way. The proofs we have carried out in this section provide some hints.

```
theorem mul_right_inv (a : G) : a * a⁻¹ = 1 :=
sorry

theorem mul_one (a : G) : a * 1 = a :=
sorry
```

(continues on next page)

(continued from previous page)

```
theorem mul_inv_rev (a b : G) : (a * b)⁻¹ = b⁻¹ * a⁻¹ :=
sorry
```

Explicitly invoking those lemmas is tedious, so mathlib provides tactics similar to *ring* in order to cover most uses: *group* is for non-commutative multiplicative groups, *abel* for abelian additive groups, and *noncomm_ring* for non-commutative groups. It may seem odd that the algebraic structures are called *ring* and *comm_ring* while the tactics are named *noncomm_ring* and *ring*. This is partly for historical reasons, but also for the convenience of using a shorter name for the tactic that deals with commutative rings, since it is used more often.

2.3 Using Theorems and Lemmas

Rewriting is great for proving equations, but what about other sorts of theorems? For example, how can we prove an inequality, like the fact that $a + e^b \leq a + e^c$ holds whenever $b \leq c$? We have already seen that theorems can be applied to arguments and hypotheses, and that the `apply` and `exact` tactics can be used to solve goals. In this section, we will make good use of these tools.

Consider the library theorems `le_refl` and `le_trans`:

```
#check (le_refl : ∀ a : ℝ, a ≤ a)
#check (le_trans : a ≤ b → b ≤ c → a ≤ c)
```

As we explain in more detail in Section 3.1, the implicit parentheses in the statement of `le_trans` associate to the right, so it should be interpreted as `a ≤ b → (b ≤ c → a ≤ c)`. The library designers have set the arguments to `le_trans` implicit, so that Lean will *not* let you provide them explicitly (unless you really insist, as we will discuss later). Rather, it expects to infer them from the context in which they are used. For example, when hypotheses `h : a ≤ b` and `h' : b ≤ c` are in the context, all the following work:

```
variables (h : a ≤ b) (h' : b ≤ c)

#check (le_refl : ∀ a : real, a ≤ a)
#check (le_refl a : a ≤ a)
#check (le_trans : a ≤ b → b ≤ c → a ≤ c)
#check (le_trans h : b ≤ c → a ≤ c)
#check (le_trans h h' : a ≤ c)
```

The `apply` tactic takes a proof of a general statement or implication, tries to match the conclusion with the current goal, and leaves the hypotheses, if any, as new goals. If the given proof matches the goal exactly (modulo *definitional* equality), you can use the `exact` tactic instead of `apply`. So, all of these work:

```
example (x y z : ℝ) (h₀ : x ≤ y) (h₁ : y ≤ z) : x ≤ z :=
begin
  apply le_trans,
  { apply h₀ },
  apply h₁
end

example (x y z : ℝ) (h₀ : x ≤ y) (h₁ : y ≤ z) : x ≤ z :=
begin
  apply le_trans h₀,
  apply h₁
end
```

(continues on next page)

(continued from previous page)

```
example (x y z : ℝ) (h₀ : x ≤ y) (h₁ : y ≤ z) : x ≤ z :=
by exact le_trans h₀ h₁

example (x y z : ℝ) (h₀ : x ≤ y) (h₁ : y ≤ z) : x ≤ z :=
le_trans h₀ h₁

example (x : ℝ) : x ≤ x :=
by apply le_refl

example (x : ℝ) : x ≤ x :=
by exact le_refl x

example (x : ℝ) : x ≤ x :=
le_refl x
```

In the first example, applying `le_trans` creates two goals, and we use the curly braces to enclose the proof of the first one. In the fourth example and in the last example, we avoid going into tactic mode entirely: `le_trans h₀ h₁` and `le_refl x` are the proof terms we need.

Here are a few more library theorems:

```
#check (le_refl  : ∀ a, a ≤ a)
#check (le_trans : a ≤ b → b ≤ c → a ≤ c)
#check (lt_of_le_of_lt : a ≤ b → b < c → a < c)
#check (lt_of_lt_of_le : a < b → b ≤ c → a < c)
#check (lt_trans : a < b → b < c → a < c)
```

Use them together with `apply` and `exact` to prove the following:

```
example (h₀ : a ≤ b) (h₁ : b < c) (h₂ : c ≤ d)
    (h₃ : d < e) :
  a < e :=
sorry
```

In fact, Lean has a tactic that does this sort of thing automatically:

```
example (h₀ : a ≤ b) (h₁ : b < c) (h₂ : c ≤ d)
    (h₃ : d < e) :
  a < e :=
by linarith
```

The `linarith` tactic is designed to handle *linear arithmetic*.

```
example (h : 2 * a ≤ 3 * b) (h' : 1 ≤ a) (h'' : d = 2) :
  d + a ≤ 5 * b :=
by linarith
```

In addition to equations and inequalities in the context, `linarith` will use additional inequalities that you pass as arguments. In the next example, `exp_le_exp.mpr h'` is a proof of `exp b ≤ exp c`, as we will explain in a moment. Notice that, in Lean, we write `f x` to denote the application of a function `f` to the argument `x`, exactly the same way we write `h x` to denote the result of applying a fact or theorem `h` to the argument `x`. Parentheses are only needed for compound arguments, as in `f (x + y)`. Without the parentheses, `f x + y` would be parsed as `(f x) + y`.

```
example (h : 1 ≤ a) (h' : b ≤ c) :
  2 + a + exp b ≤ 3 * a + exp c :=
by linarith [exp_le_exp.mpr h']
```

Here are some more theorems in the library that can be used to establish inequalities on the real numbers.

```
#check (exp_le_exp : exp a ≤ exp b ↔ a ≤ b)
#check (exp_lt_exp : exp a < exp b ↔ a < b)
#check (log_le_log : 0 < a → 0 < b → (log a ≤ log b ↔ a ≤ b))
#check (log_lt_log : 0 < a → a < b → log a < log b)
#check (add_le_add : a ≤ b → c ≤ d → a + c ≤ b + d)
#check (add_le_add_left : a ≤ b → ∀ c, c + a ≤ c + b)
#check (add_le_add_right : a ≤ b → ∀ c, a + c ≤ b + c)
#check (add_lt_add_of_le_of_lt : a ≤ b → c < d → a + c < b + d)
#check (add_lt_add_of_lt_of_le : a < b → c ≤ d → a + c < b + d)
#check (add_lt_add_left : a < b → ∀ c, c + a < c + b)
#check (add_lt_add_right : a < b → ∀ c, a + c < b + c)
#check (add_nonneg : 0 ≤ a → 0 ≤ b → 0 ≤ a + b)
#check (add_pos : 0 < a → 0 < b → 0 < a + b)
#check (add_pos_of_pos_of_nonneg : 0 < a → 0 ≤ b → 0 < a + b)
#check (exp_pos : ∀ a, 0 < exp a)

#check @add_le_add_left
```

Some of the theorems, `exp_le_exp`, `exp_lt_exp`, and `log_le_log` use a *bi-implication*, which represents the phrase "if and only if." (You can type it in VS Code with `\lr` of `\iff`). We will discuss this connective in greater detail in the next chapter. Such a theorem can be used with `rw` to rewrite a goal to an equivalent one:

```
example (h : a ≤ b) : exp a ≤ exp b :=
begin
  rw exp_le_exp,
  exact h
end
```

In this section, however, we will use the fact that if `h : A ↔ B` is such an equivalence, then `h.mp` establishes the forward direction, `A → B`, and `h.mpr` establishes the reverse direction, `B → A`. Here, `mp` stands for "modus ponens" and `mpr` stands for "modus ponens reverse." You can also use `h.1` and `h.2` for `h.mp` and `h.mpr`, respectively, if you prefer. Thus the following proof works:

```
example (h₀ : a ≤ b) (h₁ : c < d) : a + exp c + e < b + exp d + e :=
begin
  apply add_lt_add_of_lt_of_le,
  { apply add_lt_add_of_le_of_lt h₀,
    apply exp_lt_exp.mpr h₁ },
  apply le_refl
end
```

The first line, `apply add_lt_add_of_lt_of_le`, creates two goals, and once again we use the curly brackets to separate the proof of the first from the proof of the second.

Try the following examples on your own. The example in the middle shows you that the `norm_num` tactic can be used to solve concrete numeric goals.

```
example (h₀ : d ≤ e) : c + exp (a + d) ≤ c + exp (a + e) :=
begin
  sorry
end

example : (0 : ℝ) < 1 :=
by norm_num

example (h : a ≤ b) : log (1 + exp a) ≤ log (1 + exp b) :=
```

(continues on next page)

(continued from previous page)

```
begin
  have h₀ : 0 < 1 + exp a,
  { sorry },
  have h₁ : 0 < 1 + exp b,
  { sorry },
  apply (log_le_log h₀ h₁).mpr,
  sorry
end
```

From these examples, it should be clear that being able to find the library theorems you need constitutes an important part of formalization. There are a number of strategies you can use:

- You can browse mathlib in its GitHub repository.
- You can use the API documentation on the mathlib web pages.
- You can rely on mathlib naming conventions and tab completion in the editor to guess a theorem name. In Lean, a theorem named `A_of_B_of_C` establishes something of the form `A` from hypotheses of the form `B` and `C`, where `A`, `B`, and `C` approximate the way we might read the goals out loud. So a theorem establishing something like `x + y ≤ ...` will probably start with `add_le`. Typing `add_le` and hitting tab will give you some helpful choices.
- If you right-click on an existing theorem name in VS Code, the editor will show a menu with the option to jump to the file where the theorem is defined, and you can find similar theorems nearby.
- You can use the `library_search` tactic, which tries to find the relevant theorem in the library.

```
example : 0 ≤ a^2 :=
begin
  -- library_search,
  exact pow_two_nonneg a
end
```

To try out `library_search` in this example, delete the `exact` command and uncomment the previous line. If you replace `library_search` with `suggest`, you'll see a long list of suggestions. In this case, the suggestions are not helpful, but in other cases it does better. Using these tricks, see if you can find what you need to do the next example:

```
example (h : a ≤ b) : c - exp b ≤ c - exp a :=
  sorry
```

Using the same tricks, confirm that `linarith` instead of `library_search` can also finish the job.

Here is another example of an inequality:

```
example : 2*a*b ≤ a^2 + b^2 :=
begin
  have h : 0 ≤ a^2 - 2*a*b + b^2,
  calc
    a^2 - 2*a*b + b^2 = (a - b)^2      : by ring
    ... ≥ 0                            : by apply pow_two_nonneg,
  calc
    2*a*b
        = 2*a*b + 0                    : by ring
    ... ≤ 2*a*b + (a^2 - 2*a*b + b^2) : add_le_add (le_refl _) h
    ... = a^2 + b^2                    : by ring
end
```

Mathlib tends to put spaces around binary operations like `*` and `^`, but in this example, the more compressed format increases readability. There are a number of things worth noticing. First, an expression `s ≥ t` is definitionally equivalent to `t ≤ s`. In principle, this means one should be able to use them interchangeably. But some of Lean's automation does

not recognize the equivalence, so mathlib tends to favor $\leq$ over $\geq$. Second, we have used the `ring` tactic extensively. It is a real timesaver! Finally, notice that in the second line of the second `calc` proof, instead of writing `by exact add_le_add (le_refl _) h`, we can simply write the proof term `add_le_add (le_refl _) h`.

In fact, the only cleverness in the proof above is figuring out the hypothesis `h`. Once we have it, the second calculation involves only linear arithmetic, and `linarith` can handle it:

```
example : 2*a*b ≤ a^2 + b^2 :=
begin
  have h : 0 ≤ a^2 - 2*a*b + b^2,
  calc
    a^2 - 2*a*b + b^2 = (a - b)^2 : by ring
    ... ≥ 0                       : by apply pow_two_nonneg,
  linarith
end
```

How nice! We challenge you to use these ideas to prove the following theorem. You can use the theorem `abs_le'.mpr`.

```
example : abs (a*b) ≤ (a^2 + b^2) / 2 :=
sorry

#check abs_le'.mpr
```

If you managed to solve this, congratulations! You are well on your way to becoming a master formalizer.

2.4 More on Order and Divisibility

The `min` function on the real numbers is uniquely characterized by the following three facts:

```
#check (min_le_left a b : min a b ≤ a)
#check (min_le_right a b : min a b ≤ b)
#check (le_min : c ≤ a → c ≤ b → c ≤ min a b)
```

Can you guess the names of the theorems that characterize `max` in a similar way?

Notice that we have to apply `min` to a pair of arguments `a` and `b` by writing `min a b` rather than `min (a, b)`. Formally, `min` is a function of type $\mathbb{R} \to \mathbb{R} \to \mathbb{R}$. When we write a type like this with multiple arrows, the convention is that the implicit parentheses associate to the right, so the type is interpreted as $\mathbb{R} \to (\mathbb{R} \to \mathbb{R})$. The net effect is that if `a` and `b` have type $\mathbb{R}$ then `min a` has type $\mathbb{R} \to \mathbb{R}$ and `min a b` has type $\mathbb{R}$, so `min` acts like a function of two arguments, as we expect. Handling multiple arguments in this way is known as *currying*, after the logician Haskell Curry.

The order of operations in Lean can also take some getting used to. Function application binds tighter than infix operations, so the expression `min a b + c` is interpreted as `(min a b) + c`. With time, these conventions will become second nature.

Using the theorem `le_antisymm`, we can show that two real numbers are equal if each is less than or equal to the other. Using this and the facts above, we can show that `min` is commutative:

```
example : min a b = min b a :=
begin
  apply le_antisymm,
  { show min a b ≤ min b a,
    apply le_min,
    { apply min_le_right },
    apply min_le_left },
  { show min b a ≤ min a b,
```

(continues on next page)

(continued from previous page)

```
    apply le_min,
    { apply min_le_right },
    apply min_le_left }
end
```

Here we have used curly brackets to separate proofs of different goals. Our usage is inconsistent: at the outer level, we use curly brackets and indentation for both goals, whereas for the nested proofs, we use curly brackets only until a single goal remains. Both conventions are reasonable and useful. We also use the `show` tactic to structure the proof and indicate what is being proved in each block. The proof still works without the `show` commands, but using them makes the proof easier to read and maintain.

It may bother you that the the proof is repetitive. To foreshadow skills you will learn later on, we note that one way to avoid the repetition is to state a local lemma and then use it:

```
example : min a b = min b a :=
begin
  have h : ∀ x y, min x y ≤ min y x,
  { intros x y,
    apply le_min,
    apply min_le_right,
    apply min_le_left },
  apply le_antisymm, apply h, apply h
end
```

We will say more about the universal quantifier in Section 3.1, but suffice it to say here that the hypothesis `h` says that the desired inequality holds for any `x` and `y`, and the `intros` tactic introduces an arbitrary `x` and `y` to establish the conclusion. The first `apply` after `le_antisymm` implicitly uses `h a b`, whereas the second one uses `h b a`.

Another solution is to use the `repeat` tactic, which applies a tactic (or a block) as many times as it can.

```
example : min a b = min b a :=
begin
  apply le_antisymm,
  repeat {
    apply le_min,
    apply min_le_right,
    apply min_le_left }
end
```

In any case, whether or not you use these tricks, we encourage you to prove the following:

```
example : max a b = max b a :=
sorry

example : min (min a b) c = min a (min b c) :=
sorry
```

Of course, you are welcome to prove the associativity of `max` as well.

It is an interesting fact that `min` distributes over `max` the way that multiplication distributes over addition, and vice-versa. In other words, on the real numbers, we have the identity `min a (max b c) ≤ max (min a b) (min a c)` as well as the corresponding version with `max` and `min` switched. But in the next section we will see that this does *not* follow from the transitivity and reflexivity of $\leq$ and the characterizing properties of `min` and `max` enumerated above. We need to use the fact that $\leq$ on the real numbers is a *total order*, which is to say, it satisfies `∀ x y, x ≤ y ∨ y ≤ x`. Here the disjunction symbol, `∨`, represents "or". In the first case, we have `min x y = x`, and in the second case, we have `min x y = y`. We will learn how to reason by cases in Section 3.5, but for now we will stick to examples that don't require the case split.

Here is one such example:

```
lemma aux : min a b + c ≤ min (a + c) (b + c) :=
sorry

example : min a b + c = min (a + c) (b + c) :=
sorry
```

It is clear that `aux` provides one of the two inequalities needed to prove the equality, but applying it to suitable values yields the other direction as well. As a hint, you can use the theorem `add_neg_cancel_right` and the `linarith` tactic.

Lean's naming convention is made manifest in the library's name for the triangle inequality:

```
#check (abs_add : ∀ a b : ℝ, abs (a + b) ≤ abs a + abs b)
```

Use it to prove the following variant:

```
example : abs a - abs b ≤ abs (a - b) :=
sorry
```

See if you can do this in three lines or less. You can use the theorem `sub_add_cancel`.

Another important relation that we will make use of in the sections to come is the divisibility relation on the natural numbers, `x ∣ y`. Be careful: the divisibility symbol is *not* the ordinary bar on your keyboard. Rather, it is a unicode character obtained by typing `\|` in VS Code. By convention, mathlib uses `dvd` to refer to it in theorem names.

```
example (h₀ : x ∣ y) (h₁ : y ∣ z) : x ∣ z :=
dvd_trans h₀ h₁

example : x ∣ y * x * z :=
begin
  apply dvd_mul_of_dvd_left,
  apply dvd_mul_left
end

example : x ∣ x^2 :=
by apply dvd_mul_right
```

In the last example, the exponent is a natural number, and applying `dvd_mul_right` forces Lean to expand the definition of `x^2` to `x^1 * x`. See if you can guess the names of the theorems you need to prove the following:

```
example (h : x ∣ w) : x ∣ y * (x * z) + x^2 + w^2 :=
sorry
```

With respect to divisibility, the *greatest common divisor*, `gcd`, and least common multiple, `lcm`, are analogous to `min` and `max`. Since every number divides `0`, `0` is really the greatest element with respect to divisibility:

```
variables m n : ℕ
open nat

#check (gcd_zero_right n : gcd n 0 = n)
#check (gcd_zero_left n  : gcd 0 n = n)
#check (lcm_zero_right n : lcm n 0 = 0)
#check (lcm_zero_left n  : lcm 0 n = 0)
```

The functions `gcd` and `lcm` for natural numbers are in the `nat` namespace, which means that the full identifiers are `nat.gcd` and `nat.lcm`. Similarly, the names of the theorems listed are prefixed by `nat.`. The command `open nat` opens the namespace, allowing us to use the shorter names.

See if you can guess the names of the theorems you will need to prove the following:

```
example : gcd m n = gcd n m :=
sorry
```

Hint: you can use `dvd_antisymm`.

2.5 Proving Facts about Algebraic Structures

In Section 2.2, we saw that many common identities governing the real numbers hold in more general classes of algebraic structures, such as commutative rings. We can use any axioms we want to describe an algebraic structure, not just equations. For example, a *partial order* consists of a set with a binary relation that is reflexive and transitive, like $\leq$ on the real numbers. Lean knows about partial orders:

```
variables {α : Type*} [partial_order α]
variables x y z : α

#check x ≤ y
#check (le_refl x : x ≤ x)
#check (le_trans : x ≤ y → y ≤ z → x ≤ z)
```

Here we are adopting the mathlib convention of using letters like α, β, and γ (entered as `\a`, `\b`, and `\g`) for arbitrary types. The library often uses letters like `R` and `G` for the carries of algebraic structures like rings and groups, respectively, but in general Greek letters are used for types, especially when there is little or no structure associated with them.

Associated to any partial order, $\leq$, there is also a *strict partial order*, $<$, which acts somewhat like $<$ on the real numbers. Saying that `x` is less than `y` in this order is equivalent to saying that it is less-than-or-equal to `y` and not equal to `y`.

```
#check x < y
#check (lt_irrefl x : ¬ x < x)
#check (lt_trans : x < y → y < z → x < z)
#check (lt_of_le_of_lt : x ≤ y → y < z → x < z)
#check (lt_of_lt_of_le : x < y → y ≤ z → x < z)

example : x < y ↔ x ≤ y ∧ x ≠ y :=
lt_iff_le_and_ne
```

In this example, the symbol $\wedge$ stands for "and," the symbol $\neg$ stands for "not," and `x ≠ y` abbreviates `¬ (x = y)`. In Chapter 3, you will learn how to use these logical connectives to *prove* that $<$ has the properties indicated.

A *lattice* is a structure that extends a partial order with operations $\sqcap$ and $\sqcup$ that are analogous to `inf` and `max` on the real numbers:

```
variables {α : Type*} [lattice α]
variables x y z : α

#check x ⊓ y
#check (inf_le_left : x ⊓ y ≤ x)
#check (inf_le_right : x ⊓ y ≤ y)
#check (le_inf : z ≤ x → z ≤ y → z ≤ x ⊓ y)

#check x ⊔ y
#check (le_sup_left : x ≤ x ⊔ y)
#check (le_sup_right: y ≤ x ⊔ y)
#check (sup_le : x ≤ z → y ≤ z → x ⊔ y ≤ z)
```

The characterizations of ⊓ and ⊔ justify calling them the *greatest lower bound* and *least upper bound*, respectively. You can type them in VS code using `\glb` and `\lub`. The symbols are also often called then *infimum* and the *supremum*, and mathlib refers to them as `inf` and `sup` in theorem names. To further complicate matters, they are also often called *meet* and *join*. Therefore, if you work with lattices, you have to keep the following dictionary in infd:

- ⊓ is the *greatest lower bound*, *infimum*, or *meet*.
- ⊔ is the *least upper bound*, *supremum*, or *join*.

Some instances of lattices include:

- `inf` and `max` on any total order, such as the integers or real numbers with ≤
- ∩ and ∪ on the collection of subsets of some domain, with the ordering ⊆
- ∧ and ∨ on boolean truth values, with ordering `x ≤ y` if either `x` is false or `y` is true
- `gcd` and `lcm` on the natural numbers (or positive natural numbers), with the divisibility ordering, |
- the collection of linear subspaces of a vector space, where the greatest lower bound is given by the intersection, the least upper bound is given by the sum of the two spaces, and the ordering is inclusion
- the collection of topologies on a set (or, in Lean, a type), where the greatest lower bound of two topologies consists of the topology that is generated by their union, the least upper bound is their intersection, and the ordering is reverse inclusion

You can check that, as with `inf` / `max` and `gcd` / `lcm`, you can prove the commutativity and associativity of the infimum and supremum using only their characterizing axioms, together with `le_refl` and `le_trans`.

```
example : x ⊓ y = y ⊓ x := sorry
example : x ⊓ y ⊓ z = x ⊓ (y ⊓ z) := sorry
example : x ⊔ y = y ⊔ x := sorry
example : x ⊔ y ⊔ z = x ⊔ (y ⊔ z) := sorry
```

You can find these theorems in the mathlib as `inf_comm`, `inf_assoc`, `sup_comm`, and `sup_assoc`, respectively.

Another good exercise is to prove the *absorption laws* using only those axioms:

```
theorem absorb1 : x ⊓ (x ⊔ y) = x := sorry
theorem absorb2 : x ⊔ (x ⊓ y) = x := sorry
```

These can be found in mathlib with the names `inf_sup_self` and `sup_inf_self`.

A lattice that satisfies the additional identities `x ⊓ (y ⊔ z) = (x ⊓ y) ⊔ (x ⊓ z)` and `x ⊔ (y ⊓ z) = (x ⊔ y) ⊓ (x ⊔ z)` is called a *distributive lattice*. Lean knows about these too:

```
variables {α : Type*} [distrib_lattice α]
variables x y z : α

#check (inf_sup_left : x ⊓ (y ⊔ z) = (x ⊓ y) ⊔ (x ⊓ z))
#check (inf_sup_right : (x ⊔ y) ⊓ z = (x ⊓ z) ⊔ (y ⊓ z))
#check (sup_inf_left : x ⊔ (y ⊓ z) = (x ⊔ y) ⊓ (x ⊔ z))
#check (sup_inf_right : (x ⊓ y) ⊔ z = (x ⊔ z) ⊓ (y ⊔ z))
```

The left and right versions are easily shown to be equivalent, given the commutativity of ⊓ and ⊔. It is a good exercise to show that not every lattice is distributive by providing an explicit description of a nondistributive lattice with finitely many elements. It is also a good exercise to show that in any lattice, either distributivity law implies the other:

```
variables {α : Type*} [lattice α]
variables a b c : α
```

(continues on next page)

(continued from previous page)

```
example (h : ∀ x y z : α, x ⊓ (y ⊔ z) = (x ⊓ y) ⊔ (x ⊓ z)) :
  a ⊔ (b ⊓ c) = (a ⊔ b) ⊓ (a ⊔ c) :=
sorry

example (h : ∀ x y z : α, x ⊔ (y ⊓ z) = (x ⊔ y) ⊓ (x ⊔ z)) :
  a ⊓ (b ⊔ c) = (a ⊓ b) ⊔ (a ⊓ c) :=
sorry
```

It is possible to combine axiomatic structures into larger ones. For example, an *ordered ring* consists of a commutative ring together with a partial order on the carrier satisfying additional axioms that say that the ring operations are compatible with the order:

```
variables {R : Type*} [ordered_ring R]
variables a b c : R

#check (add_le_add_left : a ≤ b → ∀ c, c + a ≤ c + b)
#check (mul_pos : 0 < a → 0 < b → 0 < a * b)
```

Chapter 3 will provide the means to derive the following from `mul_pos` and the definition of <:

```
#check (mul_nonneg : 0 ≤ a → 0 ≤ b → 0 ≤ a * b)
```

It is then an extended exercise to show that many common facts used to reason about arithmetic and the ordering on the real numbers hold generically for any ordered ring. Here are a couple of examples you can try, using only properties of rings, partial orders, and the facts enumerated in the last two examples:

```
example : a ≤ b → 0 ≤ b - a := sorry

example : 0 ≤ b - a → a ≤ b := sorry

example (h : a ≤ b) (h' : 0 ≤ c) : a * c ≤ b * c := sorry
```

Finally, here is one last example. A *metric space* consists of a set equipped with a notion of distance, `dist x y`, mapping any pair of elements to a real number. The distance function is assumed to satisfy the following axioms:

```
variables {X : Type*} [metric_space X]
variables x y z : X

#check (dist_self x : dist x x = 0)
#check (dist_comm x y : dist x y = dist y x)
#check (dist_triangle x y z : dist x z ≤ dist x y + dist y z)
```

Having mastered this section, you can show that it follows from these axioms that distances are always nonnegative:

```
example (x y : X) : 0 ≤ dist x y := sorry
```

We recommend making use of the theorem `nonneg_of_mul_nonneg_left`. As you may have guessed, this theorem is called `dist_nonneg` in mathlib.

CHAPTER

THREE

LOGIC

In the last chapter, we dealt with equations, inequalities, and basic mathematical statements like "x divides y." Complex mathematical statements are built up from simple ones like these using logical terms like "and," "or," "not," and "if … then," "every," and "some." In this chapter, we show you how to work with statements that are built up in this way.

3.1 Implication and the Universal Quantifier

Consider the statement after the `#check`:

```
#check ∀ x : ℝ, 0 ≤ x → abs x = x
```

In words, we would say "for every real number `x`, if `0 ≤ x` then the absolute value of `x` equals `x`". We can also have more complicated statements like:

```
#check ∀ x y ε : ℝ, 0 < ε → ε ≤ 1 → abs x < ε → abs y < ε → abs (x * y) < ε
```

In words, we would say "for every `x`, `y`, and ε, if `0 < ε ≤ 1`, the absolute value of `x` is less than ε, and the absolute value of `y` is less than ε, then the absolute value of `x * y` is less than ε." In Lean, in a sequence of implications there are implicit parentheses grouped to the right. So the expression above means "if `0 < ε` then if `ε ≤ 1` then if `abs x < ε` …" As a result, the expression says that all the assumptions together imply the conclusion.

You have already seen that even though the universal quantifier in this statement ranges over objects and the implication arrows introduce hypotheses, Lean treats the two in very similar ways. In particular, if you have proved a theorem of that form, you can apply it to objects and hypotheses in the same way:

```
lemma my_lemma : ∀ x y ε : ℝ,
  0 < ε → ε ≤ 1 → abs x < ε → abs y < ε → abs (x * y) < ε :=
sorry

section
  variables a b δ : ℝ
  variables (h₀ : 0 < δ) (h₁ : δ ≤ 1)
  variables (ha : abs a < δ) (hb : abs b < δ)

  #check my_lemma a b δ
  #check my_lemma a b δ h₀ h₁
  #check my_lemma a b δ h₀ h₁ ha hb
end
```

You have also already seen that it is common in Lean to use curly brackets to make quantified variables implicit when they can be inferred from subsequent hypotheses. When we do that, we can just apply a lemma to the hypotheses without mentioning the objects.

```
lemma my_lemma2 : ∀ {x y ε : ℝ},
  0 < ε → ε ≤ 1 → abs x < ε → abs y < ε → abs (x * y) < ε :=
sorry

section
  variables a b δ : ℝ
  variables (h₀ : 0 < δ) (h₁ : δ ≤ 1)
  variables (ha : abs a < δ) (hb : abs b < δ)

  #check my_lemma2 h₀ h₁ ha hb
end
```

At this stage, you also know that if you use the `apply` tactic to apply `my_lemma` to a goal of the form `abs (a * b) < δ`, you are left with new goals that require you to prove each of the hypotheses.

To prove a statement like this, use the `intros` tactic. Take a look at what it does in this example:

```
lemma my_lemma3 : ∀ {x y ε : ℝ},
  0 < ε → ε ≤ 1 → abs x < ε → abs y < ε → abs (x * y) < ε :=
begin
  intros x y ε epos ele1 xlt ylt,
  sorry
end
```

We can use any names we want for the universally quantified variables; they do not have to be `x`, `y`, and ε. Notice that we have to introduce the variables even though they are marked implicit: making them implicit means that we leave them out when we write an expression *using* `my_lemma`, but they are still an essential part of the statement that we are proving. After the `intros` command, the goal is what it would have been at the start if we listed all the variables and hypotheses *before* the colon, as we did in the last section. In a moment, we will see why it is sometimes necessary to introduce variables and hypotheses after the proof begins.

To help you prove the lemma, we will start you off:

```
lemma my_lemma4 : ∀ {x y ε : ℝ},
  0 < ε → ε ≤ 1 → abs x < ε → abs y < ε → abs (x * y) < ε :=
begin
  intros x y ε epos ele1 xlt ylt,
  calc
    abs (x * y) = abs x * abs y : sorry
    ... ≤ abs x * ε             : sorry
    ... < 1 * ε                 : sorry
    ... = ε                     : sorry
end
```

Finish the proof using the theorems `abs_mul`, `mul_le_mul`, `abs_nonneg`, `mul_lt_mul_right`, and `one_mul`. Remember that you can find theorems like these using tab completion. Remember also that you can use `.mp` and `.mpr` or `.1` and `.2` to extract the two directions of an if-and-only-if statement.

Universal quantifiers are often hidden in definitions, and Lean will unfold definitions to expose them when necessary. For example, let's define two predicates, `fn_ub f a` and `fn_lb f a`, where `f` is a function from the real numbers to the real numbers and `a` is a real number. The first says that `a` is an upper bound on the values of `f`, and the second says that `a` is a lower bound on the values of `f`.

```
def fn_ub (f : ℝ → ℝ) (a : ℝ) : Prop := ∀ x, f x ≤ a
def fn_lb (f : ℝ → ℝ) (a : ℝ) : Prop := ∀ x, a ≤ f x
```

In the next example, `λ x, f x + g x` is a name for the function that maps `x` to `f x + g x`. Computer scientists refer to this as "lambda abstraction," whereas a mathematician might describe it as the function $x \mapsto f(x) + g(x)$.

```
example (hfa : fn_ub f a) (hgb : fn_ub g b) :
  fn_ub (λ x, f x + g x) (a + b) :=
begin
  intro x,
  dsimp,
  apply add_le_add,
  apply hfa,
  apply hgb
end
```

Applying `intro` to the goal `fn_ub (λ x, f x + g x) (a + b)` forces Lean to unfold the definition of `fn_ub` and introduce `x` for the universal quantifier. The goal is then `(λ (x : ℝ), f x + g x) x ≤ a + b`. But applying `(λ x, f x + g x)` to `x` should result in `f x + g x`, and the `dsimp` command performs that simplification. (The "d" stands for "definitional.") You can delete that command and the proof still works; Lean would have to perform that contraction anyhow to make sense of the next `apply`. The `dsimp` command simply makes the goal more readable and helps us figure out what to do next. Another option is to use the `change` tactic by writing `change f x + g x ≤ a + b`. This helps make the proof more readable, and gives you more control over how the goal is transformed.

The rest of the proof is routine. The last two `apply` commands force Lean to unfold the definitions of `fn_ub` in the hypotheses. Try carrying out similar proofs of these:

```
example (hfa : fn_lb f a) (hgb : fn_lb g b) :
  fn_lb (λ x, f x + g x) (a + b) :=
sorry

example (nnf : fn_lb f 0) (nng : fn_lb g 0) :
  fn_lb (λ x, f x * g x) 0 :=
sorry

example (hfa : fn_ub f a) (hfb : fn_ub g b)
    (nng : fn_lb g 0) (nna : 0 ≤ a) :
  fn_ub (λ x, f x * g x) (a * b) :=
sorry
```

Even though we have defined `fn_ub` and `fn_lb` for functions from the reals to the reals, you should recognize that the definitions and proofs are much more general. The definitions make sense for functions between any two types for which there is a notion of order on the codomain. Checking the type of the theorem `add_le_add` shows that it holds of any structure that is an "ordered additive commutative monoid"; the details of what that means don't matter now, but it is worth knowing that the natural numbers, integers, rationals, and real numbers are all instances. So if we prove the theorem `fn_ub_add` at that level of generality, it will apply in all these instances.

```
variables {α : Type*} {R : Type*} [ordered_cancel_add_comm_monoid R]

#check @add_le_add

def fn_ub' (f : α → R) (a : R) : Prop := ∀ x, f x ≤ a

theorem fn_ub_add {f g : α → R} {a b : R}
    (hfa : fn_ub' f a) (hgb : fn_ub' g b) :
  fn_ub' (λ x, f x + g x) (a + b) :=
λ x, add_le_add (hfa x) (hgb x)
```

You have already seen square brackets like these in Section Section 2.2, though we still haven't explained what they mean. For concreteness, we will stick to the real numbers for most of our examples, but it is worth knowing that mathlib contains definitions and theorems that work at a high level of generality.

For another example of a hidden universal quantifier, mathlib defines a predicate `monotone`, which says that a function is nondecreasing in its arguments:

```
example (f : ℝ → ℝ) (h : monotone f) :
  ∀ {a b}, a ≤ b → f a ≤ f b := h
```

Proving statements about monotonicity involves using `intros` to introduce two variables, say, `a` and `b`, and the hypothesis `a ≤ b`. To *use* a monotonicity hypothesis, you can apply it to suitable arguments and hypotheses, and then apply the resulting expression to the goal. Or you can apply it to the goal and let Lean help you work backwards by displaying the remaining hypotheses as new subgoals.

```
example (mf : monotone f) (mg : monotone g) :
  monotone (λ x, f x + g x) :=
begin
  intros a b aleb,
  apply add_le_add,
  apply mf aleb,
  apply mg aleb
end
```

When a proof is this short, it is often convenient to give a proof term instead. To describe a proof that temporarily introduces objects `a` and `b` and a hypothesis `aleb`, Lean uses the notation `λ a b aleb, ...`. This is analogous to the way that a lambda abstraction like `λ x, x^2` describes a function by temporarily naming an object, `x`, and then using it to describe a value. So the `intros` command in the previous proof corresponds to the lambda abstraction in the next proof term. The `apply` commands then correspond to building the application of the theorem to its arguments.

```
example (mf : monotone f) (mg : monotone g) :
  monotone (λ x, f x + g x) :=
λ a b aleb, add_le_add (mf aleb) (mg aleb)
```

Here is a useful trick: if you start writing the proof term `λ a b aleb, _` using an underscore where the rest of the expression should go, Lean will flag an error, indicating that it can't guess the value of that expression. If you check the Lean Goal window in VS Code or hover over the squiggly error marker, Lean will show you the goal that the remaining expression has to solve.

Try proving these, with either tactics or proof terms:

```
example {c : ℝ} (mf : monotone f) (nnc : 0 ≤ c) :
  monotone (λ x, c * f x) :=
sorry

example (mf : monotone f) (mg : monotone g) :
  monotone (λ x, f (g x)) :=
sorry
```

Here are some more examples. A function f from $\mathbb{R}$ to $\mathbb{R}$ is said to be *even* if $f(-x) = f(x)$ for every x, and *odd* if $f(-x) = -f(x)$ for every x. The following example defines these two notions formally and establishes one fact about them. You can complete the proofs of the others.

```
def fn_even (f : ℝ → ℝ) : Prop := ∀ x, f x = f (-x)
def fn_odd (f : ℝ → ℝ) : Prop := ∀ x, f x = - f (-x)

example (ef : fn_even f) (eg : fn_even g) : fn_even (λ x, f x + g x) :=
begin
  intro x,
  calc
    (λ x, f x + g x) x = f x + g x       : rfl
```

(continues on next page)

(continued from previous page)

```
                    ... = f (-x) + g (-x) : by rw [ef, eg]
end

example (of : fn_odd f) (og : fn_odd g) : fn_even (λ x, f x * g x) :=
sorry

example (ef : fn_even f) (og : fn_odd g) : fn_odd (λ x, f x * g x) :=
sorry

example (ef : fn_even f) (og : fn_odd g) : fn_even (λ x, f (g x)) :=
sorry
```

The first proof can be shortened using `dsimp` or `change` to get rid of the lambda. But you can check that the subsequent `rw` won't work unless we get rid of the lambda explicitly, because otherwise it cannot find the patterns `f x` and `g x` in the expression. Contrary to some other tactics, `rw` operates on the syntactic level, it won't unfold definitions or apply reductions for you (it has a variant called `erw` that tries a little harder in this direction, but not much harder).

You can find implicit universal quantifiers all over the place, once you know how to spot them. Mathlib includes a good library for rudimentary set theory. Lean's logical foundation imposes the restriction that when we talk about sets, we are always talking about sets of elements of some type. If `x` has type α and `s` has type `set α`, then `x ∈ s` is a proposition that asserts that `x` is an element of `s`. If `s` and `t` are of type `set α`, then the subset relation `s ⊆ t` is defined to mean `∀ {x : α}, x ∈ s → x ∈ t`. The variable in the quantifier is marked implicit so that given `h : s ⊆ t` and `h' : x ∈ s`, we can write `h h'` as justification for `x ∈ t`. The following example provides a tactic proof and a proof term justifying the reflexivity of the subset relation, and asks you to do the same for transitivity.

```
variables {α : Type*} (r s t : set α)

example : s ⊆ s :=
by { intros x xs, exact xs }

theorem subset.refl : s ⊆ s := λ x xs, xs

theorem subset.trans : r ⊆ s → s ⊆ t → r ⊆ t :=
sorry
```

Just as we defined `fn_ub` for functions, we can define `set_ub s a` to mean that `a` is an upper bound on the set `s`, assuming `s` is a set of elements of some type that has an order associated with it. In the next example, we ask you to prove that if `a` is a bound on `s` and `a ≤ b`, then `b` is a bound on `s` as well.

```
variables {α : Type*} [partial_order α]
variables (s : set α) (a b : α)

def set_ub (s : set α) (a : α) := ∀ x, x ∈ s → x ≤ a

example (h : set_ub s a) (h' : a ≤ b) : set_ub s b :=
sorry
```

We close this section with one last important example. A function f is said to be *injective* if for every x_1 and x_2, if $f(x_1) = f(x_2)$ then $x_1 = x_2$. Mathlib defines `function.injective f` with `x₁` and `x₂` implicit. The next example shows that, on the real numbers, any function that adds a constant is injective. We then ask you to show that multiplication by a nonzero constant is also injective.

```
open function

example (c : ℝ) : injective (λ x, x + c) :=
begin
```

(continues on next page)

(continued from previous page)

```
  intros x₁ x₂ h',
  exact (add_left_inj c).mp h',
end

example {c : ℝ} (h : c ≠ 0) : injective (λ x, c * x) :=
sorry
```

Finally, show that the composition of two injective functions is injective:

```
variables {α : Type*} {β : Type*} {γ : Type*}
variables {g : β → γ} {f : α → β}

example (injg : injective g) (injf : injective f) :
  injective (λ x, g (f x)) :=
sorry
```

3.2 The Existential Quantifier

The existential quantifier, which can be entered as `\ex` in VS Code, is used to represent the phrase "there exists." The formal expression `∃ x : ℝ, 2 < x ∧ x < 3` in Lean says that there is a real number between 2 and 3. (We will discuss the conjunction symbol, ∧, in Section 3.4.) The canonical way to prove such a statement is to exhibit a real number and show that it has the stated property. The number 2.5, which we can enter as `5 / 2` or `(5 : ℝ) / 2` when Lean cannot infer from context that we have the real numbers in mind, has the required property, and the `norm_num` tactic can prove that it meets the description.

There are a few ways we can put the information together. Given a goal that begins with an existential quantifier, the `use` tactic is used to provide the object, leaving the goal of proving the property.

```
example : ∃ x : ℝ, 2 < x ∧ x < 3 :=
begin
  use 5 / 2,
  norm_num
end
```

Alternatively, we can use Lean's *anonyomous constructor* notation to construct the proof.

```
example : ∃ x : ℝ, 2 < x ∧ x < 3 :=
begin
  have h : 2 < (5 : ℝ) / 2 ∧ (5 : ℝ) / 2 < 3,
    by norm_num,
  exact ⟨5 / 2, h⟩
end
```

The left and right angle brackets, which can be entered as `\<` and `\>` respectively, tell Lean to put together the given data using whatever construction is appropriate for the current goal. We can use the notation without going first into tactic mode:

```
example : ∃ x : ℝ, 2 < x ∧ x < 3 :=
⟨5 / 2, by norm_num⟩
```

So now we know how to *prove* an exists statement. But how do we *use* one? If we know that there exists an object with a certain property, we should be able to give a name to an arbitrary one and reason about it. For example, remember the predicates `fn_ub f a` and `fn_lb f a` from the last section, which say that `a` is an upper bound or lower bound on `f`, respectively. We can use the existential quantifier to say that "`f` is bounded" without specifying the bound:

```
def fn_ub (f : ℝ → ℝ) (a : ℝ) : Prop := ∀ x, f x ≤ a
def fn_lb (f : ℝ → ℝ) (a : ℝ) : Prop := ∀ x, a ≤ f x

def fn_has_ub (f : ℝ → ℝ) := ∃ a, fn_ub f a
def fn_has_lb (f : ℝ → ℝ) := ∃ a, fn_lb f a
```

We can use the theorem `fn_ub_add` from the last section to prove that if `f` and `g` have upper bounds, then so does `λ x, f x + g x`.

```
variables {f g : ℝ → ℝ}

example (ubf : fn_has_ub f) (ubg : fn_has_ub g) :
  fn_has_ub (λ x, f x + g x) :=
begin
  cases ubf with a ubfa,
  cases ubg with b ubfb,
  use a + b,
  apply fn_ub_add ubfa ubfb
end
```

The `cases` tactic unpacks the information in the existential quantifier. Given the hypothesis `ubf` that there is an upper bound for `f`, `cases` adds a new variable for an upper bound to the context, together with the hypothesis that it has the given property. The `with` clause allows us to specify the names we want Lean to use. The goal is left unchanged; what *has* changed is that we can now use the new object and the new hypothesis to prove the goal. This is a common pattern in mathematics: we unpack objects whose existence is asserted or implied by some hypothesis, and then use it to establish the existence of something else.

Try using this pattern to establish the following. You might find it useful to turn some of the examples from the last section into named theorems, as we did with `fn_ub_add`, or you can insert the arguments directly into the proofs.

```
example (lbf : fn_has_lb f) (lbg : fn_has_lb g) :
  fn_has_lb (λ x, f x + g x) :=
sorry

example {c : ℝ} (ubf : fn_has_ub f) (h : c ≥ 0):
  fn_has_ub (λ x, c * f x) :=
sorry
```

The task of unpacking information in a hypothesis is so important that Lean and mathlib provide a number of ways to do it. A cousin of the `cases` tactic, `rcases`, is more flexible in that it allows us to unpack nested data. (The "r" stands for "recursive.") In the `with` clause for unpacking an existential quantifier, we name the object and the hypothesis by presenting them as a pattern `⟨a, h⟩` that `rcases` then tries to match. The `rintro` tactic (which can also be written `rintros`) is a combination of `intros` and `rcases`. These examples illustrate their use:

```
example (ubf : fn_has_ub f) (ubg : fn_has_ub g) :
  fn_has_ub (λ x, f x + g x) :=
begin
  rcases ubf with ⟨a, ubfa⟩,
  rcases ubg with ⟨b, ubfb⟩,
  exact ⟨a + b, fn_ub_add ubfa ubfb⟩
end

example : fn_has_ub f → fn_has_ub g →
  fn_has_ub (λ x, f x + g x) :=
begin
  rintros ⟨a, ubfa⟩ ⟨b, ubfb⟩,
```

(continues on next page)

(continued from previous page)

```
  exact ⟨a + b, fn_ub_add ubfa ubfb⟩
end
```

In fact, Lean also supports a pattern-matching lambda in expressions and proof terms:

```
example : fn_has_ub f → fn_has_ub g →
  fn_has_ub (λ x, f x + g x) :=
λ ⟨a, ubfa⟩ ⟨b, ubfb⟩, ⟨a + b, fn_ub_add ubfa ubfb⟩
```

These are power-user moves, and there is no harm in favoring the use of `cases` until you are more comfortable with the existential quantifier. But we will come to learn that all of these tools, including `cases`, use, and the anonymous constructors, are like Swiss army knives when it comes to theorem proving. They can be used for a wide range of purposes, not just for unpacking exists statements.

To illustrate one way that `rcases` can be used, we prove an old mathematical chestnut: if two integers `x` and `y` can each be written as a sum of two squares, then so can their product, `x * y`. In fact, the statement is true for any commutative ring, not just the integers. In the next example, `rcases` unpacks two existential quantifiers at once. We then provide the magic values needed to express `x * y` as a sum of squares as a list to the `use` statement, and we use `ring` to verify that they work.

```
variables {α : Type*} [comm_ring α]

def sum_of_squares (x : α) := ∃ a b, x = a^2 + b^2

theorem sum_of_squares_mul {x y : α}
    (sosx : sum_of_squares x) (sosy : sum_of_squares y) :
  sum_of_squares (x * y) :=
begin
  rcases sosx with ⟨a, b, xeq⟩,
  rcases sosy with ⟨c, d, yeq⟩,
  rw [xeq, yeq],
  use [a*c - b*d, a*d + b*c],
  ring
end
```

This proof doesn't provide much insight, but here is one way to motivate it. A *Gaussian integer* is a number of the form $a + bi$ where a and b are integers and $i = \sqrt{-1}$. The *norm* of the Gaussian integer $a + bi$ is, by definition, $a^2 + b^2$. So the norm of a Gaussian integer is a sum of squares, and any sum of squares can be expressed in this way. The theorem above reflects the fact that norm of a product of Gaussian integers is the product of their norms: if x is the norm of $a + bi$ and y in the norm of $c + di$, then xy is the norm of $(a + bi)(c + di)$. Our cryptic proof illustrates the fact that the proof that is easiest to formalize isn't always the most perspicuous one. In the chapters to come, we will provide you with the means to define the Gaussian integers and use them to provide an alternative proof.

The pattern of unpacking an equation inside an existential quantifier and then using it to rewrite an expression in the goal comes up often, so much so that the `rcases` tactic provides an abbreviation: if you use the keyword `rfl` in place of a new identifier, `rcases` does the rewriting automatically (this trick doesn't work with pattern-matching lambdas).

```
theorem sum_of_squares_mul' {x y : α}
    (sosx : sum_of_squares x) (sosy : sum_of_squares y) :
  sum_of_squares (x * y) :=
begin
  rcases sosx with ⟨a, b, rfl⟩,
  rcases sosy with ⟨c, d, rfl⟩,
  use [a*c - b*d, a*d + b*c],
  ring
end
```

As with the universal quantifier, you can find existential quantifiers hidden all over if you know how to spot them. For example, divisibility is implicitly an "exists" statement.

```
example (divab : a | b) (divbc : b | c) : a | c :=
begin
  cases divab with d beq,
  cases divbc with e ceq,
  rw [ceq, beq],
  use (d * e), ring
end
```

And once again, this provides a nice setting for using `rcases` with `rfl`. Try it out in the proof above. It feels pretty good!

Then try proving the following:

```
example (divab : a | b) (divac : a | c) : a | (b + c) :=
sorry
```

For another important example, a function $f : \alpha \to \beta$ is said to be *surjective* if for every y in the codomain, β, there is an x in the domain, α, such that $f(x) = y$. Notice that this statement includes both a universal and an existential quantifier, which explains why the next example makes use of both `intro` and `use`.

```
example {c : ℝ} : surjective (λ x, x + c) :=
begin
  intro x,
  use x - c,
  dsimp, ring
end
```

Try this example yourself:

```
example {c : ℝ} (h : c ≠ 0) : surjective (λ x, c * x) :=
sorry
```

index:: field_simp, tactic ; field_simp

At this point, it is worth mentioning that there is a tactic, *field_simp*, that will often clear denominators in a useful way. It can be used in conjunction with the `ring` tactic.

```
example (x y : ℝ) (h : x - y ≠ 0) : (x^2 - y^2) / (x - y) = x + y :=
by { field_simp [h], ring }
```

You can use the theorem `div_mul_cancel`. The next example uses a surjectivity hypothesis by applying it to a suitable value. Note that you can use `cases` with any expression, not just a hypothesis.

```
example {f : ℝ → ℝ} (h : surjective f) : ∃ x, (f x)^2 = 4 :=
begin
  cases h 2 with x hx,
  use x,
  rw hx,
  norm_num
end
```

See if you can use these methods to show that the composition of surjective functions is surjective.

```
variables {α : Type*} {β : Type*} {γ : Type*}
variables {g : β → γ} {f : α → β}
```

(continues on next page)

(continued from previous page)

```
example (surjg : surjective g) (surjf : surjective f) :
  surjective (λ x, g (f x)) :=
sorry
```

3.3 Negation

The symbol ¬ is meant to express negation, so `¬ x < y` says that `x` is not less than `y`, `¬ x = y` (or, equivalently, `x ≠ y`) says that `x` is not equal to `y`, and `¬ ∃ z, x < z ∧ z < y` says that there does not exist a `z` strictly between `x` and `y`. In Lean, the notation `¬ A` abbreviates `A → false`, which you can think of as saying that `A` implies a contradiction. Practically speaking, this means that you already know something about how to work with negations: you can prove `¬ A` by introducing a hypothesis `h : A` and proving `false`, and if you have `h : ¬ A` and `h' : A`, then applying `h` to `h'` yields `false`.

To illustrate, consider the irreflexivity principle `lt_irrefl` for a strict order, which says that we have `¬ a < a` for every `a`. The asymmetry principle `lt_asymm` says that we have `a < b → ¬ b < a`. Let's show that `lt_asymm` follows from `lt_irrefl`.

```
example (h : a < b) : ¬ b < a :=
begin
  intro h',
  have : a < a,
    from lt_trans h h',
  apply lt_irrefl a this
end
```

This example introduces a couple of new tricks. First, when you use `have` without providing a label, Lean uses the name `this`, providing a convenient way to refer back to it. Also, the `from` tactic is syntactic sugar for `exact`, providing a nice way to justify a `have` with an explicit proof term. But what you should really be paying attention to in this proof is the result of the `intro` tactic, which leaves a goal of `false`, and the fact that we eventually prove `false` by applying `lt_irrefl` to a proof of `a < a`.

Here is another example, which uses the predicate `fn_has_ub` defined in the last section, which says that a function has an upper bound.

```
example (h : ∀ a, ∃ x, f x > a) : ¬ fn_has_ub f :=
begin
  intros fnub,
  cases fnub with a fnuba,
  cases h a with x hx,
  have : f x ≤ a,
    from fnuba x,
  linarith
end
```

See if you can prove these in a similar way:

```
example (h : ∀ a, ∃ x, f x < a) : ¬ fn_has_lb f :=
sorry

example : ¬ fn_has_ub (λ x, x) :=
sorry
```

Mathlib offers a number of useful theorems for relating orders and negations:

```
#check (not_le_of_gt : a > b → ¬ a ≤ b)
#check (not_lt_of_ge : a ≥ b → ¬ a < b)
#check (lt_of_not_ge : ¬ a ≥ b → a < b)
#check (le_of_not_gt : ¬ a > b → a ≤ b)
```

Recall the predicate `monotone f`, which says that `f` is nondecreasing. Use some of the theorems just enumerated to prove the following:

```
example (h : monotone f) (h' : f a < f b) : a < b :=
sorry

example (h : a ≤ b) (h' : f b < f a) : ¬ monotone f :=
sorry
```

Remember that it is often convenient to use `linarith` when a goal follows from linear equations and inequalities that in the context.

We can show that the first example in the last snippet cannot be proved if we replace $<$ by $\leq$. Notice that we can prove the negation of a universally quantified statement by giving a counterexample. Complete the proof.

```
example :
  ¬ ∀ {f : ℝ → ℝ}, monotone f → ∀ {a b}, f a ≤ f b → a ≤ b :=
begin
  intro h,
  let f := λ x : ℝ, (0 : ℝ),
  have monof : monotone f,
  { sorry },
  have h' : f 1 ≤ f 0,
    from le_refl _,
  sorry
end
```

This example introduces the `let` tactic, which adds a *local definition* to the context. If you put the cursor after the `let` command, in the goal window you will see that the definition `f : ℝ → ℝ := λ (x : ℝ), 0` has been added to the context. Lean will unfold the definition of `f` when it has to. In particular, when we prove `f 1 ≤ f 0` with `le_refl`, Lean reduces `f 1` and `f 0` to `0`.

Use `le_of_not_gt` to prove the following:

```
example (x : ℝ) (h : ∀ ε > 0, x < ε) : x ≤ 0 :=
sorry
```

Implicit in many of the proofs we have just done is the fact that if `P` is any property, saying that there is nothing with property `P` is the same as saying that everything fails to have property `P`, and saying that not everything has property `P` is equivalent to saying that something fails to have property `P`. In other words, all four of the following implications are valid (but one of them cannot be proved with what we explained so far):

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

example (h : ¬ ∃ x, P x) : ∀ x, ¬ P x :=
sorry

example (h : ∀ x, ¬ P x) : ¬ ∃ x, P x :=
sorry

example (h : ¬ ∀ x, P x) : ∃ x, ¬ P x :=
sorry
```

(continues on next page)

(continued from previous page)

```
example (h : ∃ x, ¬ P x) : ¬ ∀ x, P x :=
sorry
```

The first, second, and fourth are straightforward to prove using the methods you have already seen. We encourage you to try it. The third is more difficult, however, because it concludes that an object exists from the fact that its nonexistence is contradictory. This is an instance of *classical* mathematical reasoning, and, in general, you have to declare your intention of using such reasoning by adding the command `open_locale classical` to your file. With that command, we can use proof by contradiction to prove the third implication as follows.

```
open_locale classical

example (h : ¬ ∀ x, P x) : ∃ x, ¬ P x :=
begin
  by_contradiction h',
  apply h,
  intro x,
  show P x,
  by_contradiction h'',
  exact h' ⟨x, h''⟩
end
```

Make sure you understand how this works. The `by_contradiction` tactic (also abbreviated to `by_contra`) allows us to prove a goal `Q` by assuming `¬ Q` and deriving a contradiction. In fact, it is equivalent to using the equivalence `not_not : ¬ ¬ Q ↔ Q`. Confirm that you can prove the forward direction of this equivalence using `by_contradiction`, while the reverse direction follows from the ordinary rules for negation.

```
example (h : ¬ ¬ Q) : Q :=
sorry

example (h : Q) : ¬ ¬ Q :=
sorry
```

Use proof by contradiction to establish the following, which is the converse of one of the implications we proved above. (Hint: use `intro` first.)

```
example (h : ¬ fn_has_ub f) : ∀ a, ∃ x, f x > a :=
sorry
```

It is often tedious to work with compound statements with a negation in front, and it is a common mathematical pattern to replace such statements with equivalent forms in which the negation has been pushed inward. To facilitate this, mathlib offers a `push_neg` tactic, which restates the goal in this way. The command `push_neg at h` restates the hypothesis `h`.

```
example (h : ¬ ∀ a, ∃ x, f x > a) : fn_has_ub f :=
begin
  push_neg at h,
  exact h
end

example (h : ¬ fn_has_ub f) : ∀ a, ∃ x, f x > a :=
begin
  simp only [fn_has_ub, fn_ub] at h,
  push_neg at h,
  exact h
end
```

In the second example, we use Lean's simplifier to expand the definitions of `fn_has_ub` and `fn_ub`. (We need to use `simp` rather than `rw` to expand `fn_ub`, because it appears in the scope of a quantifier.) You can verify that in the examples above with `¬ ∃ x, P x` and `¬ ∀ x, P x`, the `push_neg` tactic does the expected thing. Without even knowing how to use the conjunction symbol, you should be able to use `push_neg` to prove the following:

```
example (h : ¬ monotone f) : ∃ x y, x ≤ y ∧ f y < f x :=
sorry
```

Mathlib also has a tactic, `contrapose`, which transforms a goal `A → B` to `¬ B → ¬ A`. Similarly, given a goal of proving `B` from hypothesis `h : A`, `contrapose h` leaves you with a goal of proving `¬ A` from hypothesis `¬ B`. Using `contrapose!` instead of `contrapose` applies `push_neg` to the goal and the relevant hypothesis as well.

```
example (h : ¬ fn_has_ub f) : ∀ a, ∃ x, f x > a :=
begin
  contrapose! h,
  exact h
end

example (x : ℝ) (h : ∀ ε > 0, x ≤ ε) : x ≤ 0 :=
begin
  contrapose! h,
  use x / 2,
  split; linarith
end
```

We have not yet explained the `split` command or the use of the semicolon after it, but we will do that in the next section.

We close this section with the principle of *ex falso*, which says that anything follows from a contradiction. In Lean, this is represented by `false.elim`, which establishes `false → P` for any proposition `P`. This may seem like a strange principle, but it comes up fairly often. We often prove a theorem by splitting on cases, and sometimes we can show that one of the cases is contradictory. In that case, we need to assert that the contradiction establishes the goal so we can move on to the next one. (We will see instances of reasoning by cases in Section 3.5.)

Lean provides a number of ways of closing a goal once a contradiction has been reached.

```
example (h : 0 < 0) : a > 37 :=
begin
  exfalso,
  apply lt_irrefl 0 h
end

example (h : 0 < 0) : a > 37 :=
absurd h (lt_irrefl 0)

example (h : 0 < 0) : a > 37 :=
begin
  have h' : ¬ 0 < 0,
    from lt_irrefl 0,
  contradiction
end
```

The `exfalso` tactic replaces the current goal with the goal of proving `false`. Given `h : P` and `h' : ¬ P`, the term `absurd h h'` establishes any proposition. Finally, the `contradiction` tactic tries to close a goal by finding a contradiction in the hypotheses, such as a pair of the form `h : P` and `h' : ¬ P`. Of course, in this example, `linarith` also works.

3.4 Conjunction and Bi-implication

You have already seen that the conjunction symbol, ∧, is used to express "and." The `split` tactic allows you to prove a statement of the form `A ∧ B` by proving `A` and then proving `B`.

```
example {x y : ℝ} (h₀ : x ≤ y) (h₁ : ¬ y ≤ x) : x ≤ y ∧ x ≠ y :=
begin
  split,
  { assumption },
  intro h,
  apply h₁,
  rw h
end
```

In this example, the `assumption` tactic tells Lean to find an assumption that will solve the goal. Notice that the final `rw` finishes the goal by applying the reflexivity of ≤. The following are alternative ways of carrying out the previous examples using the anonymous constructor angle brackets. The first is a slick proof-term version of the previous proof, which drops into tactic mode at the keyword `by`.

```
example {x y : ℝ} (h₀ : x ≤ y) (h₁ : ¬ y ≤ x) : x ≤ y ∧ x ≠ y :=
⟨h₀, λ h, h₁ (by rw h)⟩

example {x y : ℝ} (h₀ : x ≤ y) (h₁ : ¬ y ≤ x) : x ≤ y ∧ x ≠ y :=
begin
  have h : x ≠ y,
  { contrapose! h₁,
    rw h₁ },
  exact ⟨h₀, h⟩
end
```

Using a conjunction instead of proving one involves unpacking the proofs of the two parts. You can uses the `cases` tactic for that, as well as `rcases`, `rintros`, or a pattern-matching lambda, all in a manner similar to the way they are used with the existential quantifier.

```
example {x y : ℝ} (h : x ≤ y ∧ x ≠ y) : ¬ y ≤ x :=
begin
  cases h with h₀ h₁,
  contrapose! h₁,
  exact le_antisymm h₀ h₁
end

example {x y : ℝ} : x ≤ y ∧ x ≠ y → ¬ y ≤ x :=
begin
  rintros ⟨h₀, h₁⟩ h',
  exact h₁ (le_antisymm h₀ h')
end

example {x y : ℝ} : x ≤ y ∧ x ≠ y → ¬ y ≤ x :=
λ ⟨h₀, h₁⟩ h', h₁ (le_antisymm h₀ h')
```

In contrast to using an existential quantifier, you can also extract proofs of the two components of a hypothesis `h : A ∧ B` by writing `h.left` and `h.right`, or, equivalently, `h.1` and `h.2`.

```
example {x y : ℝ} (h : x ≤ y ∧ x ≠ y) : ¬ y ≤ x :=
begin
  intro h',
```

(continues on next page)

(continued from previous page)

```
  apply h.right,
  exact le_antisymm h.left h'
end

example {x y : ℝ} (h : x ≤ y ∧ x ≠ y) : ¬ y ≤ x :=
λ h', h.right (le_antisymm h.left h')
```

Try using these techniques to come up with various ways of proving of the following:

```
example {m n : ℕ} (h : m ∣ n ∧ m ≠ n) :
  m ∣ n ∧ ¬ n ∣ m :=
sorry
```

You can nest uses of ∃ and ∧ with anonymous constructors, `rintros`, and `rcases`.

```
example : ∃ x : ℝ, 2 < x ∧ x < 4 :=
⟨5/2, by norm_num, by norm_num⟩

example (x y : ℝ) : (∃ z : ℝ, x < z ∧ z < y) → x < y :=
begin
  rintros ⟨z, xltz, zlty⟩,
  exact lt_trans xltz zlty
end

example (x y : ℝ) : (∃ z : ℝ, x < z ∧ z < y) → x < y :=
λ ⟨z, xltz, zlty⟩, lt_trans xltz zlty
```

You can also use the `use` tactic:

```
example : ∃ x : ℝ, 2 < x ∧ x < 4 :=
begin
  use 5 / 2,
  split; norm_num
end

example : ∃ m n : ℕ,
  4 < m ∧ m < n ∧ n < 10 ∧ nat.prime m ∧ nat.prime n :=
begin
  use [5, 7],
  norm_num
end

example {x y : ℝ} : x ≤ y ∧ x ≠ y → x ≤ y ∧ ¬ y ≤ x :=
begin
  rintros ⟨h₀, h₁⟩,
  use [h₀, λ h', h₁ (le_antisymm h₀ h')]
end
```

In the first example, the semicolon after the `split` command tells Lean to use the `norm_num` tactic on both of the goals that result.

In Lean, `A ↔ B` is *not* defined to be `(A → B) ∧ (B → A)`, but it could have been, and it behaves roughly the same way. You have already seen that you can write `h.mp` and `h.mpr` or `h.1` and `h.2` for the two directions of `h : A ↔ B`. You can also use `cases` and friends. To prove an if-and-only-if statement, you can uses `split` or angle brackets, just as you would if you were proving a conjunction.

```
example {x y : ℝ} (h : x ≤ y) : ¬ y ≤ x ↔ x ≠ y :=
begin
  split,
  { contrapose!,
    rintro rfl,
    reflexivity },
  contrapose!,
  exact le_antisymm h
end

example {x y : ℝ} (h : x ≤ y) : ¬ y ≤ x ↔ x ≠ y :=
⟨λ h₀ h₁, h₀ (by rw h₁), λ h₀ h₁, h₀ (le_antisymm h h₁)⟩
```

The last proof term is inscrutable. Remember that you can use underscores while writing an expression like that to see what Lean expects.

Try out the various techniques and gadgets you have just seen in order to prove the following:

```
example {x y : ℝ} : x ≤ y ∧ ¬ y ≤ x ↔ x ≤ y ∧ x ≠ y :=
sorry
```

For a more interesting exercise, show that for any two real numbers `x` and `y`, `x^2 + y^2 = 0` if and only if `x = 0` and `y = 0`. We suggest proving an auxiliary lemma using `linarith`, `pow_two_nonneg`, and `pow_eq_zero`.

```
theorem aux {x y : ℝ} (h : x^2 + y^2 = 0) : x = 0 :=
begin
  have h' : x^2 = 0,
  { sorry },
  exact pow_eq_zero h'
end

example (x y : ℝ) : x^2 + y^2 = 0 ↔ x = 0 ∧ y = 0 :=
sorry
```

In Lean, bi-implication leads a double-life. You can treat it like a conjunction and use its two parts separately. But Lean also knows that it is a reflexive, symmetric, and transitive relation between propositions, and you can also use it with `calc` and `rw`. It is often convenient to rewrite a statement to an equivalent one. In the next example, we use `abs_lt` to replace an expression of the form `abs x < y` by the equivalent expression `- y < x ∧ x < y`, and in the one after that we use `dvd_gcd_iff` to replace an expression of the form `m ∣ gcd n k` by the equivalent expression `m ∣ n ∧ m ∣ k`.

```
example (x y : ℝ) : abs (x + 3) < 5 → -8 < x ∧ x < 2 :=
begin
  rw abs_lt,
  intro h,
  split; linarith
end

example : 3 ∣ gcd 6 15 :=
begin
  rw dvd_gcd_iff,
  split; norm_num
end
```

See if you can use `rw` with the theorem below to provide a short proof that negation is not a nondecreasing function. (Note that `push_neg` won't unfold definitions for you, so the `rw monotone` in the proof of the theorem is needed.)

```
theorem not_monotone_iff {f : ℝ → ℝ}:
  ¬ monotone f ↔ ∃ x y, x ≤ y ∧ f x > f y :=
by { rw monotone, push_neg }

example : ¬ monotone (λ x : ℝ, -x) :=
sorry
```

The remaining exercises in this section are designed to give you some more practice with conjunction and bi-implication. Remember that a *partial order* is a binary relation that is transitive, reflexive, and antisymmetric. An even weaker notion sometimes arises: a *preorder* is just a reflexive, transitive relation. For any pre-order $\leq$, Lean axiomatizes the associated strict pre-order by `a < b ↔ a ≤ b ∧ ¬ b ≤ a`. Show that if $\leq$ is a partial order, then `a < b` is equivalent to `a ≤ b ∧ a ≠ b`:

```
variables {α : Type*} [partial_order α]
variables a b : α

example : a < b ↔ a ≤ b ∧ a ≠ b :=
begin
  rw lt_iff_le_not_le,
  sorry
end
```

Beyond logical operations, you should not need anything more than `le_refl` and `le_antisymm`. Then show that even in the case where $\leq$ is only assumed to be a preorder, we can prove that the strict order is irreflexive and transitive. You do not need anything more than `le_refl` and `le_trans`. In the second example, for convenience, we use the simplifier rather than `rw` to express $<$ in terms of $\leq$ and $\neg$. We will come back to the simplifier later, but here we are only relying on the fact that it will use the indicated lemma repeatedly, even if it needs to be instantiated to different values.

```
variables {α : Type*} [preorder α]
variables a b c : α

example : ¬ a < a :=
begin
  rw lt_iff_le_not_le,
  sorry
end

example : a < b → b < c → a < c :=
begin
  simp only [lt_iff_le_not_le],
  sorry
end
```

3.5 Disjunction

The canonical way to prove a disjunction `A ∨ B` is to prove `A` or to prove `B`. The `left` tactic chooses `A`, and the `right` tactic chooses `B`.

```
variables {x y : ℝ}

example (h : y > x^2) : y > 0 ∨ y < -1 :=
by { left, linarith [pow_two_nonneg x] }
```

(continues on next page)

(continued from previous page)

```
example (h : -y > x^2 + 1) : y > 0 ∨ y < -1 :=
by { right, linarith [pow_two_nonneg x] }
```

We cannot use an anonymous constructor to construct a proof of an "or" because Lean would have to guess which disjunct we are trying to prove. When we write proof terms we can use `or.inl` and `or.inr` instead to make the choice explicitly. Here, `inl` is short for "introduction left" and `inr` is short for "introduction right."

```
example (h : y > 0) : y > 0 ∨ y < -1 :=
or.inl h

example (h : y < -1) : y > 0 ∨ y < -1 :=
or.inr h
```

It may seem strange to prove a disjunction by proving one side or the other. In practice, which case holds usually depends a case distinction that is implicit or explicit in the assumptions and the data. The `cases` tactic allows us to make use of a hypothesis of the form `A ∨ B`. In contrast to the use of `cases` with conjunction or an existential quantifier, here the `cases` tactic produces *two* goals. Both have the same conclusion, but in the first case, `A` is assumed to be true, and in the second case, `B` is assumed to be true. In other words, as the name suggests, the `cases` tactic carries out a proof by cases. As usual, we can tell Lean what names to use for the hypotheses. In the next example, we tell Lean to use the name `h` on each branch.

```
example : x < abs y → x < y ∨ x < -y :=
begin
  cases le_or_gt 0 y with h h,
  { rw abs_of_nonneg h,
    intro h, left, exact h },
  rw abs_of_neg h,
  intro h, right, exact h
end
```

The absolute value function is defined in such a way that we can immediately prove that `x ≥ 0` implies `abs x = x` (this is the theorem `abs_of_nonneg`) and `x < 0` implies `abs x = -x` (this is `abs_of_neg`). The expression `le_or_gt 0 x` establishes `0 ≤ x ∨ x < 0`, allowing us to split on those two cases. Try proving the triangle inequality using the two first two theorems in the next snippet. They are given the same names they have in mathlib.

```
namespace my_abs

theorem le_abs_self (x : ℝ) : x ≤ abs x :=
sorry

theorem neg_le_abs_self (x : ℝ) : -x ≤ abs x :=
sorry

theorem abs_add (x y : ℝ) : abs (x + y) ≤ abs x + abs y :=
sorry
```

In case you enjoyed these (pun intended) and you want more practice with disjunction, try these.

```
theorem lt_abs : x < abs y ↔ x < y ∨ x < -y :=
sorry

theorem abs_lt : abs x < y ↔ - y < x ∧ x < y :=
sorry
```

You can also use `rcases` and `rintros` with disjunctions. When these result in a genuine case split with multiple goals, the patterns for each new goal are separated by a vertical bar.

```
example {x : ℝ} (h : x ≠ 0) : x < 0 ∨ x > 0 :=
begin
  rcases lt_trichotomy x 0 with xlt | xeq | xgt,
  { left, exact xlt },
  { contradiction },
  right, exact xgt
end
```

You can still nest patterns and use the `rfl` keyword to substitute equations:

```
example {m n k : ℕ} (h : m ∣ n ∨ m ∣ k) : m ∣ n * k :=
begin
  rcases h with ⟨a, rfl⟩ | ⟨b, rfl⟩,
  { rw [mul_assoc],
    apply dvd_mul_right },
  rw [mul_comm, mul_assoc],
  apply dvd_mul_right
end
```

See if you can prove the following with a single (long) line. Use `rcases` to unpack the hypotheses and split on cases, and use a semicolon and `linarith` to solve each branch.

```
example {z : ℝ} (h : ∃ x y, z = x^2 + y^2 ∨ z = x^2 + y^2 + 1) :
  z ≥ 0 :=
sorry
```

On the real numbers, an equation `x * y = 0` tells us that `x = 0` or `y = 0`. In mathlib, this fact is known as `eq_zero_or_eq_zero_of_mul_eq_zero`, and it is another nice example of how a disjunction can arise. See if you can use it to prove the following:

```
example {x : ℝ} (h : x^2 = 1) : x = 1 ∨ x = -1 :=
sorry

example {x y : ℝ} (h : x^2 = y^2) : x = y ∨ x = -y :=
sorry
```

Remember that you can use the `ring` tactic to help with calculations.

In an arbitrary ring R, an element x such that $xy = 0$ for some nonzero y is called a *left zero divisor*, an element x such that $yx = 0$ for some nonzero y is called a *right zero divisor*, and an element that is either a left or right zero divisor is called simply a *zero divisor*. The theorem `eq_zero_or_eq_zero_of_mul_eq_zero` says that the real numbers have no nontrivial zero divisors. A commutative ring with this property is called an *integral domain*. Your proofs of the two theorems above should work equally well in any integral domain:

```
variables {R : Type*} [comm_ring R] [is_domain R]
variables (x y : R)

example (h : x^2 = 1) : x = 1 ∨ x = -1 :=
sorry

example (h : x^2 = y^2) : x = y ∨ x = -y :=
sorry
```

In fact, if you are careful, you can prove the first theorem without using commutativity of multiplication. In that case, it suffices to assume that `R` is a `domain` instead of an `integral_domain`.

Sometimes in a proof we want to split on cases depending on whether some statement is true or not. For any proposition `P`, we can use `classical.em P : P ∨ ¬ P`. The name `em` is short for "excluded middle."

```
example (P : Prop) : ¬ ¬ P → P :=
begin
  intro h,
  cases classical.em P,
  { assumption },
  contradiction
end
```

You can shorten `classical.em` to `em` by opening the `classical` namespace with the command `open classical`. Alternatively, you can use the `by_cases` tactic. The `open_locale classical` command guarantees that Lean can make implicit use of the law of the excluded middle.

```
open_locale classical

example (P : Prop) : ¬ ¬ P → P :=
begin
  intro h,
  by_cases h' : P,
  { assumption },
  contradiction
end
```

Notice that the `by_cases` tactic lets you specify a label for the hypothesis that is introduced in each branch, in this case, `h' : P` in one and `h' : ¬ P` in the other. If you leave out the label, Lean uses `h` by default. Try proving the following equivalence, using `by_cases` to establish one direction.

```
example (P Q : Prop) : (P → Q) ↔ ¬ P ∨ Q :=
sorry
```

3.6 Sequences and Convergence

We now have enough skills at our disposal to do some real mathematics. In Lean, we can represent a sequence $s_0, s_1, s_2, \ldots$ of real numbers as a function `s : ℕ → ℝ`. Such a sequence is said to *converge* to a number a if for every $\varepsilon > 0$ there is a point beyond which the sequence remains within ε of a, that is, there is a number N such that for every $n \geq N$, $|s_n - a| < \varepsilon$. In Lean, we can render this as follows:

```
def converges_to (s : ℕ → ℝ) (a : ℝ) :=
∀ ε > 0, ∃ N, ∀ n ≥ N, abs (s n - a) < ε
```

The notation `∀ ε > 0, ...` is a convenient abbreviation for `∀ ε, ε > 0 → ...`, and, similarly, `∀ n ≥ N, ...` abbreviates `∀ n, n ≥ N → ...`. And remember that `ε > 0`, in turn, is defined as `0 < ε`, and `n ≥ N` is defined as `N ≤ n`.

In this section, we'll establish some properties of convergence. But first, we will discuss three tactics for working with equality that will prove useful. The first, the `ext` tactic, gives us a way of proving that two functions are equal. Let $f(x) = x + 1$ and $g(x) = 1 + x$ be functions from reals to reals. Then, of course, $f = g$, because they return the same value for every x. The `ext` tactic enables us to prove an equation between functions by proving that their values are the same at all the values of their arguments.

```
example : (λ x y : ℝ, (x + y)^2) = (λ x y : ℝ, x^2 + 2*x*y + y^2) :=
by { ext, ring }
```

We'll see later that `ext` is actually more general, and also one can specify the name of the variables that appear. For instance you can try to replace `ext` with `ext u v` in the above proof. The second tactic, the `congr` tactic, allows us

to prove an equation between two expressions by reconciling the parts that are different:

```
example (a b : ℝ) : abs a = abs (a - b + b) :=
by  { congr, ring }
```

Here the `congr` tactic peels off the `abs` on each side, leaving us to prove `a = a - b + b`.

Finally, the `convert` tactic is used to apply a theorem to a goal when the conclusion of the theorem doesn't quite match. For example, suppose we want to prove `a < a * a` from `1 < a`. A theorem in the library, `mul_lt_mul_right`, will let us prove `1 * a < a * a`. One possibility is to work backwards and rewrite the goal so that it has that form. Instead, the `convert` tactic lets us apply the theorem as it is, and leaves us with the task of proving the equations that are needed to make the goal match.

```
example {a : ℝ} (h : 1 < a) : a < a * a :=
begin
  convert (mul_lt_mul_right _).2 h,
  { rw [one_mul] },
  exact lt_trans zero_lt_one h
end
```

This example illustrates another useful trick: when we apply an expression with an underscore and Lean can't fill it in for us automatically, it simply leaves it for us as another goal.

The following shows that any constant sequence $a, a, a, \ldots$ converges.

```
theorem converges_to_const (a : ℝ) : converges_to (λ x : ℕ, a) a :=
begin
  intros ε εpos,
  use 0,
  intros n nge, dsimp,
  rw [sub_self, abs_zero],
  apply εpos
end
```

Lean has a tactic, `simp`, which can often save you the trouble of carrying out steps like `rw [sub_self, abs_zero]` by hand. We will tell you more about it soon.

For a more interesting theorem, let's show that if `s` converges to `a` and `t` converges to `b`, then `λ n, s n + t n` converges to `a + b`. It is helpful to have a clear pen-and-paper proof in mind before you start writing a formal one. Given ε greater than `0`, the idea is to use the hypotheses to obtain an `Ns` such that beyond that point, `s` is within ε `/ 2` of `a`, and an `Nt` such that beyond that point, `t` is within ε `/ 2` of `b`. Then, whenever `n` is greater than or equal to the maximum of `Ns` and `Nt`, the sequence `λ n, s n + t n` should be within ε of `a + b`. The following example begins to implement this strategy. See if you can finish it off.

```
theorem converges_to_add {s t : ℕ → ℝ} {a b : ℝ}
  (cs : converges_to s a) (ct : converges_to t b):
converges_to (λ n, s n + t n) (a + b) :=
begin
  intros ε εpos, dsimp,
  have ε2pos : 0 < ε / 2,
  { linarith },
  cases cs (ε / 2) ε2pos with Ns hs,
  cases ct (ε / 2) ε2pos with Nt ht,
  use max Ns Nt,
  sorry
end
```

As hints, you can use `le_of_max_le_left` and `le_of_max_le_right`, and `norm_num` can prove ε `/ 2 +` ε `/ 2 =` ε. Also, it is helpful to use the `congr` tactic to show that `abs (s n + t n - (a + b))` is equal to

`abs ((s n - a) + (t n - b))`, since then you can use the triangle inequality. Notice that we marked all the variables `s`, `t`, `a`, and `b` implicit because they can be inferred from the hypotheses.

Proving the same theorem with multiplication in place of addition is tricky. We will get there by proving some auxiliary statements first. See if you can also finish off the next proof, which shows that if `s` converges to `a`, then `λ n, c * s n` converges to `c * a`. It is helpful to split into cases depending on whether `c` is equal to zero or not. We have taken care of the zero case, and we have left you to prove the result with the extra assumption that `c` is nonzero.

```
theorem converges_to_mul_const {s : ℕ → ℝ} {a : ℝ}
    (c : ℝ) (cs : converges_to s a) :
  converges_to (λ n, c * s n) (c * a) :=
begin
  by_cases h : c = 0,
  { convert converges_to_const 0,
    { ext, rw [h, zero_mul] },
    rw [h, zero_mul] },
  have acpos : 0 < abs c,
    from abs_pos.mpr h,
  sorry
end
```

The next theorem is also independently interesting: it shows that a convergent sequence is eventually bounded in absolute value. We have started you off; see if you can finish it.

```
theorem exists_abs_le_of_converges_to {s : ℕ → ℝ} {a : ℝ}
    (cs : converges_to s a) :
  ∃ N b, ∀ n, N ≤ n → abs (s n) < b :=
begin
  cases cs 1 zero_lt_one with N h,
  use [N, abs a + 1],
  sorry
end
```

In fact, the theorem could be strengthened to assert that there is a bound `b` that holds for all values of `n`. But this version is strong enough for our purposes, and we will see at the end of this section that it holds more generally.

The next lemma is auxiliary: we prove that if `s` converges to `a` and `t` converges to `0`, then `λ n, s n * t n` converges to `0`. To do so, we use the previous theorem to find a `B` that bounds `s` beyond some point N_0. See if you can understand the strategy we have outlined and finish the proof.

```
lemma aux {s t : ℕ → ℝ} {a : ℝ}
    (cs : converges_to s a) (ct : converges_to t 0) :
  converges_to (λ n, s n * t n) 0 :=
begin
  intros ε εpos, dsimp,
  rcases exists_abs_le_of_converges_to cs with ⟨N₀, B, h₀⟩,
  have Bpos : 0 < B,
    from lt_of_le_of_lt (abs_nonneg _) (h₀ N₀ (le_refl _)),
  have pos₀ : ε / B > 0,
    from div_pos εpos Bpos,
  cases ct _ pos₀ with N₁ h₁,
  sorry
end
```

If you have made it this far, congratulations! We are now within striking distance of our theorem. The following proof finishes it off.

```
theorem converges_to_mul {s t : ℕ → ℝ} {a b : ℝ}
    (cs : converges_to s a) (ct : converges_to t b):
  converges_to (λ n, s n * t n) (a * b) :=
begin
  have h₁ : converges_to (λ n, s n * (t n - b)) 0,
  { apply aux cs,
    convert converges_to_add ct (converges_to_const (-b)),
    ring },
  convert (converges_to_add h₁ (converges_to_mul_const b cs)),
  { ext, ring },
  ring
end
```

For another challenging exercise, try filling out the following sketch of a proof that limits are unique. (If you are feeling bold, you can delete the proof sketch and try proving it from scratch.)

```
theorem converges_to_unique {s : ℕ → ℝ} {a b : ℝ}
    (sa : converges_to s a) (sb : converges_to s b) :
  a = b :=
begin
  by_contradiction abne,
  have : abs (a - b) > 0,
  { sorry },
  let ε := abs (a - b) / 2,
  have εpos : ε > 0,
  { change abs (a - b) / 2 > 0, linarith },
  cases sa ε εpos with Na hNa,
  cases sb ε εpos with Nb hNb,
  let N := max Na Nb,
  have absa : abs (s N - a) < ε,
  { sorry },
  have absb : abs (s N - b) < ε,
  { sorry },
  have : abs (a - b) < abs (a - b),
  { sorry },
  exact lt_irrefl _ this
end
```

We close the section with the observation that our proofs can be generalized. For example, the only properties that we have used of the natural numbers is that their structure carries a partial order with `min` and `max`. You can check that everything still works if you replace $\mathbb{N}$ everywhere by any linear order α:

```
variables {α : Type*} [linear_order α]

def converges_to' (s : α → ℝ) (a : ℝ) :=
∀ ε > 0, ∃ N, ∀ n ≥ N, abs (s n - a) < ε
```

In a later chapter, we will see that mathlib has mechanisms for dealing with convergence in vastly more general terms, not only abstracting away particular features of the domain and codomain, but also abstracting over different types of convergence.

CHAPTER

FOUR

SETS AND FUNCTIONS

The vocabulary of sets, relations, and functions provides a uniform language for carrying out constructions in all the branches of mathematics. Since functions and relations can be defined in terms of sets, axiomatic set theory can be used as a foundation for mathematics.

Lean's foundation is based instead on the primitive notion of a *type*, and it includes ways of defining functions between types. Every expression in Lean has a type: there are natural numbers, real numbers, functions from reals to reals, groups, vector spaces, and so on. Some expressions *are* types, which is to say, their type is `Type`. Lean and mathlib provide ways of defining new types, and ways of defining objects of those types.

Conceptually, you can think of a type as just a set of objects. Requiring every object to have a type has some advantages. For example, it makes it possible to overload notation like +, and it sometimes makes input less verbose because Lean can infer a lot of information from an object's type. The type system also enables Lean to flag errors when you apply a function to the wrong number of arguments, or apply a function to arguments of the wrong type.

Lean's library does define elementary set-theoretic notions. In contrast to set theory, in Lean a set is always a set of objects of some type, such as a set natural numbers or a set of functions from real numbers to real numbers. The distinction between types and set takes some getting used to, but this chapter will take you through the essentials.

4.1 Sets

If α is any type, the type `set α` consists of sets of elements of α. This type supports the usual set-theoretic operations and relations. For example, `s ⊆ t` says that `s` is a subset of `t`, `s ∩ t` denotes the intersection of `s` and `t`, and `s ∪ t` denotes their union. The subset relation can be typed with `\ss` or `\sub`, intersection can be typed with `\i` or `\cap`, and union can be typed with `\un` or `\cup`. The library also defines the set `univ`, which consists of all the elements of type α, and the empty set, $\emptyset$, which can be typed as `\empty`. Given `x : α` and `s : set α`, the expression `x ∈ s` says that `x` is a member of `s`. Theorems that mention set membership often include `mem` in their name. The expression `x ∉ s` abbreviates `¬ x ∈ s`. You can type $\in$ as `\in` or `\mem` and $\notin$ as `\notin`.

One way to prove things about sets is to use `rw` or the simplifier to expand the definitions. In the second example below, we use `simp only` to tell the simplifier to use only the list of identities we give it, and not its full database of identities. Unlike `rw`, `simp` can perform simplifications inside a universal or existential quantifier. If you step through the proof, you can see the effects of these commands.

```
variable {α : Type*}
variables (s t u : set α)

open set

example (h : s ⊆ t) : s ∩ u ⊆ t ∩ u :=
begin
  rw [subset_def, inter_def, inter_def],
```

(continues on next page)

(continued from previous page)

```
  rw subset_def at h,
  dsimp,
  rintros x ⟨xs, xu⟩,
  exact ⟨h _ xs, xu⟩,
end

example (h : s ⊆ t) : s ∩ u ⊆ t ∩ u :=
begin
  simp only [subset_def, mem_inter_eq] at *,
  rintros x ⟨xs, xu⟩,
  exact ⟨h _ xs, xu⟩,
end
```

In this example, we open the `set` namespace to have access to the shorter names for the theorems. But, in fact, we can delete the calls to `rw` and `simp` entirely:

```
example (h : s ⊆ t) : s ∩ u ⊆ t ∩ u :=
begin
  intros x xsu,
  exact ⟨h xsu.1, xsu.2⟩
end
```

What is going on here is known as *definitional reduction*: to make sense of the `intros` command and the anonymous constructors Lean is forced to expand the definitions. The following examples also illustrate the phenomenon:

```
theorem foo (h : s ⊆ t) : s ∩ u ⊆ t ∩ u :=
λ x ⟨xs, xu⟩, ⟨h xs, xu⟩

example (h : s ⊆ t) : s ∩ u ⊆ t ∩ u :=
by exact λ x ⟨xs, xu⟩, ⟨h xs, xu⟩
```

Due to a quirk of how Lean processes its input, the first example fails if we replace `theorem foo` with `example`. This illustrates the pitfalls of relying on definitional reduction too heavily. It is often convenient, but sometimes we have to fall back on unfolding definitions manually.

To deal with unions, we can use `set.union_def` and `set.mem_union`. Since `x ∈ s ∪ t` unfolds to `x ∈ s ∨ x ∈ t`, we can also use the `cases` tactic to force a definitional reduction.

```
example : s ∩ (t ∪ u) ⊆ (s ∩ t) ∪ (s ∩ u) :=
begin
  intros x hx,
  have xs : x ∈ s := hx.1,
  have xtu : x ∈ t ∪ u := hx.2,
  cases xtu with xt xu,
  { left,
    show x ∈ s ∩ t,
    exact ⟨xs, xt⟩ },
  right,
  show x ∈ s ∩ u,
  exact ⟨xs, xu⟩
end
```

Since intersection binds tighter than union, the use of parentheses in the expression `(s ∩ t) ∪ (s ∩ u)` is unnecessary, but they make the meaning of the expression clearer. The following is a shorter proof of the same fact:

```
example : s ∩ (t ∪ u) ⊆ (s ∩ t) ∪ (s ∩ u) :=
begin
  rintros x ⟨xs, xt | xu⟩,
  { left, exact ⟨xs, xt⟩ },
  right, exact ⟨xs, xu⟩
end
```

As an exercise, try proving the other inclusion:

```
example : (s ∩ t) ∪ (s ∩ u) ⊆ s ∩ (t ∪ u):=
sorry
```

It might help to know that when using `rintros`, sometimes we need to use parentheses around a disjunctive pattern `h1 | h2` to get Lean to parse it correctly.

The library also defines set difference, `s \ t`, where the backslash is a special unicode character entered as `\\`. The expression `x ∈ s \ t` expands to `x ∈ s ∧ x ∉ t`. (The ∉ can be entered as `\notin`.) It can be rewritten manually using `set.diff_eq` and `dsimp` or `set.mem_diff`, but the following two proofs of the same inclusion show how to avoid using them.

```
example : s \ t \ u ⊆ s \ (t ∪ u) :=
begin
  intros x xstu,
  have xs : x ∈ s := xstu.1.1,
  have xnt : x ∉ t := xstu.1.2,
  have xnu : x ∉ u := xstu.2,
  split,
  { exact xs }, dsimp,
  intro xtu, -- x ∈ t ∨ x ∈ u
  cases xtu with xt xu,
  { show false, from xnt xt },
  show false, from xnu xu
end

example : s \ t \ u ⊆ s \ (t ∪ u) :=
begin
  rintros x ⟨⟨xs, xnt⟩, xnu⟩,
  use xs,
  rintros (xt | xu); contradiction
end
```

As an exercise, prove the reverse inclusion:

```
example : s \ (t ∪ u) ⊆ s \ t \ u :=
sorry
```

To prove that two sets are equal, it suffices to show that every element of one is an element of the other. This principle is known as "extensionality," and, unsurprisingly, the `ext` tactic is equipped to handle it.

```
example : s ∩ t = t ∩ s :=
begin
  ext x,
  simp only [mem_inter_eq],
  split,
  { rintros ⟨xs, xt⟩, exact ⟨xt, xs⟩ },
  rintros ⟨xt, xs⟩, exact ⟨xs, xt⟩
end
```

Once again, deleting the line `simp only [mem_inter_eq]` does not harm the proof. In fact, if you like inscrutable proof terms, the following one-line proof is for you:

```
example : s ∩ t = t ∩ s :=
set.ext $ λ x, ⟨λ ⟨xs, xt⟩, ⟨xt, xs⟩, λ ⟨xt, xs⟩, ⟨xs, xt⟩⟩
```

The dollar sign is a useful syntax: writing `f $ ...` is essentially the same as writing `f (...)`, but it saves us the trouble of having to close a set of parentheses at the end of a long expression. Here is an even shorter proof, using the simplifier:

```
example : s ∩ t = t ∩ s :=
by ext x; simp [and.comm]
```

An alternative to using `ext` is to use the theorem `subset.antisymm` which allows us to prove an equation `s = t` between sets by proving `s ⊆ t` and `t ⊆ s`.

```
example : s ∩ t = t ∩ s :=
begin
  apply subset.antisymm,
  { rintros x ⟨xs, xt⟩, exact ⟨xt, xs⟩ },
  rintros x ⟨xt, xs⟩, exact ⟨xs, xt⟩
end
```

Try finishing this proof term:

```
example : s ∩ t = t ∩ s :=
subset.antisymm sorry sorry
```

Remember that you can replace *sorry* by an underscore, and when you hover over it, Lean will show you what it expects at that point.

Here are some set-theoretic identities you might enjoy proving:

```
example : s ∩ (s ∪ t) = s :=
sorry

example : s ∪ (s ∩ t) = s :=
sorry

example : (s \ t) ∪ t = s ∪ t :=
sorry

example : (s \ t) ∪ (t \ s) = (s ∪ t) \ (s ∩ t) :=
sorry
```

When it comes to representing sets, here is what is going on underneath the hood. In type theory, a *property* or *predicate* on a type α is just a function `P : α → Prop`. This makes sense: given `a : α`, `P a` is just the proposition that `P` holds of `a`. In the library, `set α` is defined to be `α → Prop` and `x ∈ s` is defined to be `s x`. In other words, sets are really properties, treated as objects.

The library also defines set-builder notation. The expression `{ y | P y }` unfolds to `(λ y, P y)`, so `x ∈ { y | P y }` reduces to `P x`. So we can turn the property of being even into the set of even numbers:

```
def evens : set ℕ := {n | even n}
def odds :  set ℕ := {n | ¬ even n}

example : evens ∪ odds = univ :=
begin
```

(continues on next page)

(continued from previous page)

```
  rw [evens, odds],
  ext n,
  simp,
  apply classical.em
end
```

You should step through this proof and make sure you understand what is going on. Try deleting the line `rw [evens, odds]` and confirm that the proof still works.

In fact, set-builder notation is used to define

- `s ∩ t` as `{x | x ∈ s ∧ x ∈ t}`,
- `s ∪ t` as `{x | x ∈ s ∨ x ∈ t}`,
- `∅` as `{x | false}`, and
- `univ` as `{x | true}`.

We often need to indicate the type of `∅` and `univ` explicitly, because Lean has trouble guessing which ones we mean. The following examples show how Lean unfolds the last two definitions when needed. In the second one, `trivial` is the canonical proof of `true` in the library.

```
example (x : ℕ) (h : x ∈ (∅ : set ℕ)) : false :=
h

example (x : ℕ) : x ∈ (univ : set ℕ) :=
trivial
```

As an exercise, prove the following inclusion. Use `intro n` to unfold the definition of subset, and use the simplifier to reduce the set-theoretic constructions to logic. We also recommend using the theorems `nat.prime.eq_two_or_odd` and `nat.even_iff`.

```
example : { n | nat.prime n } ∩ { n | n > 2} ⊆ { n | ¬ even n } :=
sorry
```

Be careful: it is somewhat confusing that the library has multiple versions of the predicate `prime`. The most general one makes sense in any commutative monoid with a zero element. The predicate `nat.prime` is specific to the natural numbers. Fortunately, there is a theorem that says that in the specific case, the two notions agree, so you can always rewrite one to the other.

```
#print prime
#print nat.prime

example (n : ℕ) : prime n ↔ nat.prime n := nat.prime_iff.symm

example (n : ℕ) (h : prime n) : nat.prime n :=
by { rw nat.prime_iff, exact h }
```

The *rwa* tactic follows a rewrite with the assumption tactic.

```
example (n : ℕ) (h : prime n) : nat.prime n :=
by rwa nat.prime_iff
```

Lean introduces the notation `∀ x ∈ s, ...`, "for every `x` in `s` .," as an abbreviation for `∀ x, x ∈ s → ...`. It also introduces the notation `∃ x ∈ s, ...`, "there exists an `x` in `s` such that .." These are sometimes known as *bounded quantifiers*, because the construction serves to restrict their significance to the set `s`. As a result, theorems in the library that make use of them often contain `ball` or `bex` in the name. The theorem `bex_def` asserts that `∃ x ∈ s,`

`...` is equivalent to `∃ x, x ∈ s ∧ ...,` but when they are used with `rintros`, `use`, and anonymous constructors, these two expressions behave roughly the same. As a result, we usually don't need to use `bex_def` to transform them explicitly. Here is are some examples of how they are used:

```
variables (s t : set ℕ)

example (h₀ : ∀ x ∈ s, ¬ even x) (h₁ : ∀ x ∈ s, prime x) :
  ∀ x ∈ s, ¬ even x ∧ prime x :=
begin
  intros x xs,
  split,
  { apply h₀ x xs },
  apply h₁ x xs
end

example (h : ∃ x ∈ s, ¬ even x ∧ prime x) :
  ∃ x ∈ s, prime x :=
begin
  rcases h with ⟨x, xs, _, prime_x⟩,
  use [x, xs, prime_x]
end
```

See if you can prove these slight variations:

```
section
variable (ssubt : s ⊆ t)

include ssubt

example (h₀ : ∀ x ∈ t, ¬ even x) (h₁ : ∀ x ∈ t, prime x) :
  ∀ x ∈ s, ¬ even x ∧ prime x :=
sorry

example (h : ∃ x ∈ s, ¬ even x ∧ prime x) :
  ∃ x ∈ t, prime x :=
sorry

end
```

The `include` command is needed because `ssubt` does not appear in the statement of the theorem. Lean does not look inside tactic blocks when it decides what variables and hypotheses to include, so if you delete that line, you will not see the hypothesis within a `begin .end` proof. If you are proving theorems in a library, you can delimit the scope of and `include` by putting it between `section` and `end`, so that later theorems do not include it as an unnecessary hypothesis.

Indexed unions and intersections are another important set-theoretic construction. We can model a sequence $A_0, A_1, A_2, \ldots$ of sets of elements of α as a function `A : ℕ → set α`, in which case `⋃ i, A i` denotes their union, and `⋂ i, A i` denotes their intersection. There is nothing special about the natural numbers here, so `ℕ` can be replaced by any type `I` used to index the sets. The following illustrates their use.

```
variables {α I : Type*}
variables A B : I → set α
variable  s : set α
open set

example : s ∩ (⋃ i, A i) = ⋃ i, (A i ∩ s) :=
begin
  ext x,
```

(continues on next page)

(continued from previous page)

```
  simp only [mem_inter_eq, mem_Union],
  split,
  { rintros ⟨xs, ⟨i, xAi⟩⟩,
    exact ⟨i, xAi, xs⟩ },
  rintros ⟨i, xAi, xs⟩,
  exact ⟨xs, ⟨i, xAi⟩⟩
end

example : (⋂ i, A i ∩ B i) = (⋂ i, A i) ∩ (⋂ i, B i) :=
begin
  ext x,
  simp only [mem_inter_eq, mem_Inter],
  split,
  { intro h,
    split,
    { intro i,
      exact (h i).1 },
    intro i,
    exact (h i).2 },
  rintros ⟨h1, h2⟩ i,
  split,
  { exact h1 i },
  exact h2 i
end
```

Parentheses are often needed with an indexed union or intersection because, as with the quantifiers, the scope of the bound variable extends as far as it can.

Try proving the following identity. One direction requires classical logic! We recommend using `by_cases xs : x ∈ s` at an appropriate point in the proof.

```
open_locale classical

example : s ∪ (⋂ i, A i) = ⋂ i, (A i ∪ s) :=
sorry
```

Mathlib also has bounded unions and intersections, which are analogous to the bounded quantifiers. You can unpack their meaning with `mem_Union₂` and `mem_Inter₂`. As the following examples show, Lean's simplifier carries out these replacements as well.

```
def primes : set ℕ := {x | nat.prime x}

example : (⋃ p ∈ primes, {x | p^2 ∣ x}) = {x | ∃ p ∈ primes, p^2 ∣ x} :=
by { ext, rw mem_Union₂, refl }

example : (⋃ p ∈ primes, {x | p^2 ∣ x}) = {x | ∃ p ∈ primes, p^2 ∣ x} :=
by { ext, simp }

example : (⋂ p ∈ primes, {x | ¬ p ∣ x}) ⊆ {x | x < 2} :=
begin
  intro x,
  contrapose!,
  simp,
  apply nat.exists_prime_and_dvd
end
```

Try solving the following example, which is similar. If you start typing `eq_univ`, tab completion will tell you that

`apply eq_univ_of_forall` is a good way to start the proof. We also recommend using the theorem `nat.exists_infinite_primes`.

```
example : (⋃ p ∈ primes, {x | x ≤ p}) = univ :=
sorry
```

Give a collection of sets, `s : set (set α)`, their union, $\bigcup_0$ `s`, has type `set α` and is defined as `{x | ∃ t ∈ s, x ∈ t}`. Similarly, their intersection, $\bigcap_0$ `s`, is defined as `{x | ∀ t ∈ s, x ∈ t}`. These operations are called `sUnion` and `sInter`, respectively. The following examples show their relationship to bounded union and intersection.

```
variables {α : Type*} (s : set (set α))

example : ⋃₀ s = ⋃ t ∈ s, t :=
begin
  ext x,
  rw mem_Union₂,
  refl
end

example : ⋂₀ s = ⋂ t ∈ s, t :=
begin
  ext x,
  rw mem_Inter₂,
  refl
end
```

In the library, these identities are called `sUnion_eq_bUnion` and `sInter_eq_bInter`.

4.2 Functions

If `f : α → β` is a function and `p` is a set of elements of type β, the library defines `preimage f p`, written `f ⁻¹' p`, to be `{x | f x ∈ p}`. The expression `x ∈ f ⁻¹' p` reduces to `f x ∈ p`. This is often convenient, as in the following example:

```
variables {α β : Type*}
variable  f : α → β
variables s t : set α
variables u v : set β
open function
open set

example : f ⁻¹' (u ∩ v) = f ⁻¹' u ∩ f ⁻¹' v :=
by { ext, refl }
```

If `s` is a set of elements of type α, the library also defines `image f s`, written `f '' s`, to be `{y | ∃ x, x ∈ s ∧ f x = y}`. So a hypothesis `y ∈ f '' s` decomposes to a triple `⟨x, xs, xeq⟩` with `x : α` satisfying the hypotheses `xs : x ∈ s` and `xeq : f x = y`. The `rfl` tag in the `rintros` tactic (see Section 3.2) was made precisely for this sort of situation.

```
example : f '' (s ∪ t) = f '' s ∪ f '' t :=
begin
  ext y, split,
  { rintros ⟨x, xs | xt, rfl⟩,
    { left, use [x, xs] },
    right, use [x, xt] },
```

(continues on next page)

(continued from previous page)

```
  rintros (⟨x, xs, rfl⟩ | ⟨x, xt, rfl⟩),
  { use [x, or.inl xs] },
  use [x, or.inr xt]
end
```

Notice also that the `use` tactic applies `refl` to close goals when it can.

Here is another example:

```
example : s ⊆ f ⁻¹' (f '' s) :=
begin
  intros x xs,
  show f x ∈ f '' s,
  use [x, xs]
end
```

We can replace the line `use [x, xs]` by `apply mem_image_of_mem f xs` if we want to use a theorem specifically designed for that purpose. But knowing that the image is defined in terms of an existential quantifier is often convenient.

The following equivalence is a good exercise:

```
example : f '' s ⊆ v ↔ s ⊆ f ⁻¹' v :=
sorry
```

It shows that `image f` and `preimage f` are an instance of what is known as a *Galois connection* between `set` α and `set` β, each partially ordered by the subset relation. In the library, this equivalence is named `image_subset_iff`. In practice, the right-hand side is often the more useful representation, because `y ∈ f ⁻¹' t` unfolds to `f y ∈ t` whereas working with `x ∈ f '' s` requires decomposing an existential quantifier.

Here is a long list of set-theoretic identities for you to enjoy. You don't have to do all of them at once; do a few of them, and set the rest aside for a rainy day.

```
example (h : injective f) : f ⁻¹' (f '' s) ⊆ s :=
sorry

example : f '' (f⁻¹' u) ⊆ u :=
sorry

example (h : surjective f) : u ⊆ f '' (f⁻¹' u) :=
sorry

example (h : s ⊆ t) : f '' s ⊆ f '' t :=
sorry

example (h : u ⊆ v) : f ⁻¹' u ⊆ f ⁻¹' v :=
sorry

example : f ⁻¹' (u ∪ v) = f ⁻¹' u ∪ f ⁻¹' v :=
sorry

example : f '' (s ∩ t) ⊆ f '' s ∩ f '' t :=
sorry

example (h : injective f) : f '' s ∩ f '' t ⊆ f '' (s ∩ t) :=
sorry
```

(continues on next page)

(continued from previous page)

```
example : f '' s \ f '' t ⊆ f '' (s \ t) :=
sorry

example : f ⁻¹' u \ f ⁻¹' v ⊆ f ⁻¹' (u \ v) :=
sorry

example : f '' s ∩ v = f '' (s ∩ f ⁻¹' v) :=
sorry

example : f '' (s ∩ f ⁻¹' u) ⊆ f '' s ∪ u :=
sorry

example : s ∩ f ⁻¹' u ⊆ f ⁻¹' (f '' s ∩ u) :=
sorry

example : s ∪ f ⁻¹' u ⊆ f ⁻¹' (f '' s ∪ u) :=
sorry
```

You can also try your hand at the next group of exercises, which characterize the behavior of images and preimages with respect to indexed unions and intersections. In the third exercise, the argument `i : I` is needed to guarantee that the index set is nonempty. To prove any of these, we recommend using `ext` or `intro` to unfold the meaning of an equation or inclusion between sets, and then calling `simp` to unpack the conditions for membership.

```
variables {I : Type*} (A : I → set α) (B : I → set β)

example : f '' (⋃ i, A i) = ⋃ i, f '' A i :=
begin
  ext y, simp,
  split,
  { rintros ⟨x, ⟨i, xAi⟩, fxeq⟩,
    use [i, x, xAi, fxeq] },
  rintros ⟨i, x, xAi, fxeq⟩,
  exact ⟨x, ⟨i, xAi⟩, fxeq⟩
end

example : f '' (⋂ i, A i) ⊆ ⋂ i, f '' A i :=
begin
  intro y, simp,
  intros x h fxeq i,
  use [x, h i, fxeq],
end

example (i : I) (injf : injective f) :
  (⋂ i, f '' A i) ⊆ f '' (⋂ i, A i) :=
begin
  intro y, simp,
  intro h,
  rcases h i with ⟨x, xAi, fxeq⟩,
  use x, split,
  { intro i',
    rcases h i' with ⟨x', x'Ai, fx'eq⟩,
    have : f x = f x', by rw [fxeq, fx'eq],
    have : x = x', from injf this,
    rw this,
    exact x'Ai },
  exact fxeq
```

(continues on next page)

(continued from previous page)

```
end

example : f ⁻¹' (⋃ i, B i) = ⋃ i, f ⁻¹' (B i) :=
by { ext x, simp }

example : f ⁻¹' (⋂ i, B i) = ⋂ i, f ⁻¹' (B i) :=
by { ext x, simp }
```

```
example : inj_on f s ↔
  ∀ x₁ ∈ s, ∀ x₂ ∈ s, f x₁ = f x₂ → x₁ = x₂ :=
iff.refl _
```

The statement `injective f` is provably equivalent to `inj_on f univ`. Similarly, the library defines `range f` to be `{x | ∃y, f y = x}`, so `range f` is provably equal to `f '' univ`. This is a common theme in mathlib: although many properties of functions are defined relative to their full domain, there are often relativized versions that restrict the statements to a subset of the domain type.

Here is are some examples of `inj_on` and `range` in use:

```
open set real

example : inj_on log { x | x > 0 } :=
begin
  intros x xpos y ypos,
  intro e,    -- log x = log y
  calc
    x   = exp (log x) : by rw exp_log xpos
    ... = exp (log y) : by rw e
    ... = y           : by rw exp_log ypos
end

example : range exp = { y | y > 0 } :=
begin
  ext y, split,
  { rintros ⟨x, rfl⟩,
    apply exp_pos },
  intro ypos,
  use log y,
  rw exp_log ypos
end
```

Try proving these:

```
example : inj_on sqrt { x | x ≥ 0 } :=
sorry

example : inj_on (λ x, x^2) { x : ℝ | x ≥ 0 } :=
sorry

example : sqrt '' { x | x ≥ 0 } = {y | y ≥ 0} :=
sorry

example : range (λ x, x^2) = {y : ℝ  | y ≥ 0} :=
sorry
```

To define the inverse of a function `f : α → β`, we will use two new ingredients. First, we need to deal with the fact that an arbitrary type in Lean may be empty. To define the inverse to `f` at `y` when there is no `x` satisfying `f x = y`, we

want to assign a default value in α. Adding the annotation `[inhabited α]` as a variable is tantamount to assuming that α has a preferred element, which is denoted `default`. Second, in the case where there is more than one `x` such that `f x = y`, the inverse function needs to *choose* one of them. This requires an appeal to the *axiom of choice*. Lean allows various ways of accessing it; one convenient method is to use the classical `some` operator, illustrated below.

```
variables {α β : Type*} [inhabited α]

#check (default : α)

variables (P : α → Prop) (h : ∃ x, P x)

#check classical.some h

example : P (classical.some h) := classical.some_spec h
```

Given `h : ∃ x, P x`, the value of `classical.some h` is some `x` satisfying `P x`. The theorem `classical.some_spec h` says that `classical.some h` meets this specification.

With these in hand, we can define the inverse function as follows:

```
noncomputable theory
open_locale classical

def inverse (f : α → β) : β → α :=
λ y : β, if h : ∃ x, f x = y then classical.some h else default

theorem inverse_spec {f : α → β} (y : β) (h : ∃ x, f x = y) :
  f (inverse f y) = y :=
begin
  rw inverse, dsimp, rw dif_pos h,
  exact classical.some_spec h
end
```

The lines `noncomputable theory` and `open_locale classical` are needed because we are using classical logic in an essential way. On input `y`, the function `inverse f` returns some value of `x` satisfying `f x = y` if there is one, and a default element of α otherwise. This is an instance of a *dependent if* construction, since in the positive case, the value returned, `classical.some h`, depends on the assumption `h`. The identity `dif_pos h` rewrites `if h : e then a else b` to `a` given `h : e`, and, similarly, `dif_neg h` rewrites it to `b` given `h : ¬ e`. The theorem `inverse_spec` says that `inverse f` meets the first part of this specification.

Don't worry if you do not fully understand how these work. The theorem `inverse_spec` alone should be enough to show that `inverse f` is a left inverse if and only if `f` is injective and a right inverse if and only if `f` is surjective. Look up the definition of `left_inverse` and `right_inverse` by double-clicking or right-clicking on them in VS Code, or using the commands `#print left_inverse` and `#print right_inverse`. Then try to prove the two theorems. They are tricky! It helps to do the proofs on paper before you start hacking through the details. You should be able to prove each of them with about a half-dozen short lines. If you are looking for an extra challenge, try to condense each proof to a single-line proof term.

```
variable  f : α → β
open function

example : injective f ↔ left_inverse (inverse f) f  :=
sorry

example : surjective f ↔ right_inverse (inverse f) f :=
sorry
```

We close this section with a type-theoretic statement of Cantor's famous theorem that there is no surjective function from

a set to its power set. See if you can understand the proof, and then fill in the two lines that are missing.

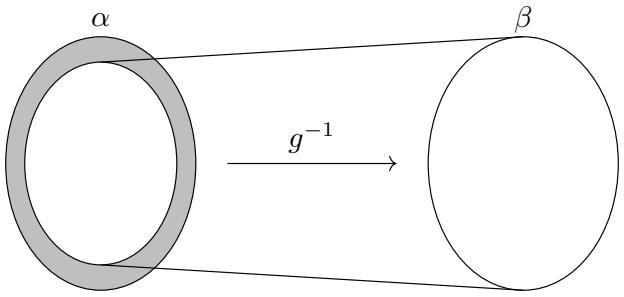
```
theorem Cantor : ∀ f : α → set α, ¬ surjective f :=
begin
  intros f surjf,
  let S := { i | i ∉ f i},
  rcases surjf S with ⟨j, h⟩,
  have h₁ : j ∉ f j,
  { intro h',
    have : j ∉ f j,
      { by rwa h at h' },
    contradiction },
  have h₂ : j ∈ S,
    sorry,
  have h₃ : j ∉ S,
    sorry,
  contradiction
end
```

4.3 The Schröder-Bernstein Theorem

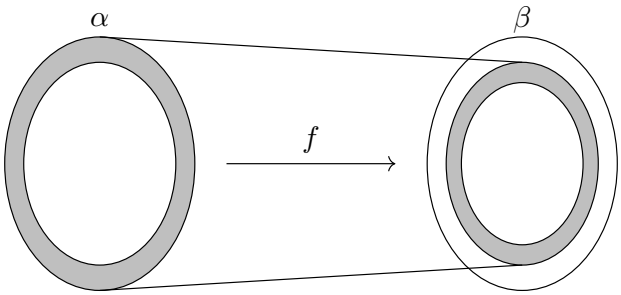
We close this chapter with an elementary but nontrivial theorem of set theory. Let α and β be sets. (In our formalization, they will actually be types.) Suppose $f : \alpha \to \beta$ and $g : \beta \to \alpha$ are both injective. Intuitively, this means that α is no bigger than β and vice-versa. If α and β are finite, this implies that they have the same cardinality, which is equivalent to saying that there is a bijection between them. In the nineteenth century, Cantor stated that same result holds even in the case where α and β are infinite. This was eventually established by Dedekind, Schröder, and Bernstein independently.

Our formalization will introduce some new methods that we will explain in greater detail in chapters to come. Don't worry if they go by too quickly here. Our goal is to show you that you already have the skills to contribute to the formal proof of a real mathematical result.

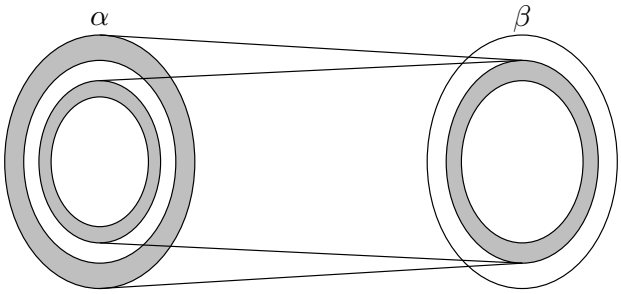
To understand the idea behind the proof, consider the image of the map g in α. On that image, the inverse of g is defined and is a bijection with β.



The problem is that the bijection does not include the shaded region in the diagram, which is nonempty if g is not surjective. Alternatively, we can use f to map all of α to β, but in that case the problem is that if f is not surjective, it will miss some elements of β.



But now consider the composition $g \circ f$ from α to itself. Because the composition is injective, it forms a bijection between α and its image, yielding a scaled-down copy of α inside itself.



This composition maps the inner shaded ring to yet another such set, which we can think of as an even smaller concentric shaded ring, and so on. This yields a concentric sequence of shaded rings, each of which is in bijective correspondence with the next. If we map each ring to the next and leave the unshaded parts of α alone, we have a bijection of α with the image of g. Composing with g^{-1}, this yields the desired bijection between α and β.

We can describe this bijection more simply. Let A be the union of the sequence of shaded regions, and define $h : \alpha \to \beta$ as follows:

$$h(x) = \begin{cases} f(x) & \text{if } x \in A \\ g^{-1}(x) & \text{otherwise.} \end{cases}$$

In other words, we use f on the shaded parts, and we use the inverse of g everywhere else. The resulting map h is injective because each component is injective and the images of the two components are disjoint. To see that it is surjective, suppose we are given a y in β, and consider $g(y)$. If $g(y)$ is in one of the shaded regions, it cannot be in the first ring, so we have $g(y) = g(f(x))$ for some x is in the previous ring. By the injectivity of g, we have $h(x) = f(x) = y$. If $g(y)$ is not in the shaded region, then by the definition of h, we have $h(g(y)) = y$. Either way, y is in the image of h.

This argument should sound plausible, but the details are delicate. Formalizing the proof will not only improve our confidence in the result, but also help us understand it better. Because the proof uses classical logic, we tell Lean that our definitions will generally not be computable.

```
noncomputable theory
open_locale classical

variables {α β : Type*} [nonempty β]
```

The annotation `[nonempty β]` specifies that β is nonempty. We use it because the mathlib primitive that we will use to construct g^{-1} requires it. The case of the theorem where β is empty is trivial, and even though it would not be hard to generalize the formalization to cover that case as well, we will not bother. Specifically, we need the hypothesis `[nonempty β]` for the operation `inv_fun` that is defined in mathlib. Given `x : α`, `inv_fun g x` chooses a preimage of `x` in β if there is one, and returns an arbitrary element of β otherwise. The function `inv_fun g` is always a left inverse if `g` is injective and a right inverse if `g` is surjective.

```
#check (inv_fun g : α → β)

#check (left_inverse_inv_fun : injective g → left_inverse (inv_fun g) g)
#check (left_inverse_inv_fun : injective g → ∀ y, inv_fun g (g y) = y)

#check (inv_fun_eq : (∃ y, g y = x) → g (inv_fun g x) = x)
```

We define the set corresponding to the union of the shaded regions as follows.

```
variables (f : α → β) (g : β → α)

def sb_aux : ℕ → set α
| 0       := univ \ (g '' univ)
| (n + 1) := g '' (f '' sb_aux n)

def sb_set := ⋃ n, sb_aux f g n
```

The definition `sb_aux` is an example of a *recursive definition*, which we will explain in the next chapter. It defines a sequence of sets

$$S_0 = \alpha \setminus g(\beta)$$
$$S_{n+1} = g(f(S_n)).$$

The definition `sb_set` corresponds to the set $A = \bigcup_{n \in \mathbb{N}} S_n$ in our proof sketch. The function h described above is now defined as follows:

```
def sb_fun (x : α) : β := if x ∈ sb_set f g then f x else inv_fun g x
```

We will need the fact that our definition of g^{-1} is a right inverse on the complement of A, which is to say, on the non-shaded regions of α. This is so because the outermost ring, S_0, is equal to $\alpha \setminus g(\beta)$, so the complement of A is contained in $g(\beta)$. As a result, for every x in the complement of A, there is a y such that $g(y) = x$. (By the injectivity of g, this y is unique, but next theorem says only that `inv_fun g x` returns some `y` such that `g y = x`.)

Step through the proof below, make sure you understand what is going on, and fill in the remaining parts. You will need to use `inv_fun_eq` at the end. Notice that rewriting with `sb_aux` here replaces `sb_aux f g 0` with the right-hand side of the corresponding defining equation.

```
theorem sb_right_inv {x : α} (hx : x ∉ sb_set f g) :
    g (inv_fun g x) = x :=
begin
  have : x ∈ g '' univ,
  { contrapose! hx,
    rw [sb_set, mem_Union],
    use [0],
    rw [sb_aux, mem_diff],
    sorry },
  have : ∃ y, g y = x,
  { sorry },
  sorry
end
```

We now turn to the proof that h is injective. Informally, the proof goes as follows. First, suppose $h(x_1) = h(x_2)$. If x_1 is in A, then $h(x_1) = f(x_1)$, and we can show that x_2 is in A as follows. If it isn't, then we have $h(x_2) = g^{-1}(x_2)$. From $f(x_1) = h(x_1) = h(x_2)$ we have $g(f(x_1)) = x_2$. From the definition of A, since x_1 is in A, x_2 is in A as well, a contradiction. Hence, if x_1 is in A, so is x_2, in which case we have $f(x_1) = h(x_1) = h(x_2) = f(x_2)$. The injectivity of f then implies $x_1 = x_2$. The symmetric argument shows that if x_2 is in A, then so is x_1, which again implies $x_1 = x_2$.

The only remaining possibility is that neither x_1 nor x_2 is in A. In that case, we have $g^{-1}(x_1) = h(x_1) = h(x_2) = g^{-1}(x_2)$. Applying g to both sides yields $x_1 = x_2$.

Once again, we encourage you to step through the following proof to see how the argument plays out in Lean. See if you can finish off the proof using `sb_right_inv`.

```
theorem sb_injective (hf: injective f) (hg : injective g) :
  injective (sb_fun f g) :=
begin
  set A := sb_set f g with A_def,
  set h := sb_fun f g with h_def,
  intros x₁ x₂,
  assume hxeq : h x₁ = h x₂,
  show x₁ = x₂,
  simp only [h_def, sb_fun, ←A_def] at hxeq,
  by_cases xA : x₁ ∈ A ∨ x₂ ∈ A,
  { wlog : x₁ ∈ A := xA using [x₁ x₂, x₂ x₁],
    have x₂A : x₂ ∈ A,
    { apply not_imp_self.mp,
      assume x₂nA : x₂ ∉ A,
      rw [if_pos xA, if_neg x₂nA] at hxeq,
      rw [A_def, sb_set, mem_Union] at xA,
      have x₂eq : x₂ = g (f x₁),
      { sorry },
      rcases xA with ⟨n, hn⟩,
      rw [A_def, sb_set, mem_Union],
      use n + 1,
      simp [sb_aux],
      exact ⟨x₁, hn, x₂eq.symm⟩ },
    sorry },
  push_neg at xA,
  sorry
end
```

The proof introduces some new tactics. To start with, notice the `set` tactic, which introduces abbreviations `A` and `h` for `sb_set f g` and `sb_fun f g` respectively. We name the corresponding defining equations `A_def` and `h_def`. The abbreviations are definitional, which is to say, Lean will sometimes unfold them automatically when needed. But not always; for example, when using `rw`, we generally need to use `A_def` and `h_def` explicitly. So the definitions bring a tradeoff: they can make expressions shorter and more readable, but they sometimes require us to do more work.

A more interesting tactic is the `wlog` tactic, which encapsulates the symmetry argument in the informal proof above. We will not dwell on it now, but notice that it does exactly what we want. If you hover over the tactic you can take a look at its documentation.

The argument for surjectivity is even easier. Given y in β, we consider two cases, depending on whether $g(y)$ is in A. If it is, it can't be in S_0, the outermost ring, because by definition that is disjoint from the image of g. Thus it is an element of S_{n+1} for some n. This means that it is of the form $g(f(x))$ for some x in S_n. By the injectivity of g, we have $f(x) = y$. In the case where $g(y)$ is in the complement of A, we immediately have $h(g(y)) = y$, and we are done.

Once again, we encourage you to step through the proof and fill in the missing parts. The tactic `cases n with n` splits on the cases `g y ∈ sb_aux f g 0` and `g y ∈ sb_aux f g n.succ`. In both cases, calling the simplifier with `simp [sb_aux]` applies the corresponding defining equation of `sb_aux`.

```
theorem sb_surjective (hf: injective f) (hg : injective g) :
  surjective (sb_fun f g) :=
begin
  set A := sb_set f g with A_def,
  set h := sb_fun f g with h_def,
```

(continues on next page)

(continued from previous page)

```
  intro y,
  by_cases gyA : g y ∈ A,
  { rw [A_def, sb_set, mem_Union] at gyA,
    rcases gyA with ⟨n, hn⟩,
    cases n with n,
    { simp [sb_aux] at hn,
      contradiction },
    simp [sb_aux] at hn,
    rcases hn with ⟨x, xmem, hx⟩,
    use x,
    have : x ∈ A,
    { rw [A_def, sb_set, mem_Union],
      exact ⟨n, xmem⟩ },
    simp only [h_def, sb_fun, if_pos this],
    exact hg hx },
  sorry
end
```

We can now put it all together. The final statement is short and sweet, and the proof uses the fact that `bijective h` unfolds to `injective h ∧ surjective h`.

```
theorem schroeder_bernstein {f : α → β} {g : β → α}
    (hf: injective f) (hg : injective g) :
  ∃ h : α → β, bijective h :=
⟨sb_fun f g, sb_injective f g hf hg, sb_surjective f g hf hg⟩
```

CHAPTER

FIVE

NUMBER THEORY

In this chapter, we show you how to formalize some elementary results in number theory. As we deal with more substantive mathematical content, the proofs will get longer and more involved, building on the skills you have already mastered.

5.1 Irrational Roots

Let's start with a fact known to the ancient greeks, namely, that the square root of 2 is irrational. If we suppose otherwise, we can write $\sqrt{2} = a/b$ as a fraction in lowest terms. Squaring both sides yields $a^2 = 2b^2$, which implies that a is even. If we write $a = 2c$, then we get $4c^2 = 2b^2$ and hence $b^2 = 2c^2$. This implies that b is also even, contradicting the fact that we have assumed that a/b has been reduced to lowest terms.

Saying that a/b is a fraction in lowest terms means that a and b do not have any factors in common, which is to say, they are *coprime*. Mathlib defines the predicate `nat.coprime m n` to be `nat.gcd m n = 1`. Using Lean's anonymous projection notation, if `s` and `t` are expressions of type `nat`, we can write `s.coprime t` instead of `nat.coprime s t`, and similarly for `nat.gcd`. As usual, Lean will often unfold the definition of `nat.coprime` automatically when necessary, but we can also do it manually by rewriting or simplifying with the identifier `nat.coprime`. The `norm_num` tactic is smart enough to compute concrete values.

```
#print nat.coprime

example (m n : nat) (h : m.coprime n) : m.gcd n = 1 := h

example (m n : nat) (h : m.coprime n) : m.gcd n = 1 :=
by { rw nat.coprime at h, exact h }

example : nat.coprime 12 7 := by norm_num
example : nat.gcd 12 8 = 4 := by norm_num
```

We have already encountered the `gcd` function in Section 2.4. There is also a version of `gcd` for the integers; we will return to a discussion of the relationship between different number systems below. There are even a generic `gcd` function and generic notions of `prime` and `is_coprime` that make sense in general classes of algebraic structures. We will come to understand how Lean manages this generality in the next chapter. In the meanwhile, in this section, we will restrict attention to the natural numbers.

We also need the notion of a prime number, `nat.prime`. The theorem `nat.prime_def_lt` provides one familiar characterization, and `nat.prime.eq_one_or_self_of_dvd` provides another.

```
#check @nat.prime_def_lt

example (p : ℕ) (prime_p : nat.prime p) : 2 ≤ p ∧ ∀ (m : ℕ), m < p → m ∣ p → m = 1 :=
by rwa nat.prime_def_lt at prime_p
```

(continues on next page)

(continued from previous page)

```
#check nat.prime.eq_one_or_self_of_dvd

example (p : ℕ) (prime_p : nat.prime p) : ∀ (m : ℕ), m ∣ p → m = 1 ∨ m = p :=
prime_p.eq_one_or_self_of_dvd

example : nat.prime 17 := by norm_num

-- commonly used
example : nat.prime 2 := nat.prime_two
example : nat.prime 3 := nat.prime_three
```

In the natural numbers, a prime number has the property that it cannot be written as a product of nontrivial factors. In a broader mathematical context, an element of a ring that has this property is said to be *irreducible*. An element of a ring is said to be *prime* if whenever it divides a product, it divides one of the factors. It is an important property of the natural numbers that in that setting the two notions coincide, giving rise to the theorem `nat.prime.dvd_mul`.

We can use this fact to establish a key property in the argument above: if the square of a number is even, then that number is even as well. Mathlib defines the predicate `even` in `data.nat.parity`, but for reasons that will become clear below, we will simply use `2 ∣ m` to express that `m` is even.

```
#check @nat.prime.dvd_mul
#check nat.prime.dvd_mul nat.prime_two
#check nat.prime_two.dvd_mul

lemma even_of_even_sqr {m : ℕ} (h : 2 ∣ m^2) : 2 ∣ m :=
begin
  rw [pow_two, nat.prime_two.dvd_mul] at h,
  cases h; assumption
end

example {m : ℕ} (h : 2 ∣ m^2) : 2 ∣ m :=
nat.prime.dvd_of_dvd_pow nat.prime_two h
```

As we proceed, you will need to become proficient at finding the facts you need. Remember that if you can guess the prefix of the name and you have imported the relevent library, you can use tab completion (sometimes with `ctrl-tab`) to find what you are looking for. You can use `ctrl-click` on any identifier to jump to the file where it is defined, which enables you to browse definitions and theorems nearby. You can also use the search engine on the Lean community web pages, and if all else fails, don't hesitate to ask on Zulip.

```
example (a b c : nat) (h : a * b = a * c) (h' : a ≠ 0) :
  b = c :=
begin
  -- library_search suggests the following:
  exact (mul_right_inj' h').mp h
end
```

The heart of our proof of the irrationality of the square root of two is contained in the following theorem. See if you can fill out the proof sketch, using `even_of_even_sqr` and the theorem `nat.dvd_gcd`.

```
example {m n : ℕ} (coprime_mn : m.coprime n) : m^2 ≠ 2 * n^2 :=
begin
  intro sqr_eq,
  have : 2 ∣ m,
    sorry,
  obtain ⟨k, meq⟩ := dvd_iff_exists_eq_mul_left.mp this,
  have : 2 * (2 * k^2) = 2 * n^2,
```

(continues on next page)

(continued from previous page)

```
  { rw [←sqr_eq, meq], ring },
  have : 2 * k^2 = n^2,
    sorry,
  have : 2 ∣ n,
    sorry,
  have : 2 ∣ m.gcd n,
    sorry,
  have : 2 ∣ 1,
    sorry,
  norm_num at this
end
```

In fact, with very few changes, we can replace `2` by an arbitrary prime. Give it a try in the next example. At the end of the proof, you'll need to derive a contradiction from `p ∣ 1`. You can use `nat.prime.two_le`, which says that any prime number is greater than or equal to two, and `nat.le_of_dvd`.

```
example {m n p : ℕ} (coprime_mn : m.coprime n) (prime_p : p.prime) : m^2 ≠ p * n^2 :=
    sorry
```

Let us consider another approach. Here is a quick proof that if p is prime, then $m^2 \neq pn^2$: if we assume $m^2 = pn^2$ and consider the factorization of m and n into primes, then p occurs an even number of times on the left side of the equation and an odd number of times on the right, a contradiction. Note that this argument requires that n and hence m are not equal to zero. The formalization below confirms that this assumption is sufficient.

The unique factorization theorem says that any natural number other than zero can be written as the product of primes in a unique way. Mathlib contains a formal version of this, expressed in terms of a function `nat.factors`, which returns the list of prime factors of a number in nondecreasing order. The library proves that all the elements of `nat.factors n` are prime, that any `n` greater than zero is equal to the product of its factors, and that if `n` is equal to the product of another list of prime numbers, then that list is a permutation of `nat.factors n`.

```
#check nat.factors
#check nat.prime_of_mem_factors
#check nat.prod_factors
#check nat.factors_unique
```

You can browse these theorems and others nearby, even though we have not talked about list membership, products, or permutations yet. We won't need any of that for the task at hand. We will instead use the fact that Mathlib has a function `list.count` that counts the number of times an element occurs in a list. Counting the number of times that a prime `p` occurs in `n.factors` yields the multiplicity of `p` in the prime factorization of `n`, and the library contains some key identities.

```
#check @nat.count_factors_mul_of_pos
#check @nat.factors_count_pow
#check @nat.factors_prime

example (m n p : ℕ) (mpos : 0 < m) (npos : 0 < n) :
  (m * n).factors.count p = m.factors.count p + n.factors.count p :=
nat.count_factors_mul_of_pos mpos npos

example (n k p : ℕ) : (n^k).factors.count p = k * n.factors.count p :=
nat.factors_count_pow

example (p : ℕ) (prime_p : p.prime) : p.factors.count p = 1 :=
begin
  rw nat.factors_prime prime_p,
```

(continues on next page)

(continued from previous page)

```
  simp
end
```

Notice that in the third example, we use the simplifier to close the goal `list.count p [p] = 1`. This is the sort of thing that the simplifier is good for. In fact, the equation `list.count_singleton` would do the trick. If you replace `simp` by `squeeze_simp`, the information window will show you that the simplifier has found a more roundabout way of discharging the goal. The next example shows that the simplifier is also smart enough to replace `n^2 ≠ 0` by `n ≠ 0`. The tactic `simpa` just calls `simp` followed by `assumption`.

See if you can use the identities above to fill in the missing parts of the proof.

```
example {m n p : ℕ} (nnz : n ≠ 0) (prime_p : p.prime) : m^2 ≠ p * n^2 :=
begin
  intro sqr_eq,
  have nsqr_nez : n^2 ≠ 0,
    by simpa,
  have eq1 : (m^2).factors.count p = 2 * m.factors.count p,
    sorry,
  have eq2 : (p * n^2).factors.count p = 2 * n.factors.count p + 1,
    sorry,
  have : (2 * m.factors.count p) % 2 = (2 * n.factors.count p + 1) % 2,
  { rw [←eq1, sqr_eq, eq2] },
  rw [add_comm, nat.add_mul_mod_self_left, nat.mul_mod_right] at this,
  norm_num at this
end
```

A nice thing about this proof is that it also generalizes. There is nothing special about `2`; with small changes, the proof shows that whenever we write `m^k = r * n^k`, the multiplicity of any prime `p` in `r` has to be a multiple of `k`.

To use `nat.count_factors_mul_of_pos` with `r * n^k`, we need to know that `r` is positive. But when `r` is zero, the theorem below is trivial, and easily proved by the simplifier. So the proof is carried out in cases. The line `cases r with r` replaces the goal with two versions: one in which `r` is replaced by `0`, and the other in which `r` is replaces by `r.succ`, the successor of `r`. In the second case, we can use the theorem `r.succ_pos`, which establishes `0 < r.succ`.

Notice also that the line that begins `have : npow_nz` provides a short proof-term proof of `n^k ≠ 0`. To understand how it works, try replacing it with a tactic proof, and then think about how the tactics describe the proof term.

See if you can fill in the missing parts of the proof below. At the very end, you can use `nat.dvd_sub'` and `nat.dvd_mul_right` to finish it off.

```
example {m n k r : ℕ} (nnz : n ≠ 0) (pow_eq : m^k = r * n^k) :
  ∀ p : ℕ, p.prime → k ∣ r.factors.count p :=
begin
  intros p prime_p,
  cases r with r,
  { simp },
  have npow_nz : n^k ≠ 0 := λ npowz, nnz (pow_eq_zero npowz),
  have eq1 : (m^k).factors.count p = k * m.factors.count p,
    sorry,
  have eq2 : (r.succ * n^k).factors.count p =
      k * n.factors.count p + r.succ.factors.count p,
    sorry,
  have : r.succ.factors.count p = k * m.factors.count p - k * n.factors.count p,
  { rw [←eq1, pow_eq, eq2, add_comm, nat.add_sub_cancel] },
  rw this,
```

(continues on next page)

(continued from previous page)

```
  sorry
end
```

There are a number of ways in which we might want to improve on these results. To start with, a proof that the square root of two is irrational should say something about the square root of two, which can be understood as an element of the real or complex numbers. And stating that it is irrational should say something about the rational numbers, namely, that no rational number is equal to it. Moreover, we should extend the theorems in this section to the integers. Although it is mathematically obvious that if we could write the square root of two as a quotient of two integers then we could write it as a quotient of two natural numbers, proving this formally requires some effort.

In Mathlib, the natural numbers, the integers, the rationals, the reals, and the complex numbers are represented by separate data types. Restricting attention to the separate domains is often helpful: we will see that it is easy to do induction on the natural numbers, and it is easiest to reason about divisibility of integers when the real numbers are not part of the picture. But having to mediate between the different domains is a headache, one we will have to contend with. We will return to this issue later in this chapter.

We should also expect to be able to strengthen the conclusion of the last theorem to say that the number `r` is a `k`-th power, since its `k`-th root is just the product of each prime dividing `r` raised to its multiplicity in `r` divided by `k`. To be able to do that we will need better means for reasoning about products and sums over a finite set, which is also a topic we will return to.

In fact, the results in this section are all established in much greater generality in mathlib, in `data.real.irrational`. The notion of `multiplicity` is defined for an arbitrary commutative monoid, and that it takes values in the extended natural numbers `enat`, which adds the value infinity to the natural numbers. In the next chapter, we will begin to develop the means to appreciate the way that Lean supports this sort of generality.

5.2 Induction and Recursion

The set of natural numbers $\mathbb{N} = \{0, 1, 2, \ldots\}$ is not only fundamentally important in its own right, but also a plays a central role in the construction of new mathematical objects. Lean's foundation allows us to declare *inductive types*, which are types generated inductively by a given list of *constructors*. In Lean, the natural numbers are declared as follows.

```
inductive nat
| zero : nat
| succ (n : nat) : nat
```

You can find this in the library by writting `#check nat` and then using `ctrl-click` on the identifier `nat`. The command specifies that `nat` is the datatype generated freely and inductively by the two constructors `zero : nat` and `succ : nat → nat`. Of course, the library introduces notation $\mathbb{N}$ and `0` for `nat` and `zero` respectively. (Numerals are translated to binary representations, but we don't have to worry about the details of that now.)

What "freely" means for the working mathematician is that the type `nat` has an element `zero` and an injective successor function `succ` whose image does not include `zero`.

```
example (n : nat) : n.succ ≠ nat.zero := nat.succ_ne_zero n

example (m n : nat) (h : m.succ = n.succ) : m = n := nat.succ.inj h
```

What the word "inductively" means for the working mathematician is that the natural numbers comes with a principle of proof by induction and a principle of definition by recursion. This section will show you how to use these.

Here is an example of a recursive definition of the factorial function.

```
def fac : ℕ → ℕ
| 0       := 1
| (n + 1) := (n + 1) * fac n
```

The syntax takes some getting used to. Notice that there is no `:=` on the first line. The next two lines provide the base case and inductive step for a recursive definition. These equations hold definitionally, but they can also be used manually by giving the name `fac` to `simp` or `rw`.

```
example : fac 0 = 1 := rfl
example : fac 0 = 1 := by rw fac
example : fac 0 = 1 := by simp [fac]

example (n : ℕ) : fac (n + 1) = (n + 1) * fac n := rfl
example (n : ℕ) : fac (n + 1) = (n + 1) * fac n := by rw fac
example (n : ℕ) : fac (n + 1) = (n + 1) * fac n := by simp [fac]
```

The factorial function is actually already defined in mathlib as `nat.factorial`. Once again, you can jump to it by typing `#check nat.factorial` and using `ctrl-click`. For illustrative purposes, we will continue using `fac` in the examples. The annotation `@[simp]` before the definition of `nat.factorial` specifies that the defining equation should be added to the database of identities that the simplifier uses by default.

The principle of induction says that we can prove a general statement about the natural numbers by proving that the statement holds of 0 and that whenever it holds of a natural number n, it also holds of $n + 1$. The line `induction n with n ih` in the proof below therefore results in two goals: in the first we need to prove `0 < fac 0`, and in the second we have the added assumption `ih : 0 < fac n` and a required to prove `0 < fac (n + 1)`. The phrase `with n ih` serves to name the variable and the assumption for the inductive hypothesis, and you can choose whatever names you want for them.

```
theorem fac_pos (n : ℕ) : 0 < fac n :=
begin
  induction n with n ih,
  { rw fac, exact zero_lt_one },
  rw fac,
  exact mul_pos n.succ_pos ih,
end
```

The `induction` tactic is smart enough to include hypotheses that depend on the induction variable as part of the induction hypothesis. Step through the next example to see what is going on.

```
theorem dvd_fac {i n : ℕ} (ipos : 0 < i) (ile : i ≤ n) : i ∣ fac n :=
begin
  induction n with n ih,
  { exact absurd ipos (not_lt_of_ge ile) },
  rw fac,
  cases nat.of_le_succ ile with h h,
  { apply dvd_mul_of_dvd_right (ih h) },
  rw h,
  apply dvd_mul_right
end
```

The following example provides a crude lower bound for the factorial function. It turns out to be easier to start with a proof by cases, so that the remainder of the proof starts with the case $n = 1$. See if you can complete the argument with a proof by induction.

```
theorem pow_two_le_fac (n : ℕ) : 2^(n-1) ≤ fac n :=
begin
```

(continues on next page)

(continued from previous page)

```
  cases n with n,
  { simp [fac] },
  sorry
end
```

Induction is often used to prove identities involving finite sums and products. Mathlib defines the expressions `finset.sum s f` where `s : finset α` if a finite set of elements of the type α and `f` is a function defined on α. The codomain of `f` can be any type that supports a commutative, associative addition operation with a zero element. If you import `algebra.big_operators` and issue the command `open_locale big_operators`, you can use the more suggestive notation Σ `x in s, f x`. Of course, there is are an analogous operation and notation for finite products.

We will talk about the `finset` type and the operations it supports in the next section, and again in a later chapter. For now, we will only make use of `finset.range n`, which is the finite set of natural numbers less than `n`.

```
variables {α : Type*} (s : finset ℕ) (f : ℕ → ℕ) (n : ℕ)

#check finset.sum s f
#check finset.prod s f

open_locale big_operators
open finset

example : s.sum f = Σ x in s, f x := rfl
example : s.prod f = Π x in s, f x := rfl

example : (range n).sum f = Σ x in range n, f x := rfl
example : (range n).prod f = Π x in range n, f x := rfl
```

The facts `finset.sum_range_zero` and `finset.sum_range_succ` provide a recursive description summation up to n, and similarly for products.

```
example (f : ℕ → ℕ) : Σ x in range 0, f x = 0 :=
finset.sum_range_zero f

example (f : ℕ → ℕ) (n : ℕ): Σ x in range n.succ, f x = (Σ x in range n, f x) + f n␣
↪:=
finset.sum_range_succ f n

example (f : ℕ → ℕ) : Π x in range 0, f x = 1 :=
finset.prod_range_zero f

example (f : ℕ → ℕ) (n : ℕ): Π x in range n.succ, f x = (Π x in range n, f x) * f n␣
↪:=
finset.prod_range_succ f n
```

The first identity in each pair holds definitionally, which is to say, you can replace the proofs by `rfl`.

The following expresses the factorial function that we defined as a product.

```
example (n : ℕ) : fac n = Π i in range n, (i + 1) :=
begin
  induction n with n ih,
  { simp [fac] },
  simp [fac, ih, prod_range_succ, mul_comm]
end
```

The fact that we include `mul_comm` as a simplification rule deserves comment. It should seem dangerous to simplify with the identity `x * y = y * x`, which would ordinarily loop indefinitely. Lean's simplifier is smart enough to recognize that, and applies the rule only in the case where the resulting term has a smaller value in some fixed but arbitrary ordering of the terms. The following example shows that simplifying using the three rules `mul_assoc`, `mul_comm`, and `mul_left_comm` manages to identify products that are the same up to the placement of parentheses and ordering of variables.

```
example (a b c d e f : ℕ) : a * ((b * c) * f * (d * e)) = d * (a * f * e) * (c * b) :=
by simp [mul_assoc, mul_comm, mul_left_comm]
```

Roughly, the rules work by pushing parentheses to the right and then re-ordering the expressions on both sides until they both follow the same canonical order. Simplifying with these rules, and the corresponding rules for addition, is a handy trick.

Returning to summation identities, we suggest stepping through the following proof that the sum of the natual numbers up to an including n is $n(n+1)/2$. The first step of the proof clears the denominator. This is generally useful when formalizing identities, because calculations with division generally have side conditions. (It is similarly useful to avoid using subtraction on the natural numbers when possible.)

```
theorem sum_id (n : ℕ) : Σ i in range (n + 1), i = n * (n + 1) / 2 :=
begin
  symmetry, apply nat.div_eq_of_eq_mul_right (by norm_num : 0 < 2),
  induction n with n ih,
  { simp },
  rw [finset.sum_range_succ, mul_add 2, ←ih, nat.succ_eq_add_one],
  ring
end
```

We encourage you to prove the analogous identity for sums of squares, and other identities you can find on the web.

```
theorem sum_sqr (n : ℕ) : Σ i in range (n + 1), i^2 = n * (n + 1) * (2 *n + 1) / 6 :=
sorry
```

In Lean's core library, addition and multiplication are themselves defined using recursive definitions, and their fundamental properties are established using induction. If you like thinking about foundational topics like that, you might enjoy working through proofs of the commutativity and associativity of multiplication and addition and the distributivity of multiplication over addition. You can do this on a copy of the natural numbers following the outline below. Notice that we can use the `induction` tactic with `my_nat`; Lean is smart enough to know to use the relevant induction principle (which is, of course, the same as that for `nat`).

We start you off with the commutativity of addition. A good rule of thumb is that because addition and multiplication are defined by recursion on the second argument, it is generally advantageous to do proofs by induction on a variable that occurs in that position. It is a bit tricky to decide which variable to use in the proof of associativity.

It can be confusing to write things without the usual notation for zero, one, addition, and multiplication. We will learn how to define such notation later. Working in the namespace `my_nat` means that we can write `zero` and `succ` rather than `my_nat.zero` and `my_nat.succ`, and that these interpretations of the names take precedence over others. Outside the namespace, the full name of the `add` defined below, for example, is `my_nat.add`.

If you find that you *really* enjoy this sort of thing, try defining truncated subtraction and exponentiation and proving some of their properties as well. Remember that truncated subtraction cuts off at zero. To define that, it is useful to define a predecessor function, `pred`, that subtracts one from any nonzero number and fixes zero. The function `pred` can be defined by a simple instance of recursion.

```
inductive my_nat
| zero : my_nat
| succ : my_nat → my_nat
```

(continues on next page)

(continued from previous page)

```
namespace my_nat

def add : my_nat → my_nat → my_nat
| x zero     := x
| x (succ y) := succ (add x y)

def mul : my_nat → my_nat → my_nat
| x zero     := zero
| x (succ y) := add (mul x y) x

theorem zero_add (n : my_nat) : add zero n = n :=
begin
  induction n with n ih,
  { refl },
  rw [add, ih]
end

theorem succ_add (m n : my_nat) : add (succ m) n = succ (add m n) :=
begin
  induction n with n ih,
  { refl },
  rw [add, ih],
  refl
end

theorem add_comm (m n : my_nat) : add m n = add n m :=
begin
  induction n with n ih,
  { rw zero_add, refl },
  rw [add, succ_add, ih]
end

theorem add_assoc (m n k : my_nat) : add (add m n) k = add m (add n k) :=
sorry

theorem mul_add  (m n k : my_nat) : mul m (add n k) = add (mul m n) (mul m k) :=
sorry

theorem zero_mul (n : my_nat) : mul zero n = zero :=
sorry

theorem succ_mul (m n : my_nat) : mul (succ m) n = add (mul m n) n :=
sorry

theorem mul_comm (m n : my_nat) : mul m n = mul n m :=
sorry

end my_nat
```

5.3 Infinitely Many Primes

Let us continue our exploration of induction and recursion with another mathematical standard: a proof that there are infinitely many primes. One way to formulate this is as the statement that for every natural number n, there is a prime number greater than n. To prove this, let p be any prime factor of $n! + 1$. If p is less than n, it divides $n!$. Since it also divides $n! + 1$, it divides 1, a contradiction. Hence p is greater than n.

To formalize that proof, we need to show that any number greater than or equal to 2 has a prime factor. To do that, we will need to show that any natural number that is not equal to 0 or 1 is greater-than or equal to 2. And this brings us to a quirky feature of formalization: it is often trivial statements like this that are among the most annoying to formalize. Here we consider a few ways to do it.

To start with, we can use the `cases` tactic and the fact that the successor function respects the ordering on the natural numbers.

```
theorem two_le {m : ℕ} (h0 : m ≠ 0) (h1 : m ≠ 1) : 2 ≤ m :=
begin
  cases m, contradiction,
  cases m, contradiction,
  repeat { apply nat.succ_le_succ },
  apply zero_le
end
```

Another strategy is to use the tactic `interval_cases`, which automatically splits the goal into cases when the variable in question is contained in an interval of natural numbers or integers. Remember that you can hover over it to see its documentation.

```
example {m : ℕ} (h0 : m ≠ 0) (h1 : m ≠ 1) : 2 ≤ m :=
begin
  by_contradiction h,
  push_neg at h,
  interval_cases m; contradiction
end
```

Recall that the semicolon after `interval_cases m` means that the next tactic is applied to each of the cases that it generates. Yet another option is to use the tactic, `dec_trivial`, which tries to find a decision procedure to solve the problem. Lean knows that you can decide the truth value of a statement that begins with a bounded quantifier `∀ x, x < n → ...` or `∃ x, x < n ∧ ...` by deciding each of the finitely many instances.

```
example {m : ℕ} (h0 : m ≠ 0) (h1 : m ≠ 1) : 2 ≤ m :=
begin
  by_contradiction h,
  push_neg at h,
  revert m h h0 h1,
  dec_trivial
end
```

In fact, the variant `dec_trivial!` will revert all the hypotheses that contain a variable that is found in the target.

```
example {m : ℕ} (h : m < 2) : m = 0 ∨ m = 1 :=
by dec_trivial!
```

Finally, in this case we can use the `omega` tactic, which is designed to reason about linear expressions in the natural numbers.

```
example {m : ℕ} (h0 : m ≠ 0) (h1 : m ≠ 1) : 2 ≤ m :=
by omega
```

With the theorem `two_le` in hand, let's start by showing that every natural number greater than two has a prime divisor. Mathlib contains a function `nat.min_fac` that returns the smallest prime divisor, but for the sake of learning new parts of the library, we'll avoid using it and prove the theorem directly.

Here, ordinary induction isn't enough. We want to use *strong induction*, which allows us to prove that every natural number n has a property P by showing that for every number n, if P holds of all values less than n, it holds at n as well. In Lean, this principle is called `nat.strong_induction_on`, and we can use the `with` keyword to tell the induction tactic to use it. Notice that when we do that, there is no base case; it is subsumed by the general induction step.

The argument is simply as follows. Assuming $n \geq 2$, if n is prime, we're done. If it isn't, then by one of the characterizations of what it means to be a prime number, it has a nontrivial factor, m, and we can apply the inductive hypothesis to that. Step through the next proof to see how that plays out. The line `dsimp at ih` simplifies the expression of the inductive hypothesis to make it more readable. The proof still works if you delete that line.

```
theorem exists_prime_factor {n : nat} (h : 2 ≤ n) :
  ∃ p : nat, p.prime ∧ p ∣ n :=
begin
  by_cases np : n.prime,
  { use [n, np, dvd_rfl] },
  induction n using nat.strong_induction_on with n ih,
  dsimp at ih,
  rw nat.prime_def_lt at np,
  push_neg at np,
  rcases np h with ⟨m, mltn, mdvdn, mne1⟩,
  have : m ≠ 0,
  { intro mz,
    rw [mz, zero_dvd_iff] at mdvdn,
    linarith },
  have mgt2 : 2 ≤ m := two_le this mne1,
  by_cases mp : m.prime,
  { use [m, mp, mdvdn] },
  rcases ih m mltn mgt2 mp with ⟨p, pp, pdvd⟩,
  use [p, pp, pdvd.trans mdvdn]
end
```

We can now prove the following formulation of our theorem. See if you can fill out the sketch. You can use `nat.factorial_pos`, `nat.dvd_factorial`, and `nat.dvd_sub`.

```
theorem primes_infinite : ∀ n, ∃ p > n, nat.prime p :=
begin
  intro n,
  have : 2 ≤ nat.factorial (n + 1) + 1,
    sorry,
  rcases exists_prime_factor this with ⟨p, pp, pdvd⟩,
  refine ⟨p, _, pp⟩,
  show p > n,
  by_contradiction ple, push_neg at ple,
  have : p ∣ nat.factorial (n + 1),
    sorry,
  have : p ∣ 1,
    sorry,
  show false,
    sorry
end
```

Let's consider a variation of the proof above, where instead of using the factorial function, we suppose that we are given by a finite set $\{p_1, \ldots, p_n\}$ and we consider a prime factor of $\prod_{i=1}^{n} p_i + 1$. That prime factor has to be distinct from each p_i, showing that there is no finite set that contains all the prime numbers.

Formalizing this argument requires us to reason about finite sets. In Lean, for any type α, the type `finset α` represents finite sets of elements of type α. Reasoning about finite sets computationally requires having a procedure to test equality on α, which is why the snippet below includes the assumption `[decidable_eq α]`. For concrete data types like $\mathbb{N}$, $\mathbb{Z}$, and $\mathbb{Q}$, the assumption is satisfied automatically. When reasoning about the real numbers, it can be satisfied using classical logic and abandoning the computational interpretation.

We use the command `open finset` to avail ourselves of shorter names for the relevant theorems. Unlike the case with sets, most equivalences involving finsets do not hold definitionally, so they need to be expanded manually using equivalances like `finset.subset_iff`, `finset.mem_union`, `finset.mem_inter`, and `finset.mem_sdiff`. The `ext` tactic can still be used to reduce show that two finite sets are equal by showing that every element of one is an element of the other.

```
open finset

section
variables {α : Type*} [decidable_eq α] (r s t : finset α)

example : r ∩ (s ∪ t) ⊆ (r ∩ s) ∪ (r ∩ t) :=
begin
  rw subset_iff,
  intro x,
  rw [mem_inter, mem_union, mem_union, mem_inter, mem_inter],
  tauto
end

example : r ∩ (s ∪ t) ⊆ (r ∩ s) ∪ (r ∩ t) :=
by { simp [subset_iff], intro x, tauto }

example : (r ∩ s) ∪ (r ∩ t) ⊆ r ∩ (s ∪ t) :=
by { simp [subset_iff], intro x, tauto }

example : (r ∩ s) ∪ (r ∩ t) = r ∩ (s ∪ t) :=
by { ext x, simp, tauto }

end
```

We have used a new trick: the `tauto` tactic (and a strengthened version, `tauto!`, which uses classical logic) can be used to dispense with propositional tautologies. See if you can use these methods to prove the two examples below.

```
example : (r ∪ s) ∩ (r ∪ t) = r ∪ (s ∩ t) :=
sorry

example : (r \ s \ t) = r \ (s ∪ t) :=
sorry
```

The theorem `finset.dvd_prod_of_mem` tells us that if an `n` is an element of a finite set `s`, then `n` divides `∏ i in s, i`.

```
example (s : finset ℕ) (n : ℕ) (h : n ∈ s) : n ∣ (∏ i in s, i) :=
finset.dvd_prod_of_mem _ h
```

We also need to know that the converse holds in the case where `n` is prime and `s` is a set of primes. To show that, we need the following lemma, which you should be able to prove using the theorem `nat.prime.eq_one_or_self_of_dvd`.

```
theorem nat.prime.eq_of_dvd_of_prime {p q : ℕ}
    (prime_p : nat.prime p) (prime_q : nat.prime q) (h : p ∣ q) :
```

(continues on next page)

(continued from previous page)

```
  p = q :=
sorry
```

We can use this lemma to show that if a prime `p` divides a product of a finite set of primes, then it divides one of them. Mathlib provides a useful principle of induction on finite sets: to show that a property holds of an arbitrary finite set `s`, show that it holds of the empty set, and show that it is preserved when we add a single new element `a ∉ s`. The principle is known as `finset.induction_on`. When we tell the induction tactic to use it, we can also specify the names `a` and `s`, the name for the assumption `a ∉ s` in the inductive step, and the name for the inductive hypothesis. The expression `finset.insert a s` denotes the union of `s` with the singleton `a`. The identities `finset.prod_empty` and `finset.prod_insert` then provide the relevant rewrite rules for the product. In the proof below, the first `simp` applies `finset.prod_empty`. Step through the beginning of the proof to see the induction unfold, and then finish it off.

```
theorem mem_of_dvd_prod_primes {s : finset ℕ} {p : ℕ} (prime_p : p.prime) :
  (∀ n ∈ s, nat.prime n) → (p ∣ ∏ n in s, n) → p ∈ s :=
begin
  intros h₀ h₁,
  induction s using finset.induction_on with a s ans ih,
  { simp at h₁,
    linarith [prime_p.two_le] },
  simp [finset.prod_insert ans, prime_p.dvd_mul] at h₀ h₁,
  rw mem_insert,
  sorry
end
```

We need one last property of finite sets. Given an element `s : set α` and a predicate `P` on α, in Chapter 4 we wrote `{ x ∈ s | P x }` for the set of elements of `s` that satisfy `P`. Given `s : finset α`, the analogous notion is written `s.filter P`.

```
example (s : finset ℕ) (x : ℕ) : x ∈ s.filter nat.prime ↔ x ∈ s ∧ x.prime :=
mem_filter
```

We now prove an alternative formulation of the statement that there are infinitely many primes, namely, that given any `s : finset ℕ`, there is a prime `p` that is not an element of `s`. Aiming for a contradiction, we assume that all the primes are in `s`, and then cut down to a set `s'` that contains all and only the primes. Taking the product of that set, adding one, and finding a prime factor of the result leads to the contradiction we are looking for. See if you can complete the sketch below. You can use `finset.prod_pos` in the proof of the first `have`.

```
theorem primes_infinite' : ∀ (s : finset nat), ∃ p, nat.prime p ∧ p ∉ s :=
begin
  intro s,
  by_contradiction h,
  push_neg at h,
  set s' := s.filter nat.prime with s'_def,
  have mem_s' : ∀ {n : ℕ}, n ∈ s' ↔ n.prime,
  { intro n,
    simp [s'_def],
    apply h },
  have : 2 ≤ (∏ i in s', i) + 1,
    sorry,
  rcases exists_prime_factor this with ⟨p, pp, pdvd⟩,
  have : p ∣ (∏ i in s', i),
    sorry,
  have : p ∣ 1,
  { convert nat.dvd_sub' pdvd this, simp },
```

(continues on next page)

(continued from previous page)

```
  show false,
    sorry
end
```

We have thus seen two ways of saying that there are infinitely many primes: saying that they are not bounded by any `n`, and saying that they are not contained in any finite set `s`. The two proofs below show that these formulations are equivalent. In the second, in order to form `s.filter Q`, we have to assume that there is a procedure for deciding whether or not `Q` holds. Lean knows that there is a procedure for `nat.prime`. In general, if we use classical logic by writing `open_locale classical`, we can dispense with the assumption.

In mathlib, `finset.sup s f` denotes the supremum of the values of `f x` as `x` ranges over `s`, returning `0` in the case where `s` is empty and the codomain of `f` is $\mathbb{N}$. In the first proof, we use `s.sup id`, where `id` is the identity function, to refer to the maximum value in `s`.

```
theorem bounded_of_ex_finset (Q : ℕ → Prop):
  (∃ s : finset ℕ, ∀ k, Q k → k ∈ s) → ∃ n, ∀ k, Q k → k < n :=
begin
  rintros ⟨s, hs⟩,
  use s.sup id + 1,
  intros k Qk,
  apply nat.lt_succ_of_le,
  show id k ≤ s.sup id,
  apply le_sup (hs k Qk)
end

theorem ex_finset_of_bounded (Q : ℕ → Prop) [decidable_pred Q] :
  (∃ n, ∀ k, Q k → k ≤ n) → (∃ s : finset ℕ, ∀ k, Q k ↔ k ∈ s) :=
begin
  rintros ⟨n, hn⟩,
  use (range (n + 1)).filter Q,
  intro k,
  simp [nat.lt_succ_iff],
  exact hn k
end
```

A small variation on our second proof that there are infinitely many primes shows that there are infinitely many primes congruent to 3 modulo 4. The argument goes as follows. First, notice that if the product of two numbers m and n is equal to 3 modulo 4, then one of the two numbers is congruent to three modulo 4. After all, both have to be odd, and if they are both congruent to 1 modulo 4, so is their product. We can use this observation to show that if some number greater than 2 is congruent to 3 modulo 4, then that number has a prime divisor that is also congruent to 3 modulo 4.

Now suppose there are only finitely many prime numbers congruent to 3 modulo 4, say, $p_1, \ldots, p_k$. Without loss of generality, we can assume that $p_1 = 3$. Consider the product $4 \prod_{i=2}^{k} p_i + 3$. It is easy to see that this is congruent to 3 modulo 4, so it has a prime factor p congruent to 3 modulo 4. It can't be the case that $p = 3$; since p divides $4 \prod_{i=2}^{k} p_i + 3$, if p were equal to 3 then it would also divide $\prod_{i=2}^{k} p_i$, which implies that p is equal to one of the p_i for $i = 2, \ldots, k$; and we have excluded 3 from this list. So p has to be one of the other elements p_i. But in that case, p divides $4 \prod_{i=2}^{k} p_i$ and hence 3, which contradicts the fact that it is not 3.

In Lean, the notation `n % m`, read "`n` modulo `m`," denotes the remainder of the division of `n` by `m`.

```
example : 27 % 4 = 3 := by norm_num
```

We can then render the statment "`n` is congruent to 3 modulo 4" as `n % 4 = 3`. The following example and theorems sum up the facts about this function that we will need to use below. The first named theorem is another illustration of reasoning by a small number of cases. In the second named theorem, remember that the semicolon means that the subsequent tactic block is applied to both of the goals that result from the application of `two_le`.

```
example (n : ℕ) : (4 * n + 3) % 4 = 3 :=
by { rw [add_comm, nat.add_mul_mod_self_left], norm_num }

theorem mod_4_eq_3_or_mod_4_eq_3 {m n : ℕ} (h : m * n % 4 = 3) :
  m % 4 = 3 ∨ n % 4 = 3 :=
begin
  revert h,
  rw [nat.mul_mod],
  have : m % 4 < 4 := nat.mod_lt m (by norm_num),
  interval_cases m % 4 with hm; simp [hm],
  have : n % 4 < 4 := nat.mod_lt n (by norm_num),
  interval_cases n % 4 with hn; simp [hn]; norm_num
end

theorem two_le_of_mod_4_eq_3 {n : ℕ} (h : n % 4 = 3) : 2 ≤ n :=
by apply two_le; { intro neq, rw neq at h, norm_num at h }
```

We will also need the following fact, which says that if `m` is a nontrivial divisor of `n`, then so is `m / n`. See if you can complete the proof using `nat.div_dvd_of_dvd` and `nat.div_lt_self`.

```
theorem aux {m n : ℕ} (h₀ : m ∣ n) (h₁ : 2 ≤ m) (h₂ : m < n) :
  (n / m) ∣ n ∧ n / m < n :=
sorry
```

Now put all the pieces together to prove that any number congruent to 3 modulo 4 has a prime divisor with that same property.

```
theorem exists_prime_factor_mod_4_eq_3 {n : nat} (h : n % 4 = 3) :
  ∃ p : nat, p.prime ∧ p ∣ n ∧ p % 4 = 3 :=
begin
  by_cases np : n.prime,
  { use [n, np, dvd_rfl, h] },
  induction n using nat.strong_induction_on with n ih,
  dsimp at ih,
  rw nat.prime_def_lt at np,
  push_neg at np,
  rcases np (two_le_of_mod_4_eq_3 h) with ⟨m, mltn, mdvdn, mne1⟩,
  have mge2 : 2 ≤ m,
  { apply two_le _ mne1,
    intro mz,
    rw [mz, zero_dvd_iff] at mdvdn, linarith },
  have neq : m * (n / m) = n := nat.mul_div_cancel' mdvdn,
  have : m % 4 = 3 ∨ (n / m) % 4 = 3,
  { apply mod_4_eq_3_or_mod_4_eq_3, rw [neq, h] },
  cases this with h1 h1,
  { sorry },
  sorry
end
```

We are in the home stretch. Given a set `s` of prime numbers, we need to talk about the result of removing 3 from that set, if it is present. The function `finset.erase` handles that.

```
example (m n : ℕ) (s : finset ℕ) (h : m ∈ erase s n) : m ≠ n ∧ m ∈ s :=
by rwa mem_erase at h

example (m n : ℕ) (s : finset ℕ) (h : m ∈ erase s n) : m ≠ n ∧ m ∈ s :=
by { simp at h, assumption }
```

We are now ready to prove that there are infinitely many primes congruent to 3 modulo 4. Fill in the missing parts below. Our solution uses `nat.dvd_add_iff_left` and `nat.dvd_sub'` along the way.

```
theorem primes_mod_4_eq_3_infinite : ∀ n, ∃ p > n, nat.prime p ∧ p % 4 = 3 :=
begin
  by_contradiction h,
  push_neg at h,
  cases h with n hn,
  have : ∃ s : finset nat, ∀ p : ℕ, p.prime ∧ p % 4 = 3 ↔ p ∈ s,
  { apply ex_finset_of_bounded,
    use n,
    contrapose! hn,
    rcases hn with ⟨p, ⟨pp, p4⟩, pltn⟩,
    exact ⟨p, pltn, pp, p4⟩ },
  cases this with s hs,
  have h₀ : 2 ≤ 4 * (∏ i in erase s 3, i) + 3,
    sorry,
  have h₁ : (4 * (∏ i in erase s 3, i) + 3) % 4 = 3,
    sorry,
  rcases exists_prime_factor_mod_4_eq_3 h₁ with ⟨p, pp, pdvd, p4eq⟩,
  have ps : p ∈ s,
    sorry,
  have pne3 : p ≠ 3,
    sorry,
  have : p ∣ 4 * (∏ i in erase s 3, i),
    sorry,
  have : p ∣ 3,
    sorry,
  have : p = 3,
    sorry,
  contradiction
end
```

If you managed to complete the proof, congratulations! This has been a serious feat of formalization.

CHAPTER

SIX

ABSTRACT ALGEBRA

Modern mathematics makes essential use of algebraic structures, which encapsulate patterns that can be instantiated in multiple settings. The subject provides various ways of defining such structures and constructing particular instances.

Lean therefore provides corresponding ways of defining structures formally and working with them. You have already seen examples of algebraic structures in Lean, such as rings and lattices, which were discussed in Chapter 2. This chapter will explain the mysterious square bracket annotations that you saw there, `[ring α]` and `[lattice α]`. It will also show you how to define and use algebraic structures on your own.

For more technical detail, you can consult Theorem Proving in Lean, and a paper by Anne Baanen, Use and abuse of instance parameters in the Lean mathematical library.

6.1 Structures

In the broadest sense of the term, a *structure* is a specification of a collection of data, possibly with contraints that the data is required to satisfy. An *instance* of the structure is a particular bundle of data satisfying the constraints. For example, we can specify that a point is a tuple of three real numbers:

```
@[ext] structure point := (x : ℝ) (y : ℝ) (z : ℝ)
```

The `@[ext]` annotation tells Lean to automatically generate theorems that can be used to prove that two instances of a structure are equal when their components are equal, a property known as *extensionality*.

```
#check point.ext

example (a b : point) (hx : a.x = b.x) (hy : a.y = b.y) (hz : a.z = b.z) :
  a = b :=
begin
  ext,
  repeat { assumption }
end
```

We can then define particular instances of the `point` structure. Lean provides multiple ways of doing that.

```
def my_point1 : point :=
{ x := 2,
  y := -1,
  z := 4 }

def my_point2 :=
{ point .
  x := 2,
```

(continues on next page)

(continued from previous page)

```
  y := -1,
  z := 4 }

def my_point3 : point := ⟨2, -1, 4⟩

def my_point4 := point.mk 2 (-1) 4
```

In the first two examples, the fields of the structure are named explicitly. In the first case, because Lean knows that the expected type of `my_point1` is a `point`, you can start the definition by writing an underscore, `_`. Clicking on the light bulb that appears nearby in VS Code will then give you the option of inserting a template definition with the field names listed for you.

The function `point.mk` referred to in the definition of `my_point4` is known as the *constructor* for the `point` structure, because it serves to construct elements. You can specify a different name if you want, like `build`.

```
structure point' := build :: (x : ℝ) (y : ℝ) (z : ℝ)

#check point'.build 2 (-1) 4
```

The next two examples show how to define functions on structures. Whereas the second example makes the `point.mk` constructor explicit, the first example uses anonymous constructors for brevity. Lean can infer the relevant constructor from the indicated type of `add`. It is conventional to put definitions and theorems associated with a structure like `point` in a namespace with the same name. In the example below, because we have opened the `point` namespace, the full name of `add` is `point.add`. When the namespace is not open, we have to use the full name. But remember that it is often convenient to use anonymous projection notation, which allows us to write `a.add b` instead of `point.add a b`. Lean interprets the former as the latter because `a` has type `point`.

```
namespace point

def add (a b : point) : point := ⟨a.x + b.x, a.y + b.y, a.z + b.z⟩

def add' (a b : point) : point :=
{ x := a.x + b.x,
  y := a.y + b.y,
  z := a.z + b.z }

#check add my_point1 my_point2
#check my_point1.add my_point2

end point

#check point.add my_point1 my_point2
#check my_point1.add my_point2
```

Below we will continue to put definitions in the relevant namespace, but we will leave the namespacing commands out of the quoted snippets. To prove properties of the addition function, we can use `rw` to expand the definition and `ext` to reduce an equation between two elements of the structure to equations between the components. Below we use the `protected` keyword so that the name of the theorem is `point.add_comm`, even when the namespace is open. This is helpful when we want to avoid ambiguity with a generic theorem like `add_comm`.

```
protected theorem add_comm (a b : point) : add a b = add b a :=
begin
  rw [add, add],
  ext; dsimp,
  repeat { apply add_comm }
end
```

(continues on next page)

(continued from previous page)

```
example (a b : point) : add a b = add b a :=
by simp [add, add_comm]
```

Because Lean can unfold definitions and simplify projections internally, sometimes the equations we want hold definitionally.

```
theorem add_x (a b : point) : (a.add b).x = a.x + b.x := rfl
```

It is also possible to define functions on structures using pattern matching, in a manner similar to the way we defined recursive functions in Section 5.2. The definitions `add_alt` and `add_alt'` below are essentially the same; the only difference is that we use anonymous constructor notation in the second. Although it is sometimes convenient to define functions this way, the definitional properties are not as convenient. For example, the expressions `add_alt a b` and `add_alt' a b` cannot be simplified until we decompose `a` and `b` into components, which we can do with `cases`, `rcases`, etc.

```
def add_alt : point → point → point
| (point.mk x₁ y₁ z₁) (point.mk x₂ y₂ z₂) := ⟨x₁ + x₂, y₁ + y₂, z₁ + z₂⟩

def add_alt' : point → point → point
| ⟨x₁, y₁, z₁⟩ ⟨x₂, y₂, z₂⟩ := ⟨x₁ + x₂, y₁ + y₂, z₁ + z₂⟩

theorem add_alt_x (a b : point) : (a.add_alt b).x = a.x + b.x :=
by { cases a, cases b, refl }

theorem add_alt_comm (a b : point) : add_alt a b = add_alt b a :=
begin
  rcases a with ⟨xa, ya, za⟩,
  rcases b with ⟨xb, yb, zb⟩,
  rw [add_alt, add_alt],
  ext; dsimp,
  apply add_comm,
  repeat { apply add_comm },
end

example (a b : point) : add_alt a b = add_alt b a :=
begin
  rcases a with ⟨xa, ya, za⟩,
  rcases b with ⟨xb, yb, zb⟩,
  simp [add_alt, add_comm]
end

example : ∀ a b : point, add_alt a b = add_alt b a :=
begin
  rintros ⟨xa, ya, za⟩ ⟨xb, yb, zb⟩,
  simp [add_alt, add_comm]
end

example : ∀ a b : point, add a b = add b a :=
λ ⟨xa, ya, za⟩ ⟨xb, yb, zb⟩, by simp [add, add_comm]
```

Mathematical constructions often involve taking apart bundled information and putting it together again in different ways. It therefore makes sense that Lean and mathlib offer so many ways of doing this efficiently. As an exercise, try proving that `point.add` is associative. Then define scalar multiplication for a point and show that it distributes over addition.

```
protected theorem add_assoc (a b c : point) :
  (a.add b).add c = a.add (b.add c) :=
sorry

def smul (r : ℝ) (a : point) : point :=
sorry

theorem smul_distrib (r : ℝ) (a b : point) :
  (smul r a).add (smul r b) = smul r (a.add b) :=
sorry
```

Using structures is only the first step on the road to algebraic abstraction. We don't yet have a way to link `point.add` to the generic + symbol, or to connect `point.add_comm` and `point.add_assoc` to the generic `add_comm` and `add_assoc` theorems. These tasks belong to the *algebraic* aspect of using structures, and we will explain how to do carry them out in the next section. For now, just think of a structure as a way of bundling together objects and information.

It is especially useful that a structure can specify not only data types but also constraints that the data must satisfy. In Lean, the latter are represented as fields of type `Prop`. For example, the *standard 2-simplex* is defined to be the set of points (x, y, z) satisfying $x \geq 0$, $y \geq 0$, $z \geq 0$, and $x + y + z = 1$. If you are not familiar with the notion, you should draw a picture, and convince yourself that this set is the equilateral triangle in three-space with vertices $(1, 0, 0)$, $(0, 1, 0)$, and $(0, 0, 1)$, together with its interior. We can represent it in Lean as follows:

```
structure standard_two_simplex :=
(x : ℝ)
(y : ℝ)
(z : ℝ)
(x_nonneg : 0 ≤ x)
(y_nonneg : 0 ≤ y)
(z_nonneg : 0 ≤ z)
(sum_eq   : x + y + z = 1)
```

Notice that the last four fields refer to `x`, `y`, and `z`, that is, the first three fields. We can define a map from the two-simplex to itself that swaps `x` and `y`:

```
def swap_xy (a : standard_two_simplex) : standard_two_simplex :=
{ x := a.y,
  y := a.x,
  z := a.z,
  x_nonneg := a.y_nonneg,
  y_nonneg := a.x_nonneg,
  z_nonneg := a.z_nonneg,
  sum_eq   := by rw [add_comm a.y a.x, a.sum_eq] }
```

More interestingly, we can compute the midpoint of two points on the simplex. We need to add `noncomputable theory` in order to use division on the real numbers.

```
noncomputable theory

def midpoint (a b : standard_two_simplex) : standard_two_simplex :=
{ x        := (a.x + b.x) / 2,
  y        := (a.y + b.y) / 2,
  z        := (a.z + b.z) / 2,
  x_nonneg := div_nonneg (add_nonneg a.x_nonneg b.x_nonneg) (by norm_num),
  y_nonneg := div_nonneg (add_nonneg a.y_nonneg b.y_nonneg) (by norm_num),
  z_nonneg := div_nonneg (add_nonneg a.z_nonneg b.z_nonneg) (by norm_num),
  sum_eq   := by { field_simp, linarith [a.sum_eq, b.sum_eq]} }
```

Here we have established `x_nonneg`, `y_nonneg`, and `z_nonneg` with concise proof terms, but establish `sum_eq` in tactic mode, using `by`. You can just as well use a `begin ... end` block for that purpose.

Given a parameter λ satisfying $0 \le \lambda \le 1$, we can take the weighted average $\lambda a + (1 - \lambda)b$ of two points a and b in the standard 2-simplex. We challenge you to define that function, in analogy to the `midpoint` function above.

```
def weighted_average (lambda : real)
    (lambda_nonneg : 0 ≤ lambda) (lambda_le : lambda ≤ 1)
    (a b : standard_two_simplex) :
  standard_two_simplex :=
sorry
```

Structures can depend on parameters. For example, we can generalize the standard 2-simplex to the standard n-simplex for any n. At this stage, you don't have to know anything about the type *fin n* except that it has n elements, and that Lean knows how to sum over it.

```
open_locale big_operators

structure standard_simplex (n : ℕ) :=
(v          : fin n → ℝ)
(nonneg     : ∀ i : fin n, 0 ≤ v i)
(sum_eq_one : Σ i, v i = 1)

namespace standard_simplex

def midpoint (n : ℕ) (a b : standard_simplex n) : standard_simplex n :=
{ v := λ i, (a.v i + b.v i) / 2,
  nonneg :=
    begin
      intro i,
      apply div_nonneg,
      { linarith [a.nonneg i, b.nonneg i] },
      norm_num
    end,
  sum_eq_one :=
    begin
      simp [div_eq_mul_inv, ←finset.sum_mul, finset.sum_add_distrib,
        a.sum_eq_one, b.sum_eq_one],
      field_simp
    end  }

end standard_simplex
```

As an exercise, see if you can define the weighted average of two points in the standard n-simplex. You can use `finset.sum_add_distrib` and `finset.mul_sum` to manipulate the relevant sums.

We have seen that structures can be used to bundle together data and properties. Interestingly, they can also be used to bundle together properties without the data. For example, the next structure, `is_linear`, bundles together the two components of linearity.

```
structure is_linear (f : ℝ → ℝ) :=
(is_additive : ∀ x y, f (x + y) = f x + f y)
(preserves_mul : ∀ x c, f (c * x) = c * f x)

section
variables (f : ℝ → ℝ) (linf : is_linear f)

#check linf.is_additive
```

(continues on next page)

(continued from previous page)

```
#check linf.preserves_mul

end
```

It is worth pointing out that structures are not the only way to bundle together data. The `point` data structure can be defined using the generic type product, and `is_linear` can be defined with a simple `and`.

```
def point'' := ℝ × ℝ × ℝ

def is_linear' (f : ℝ → ℝ) :=
(∀ x y, f (x + y) = f x + f y) ∧ (∀ x c, f (c * x) = c * f x)
```

Generic type constructions can even be used in place of structures with dependencies between their components. For example, the *subtype* construction combines a piece of data with a property. You can think of of the type `preal` in the next example as being the type of positive real numbers. Any `x : preal` has two components: the value, and the property of being positive. You can access these components as `x.val`, which has type $\mathbb{R}$, and `x.property`, which represents the fact `0 < x.val`.

```
def preal := { y : ℝ // 0 < y }

section
variable x : preal

#check x.val
#check x.property

#check x.1
#check x.2

end
```

We could have used subtypes to define the standard 2-simplex, as well as the standard n-simplex for an arbitrary n.

```
def standard_two_simplex' :=
{ p : ℝ × ℝ × ℝ // 0 ≤ p.1 ∧ 0 ≤ p.2.1 ∧ 0 ≤ p.2.2 ∧ p.1 + p.2.1 + p.2.2 = 1 }

def standard_simplex' (n : ℕ) :=
{ v : fin n → ℝ // (∀ i : fin n, 0 ≤ v i) ∧ (Σ i, v i = 1) }
```

Similarly, *Sigma types* are generalizations of ordered pairs, whereby the type of the second component depends on the type of the first.

```
def std_simplex := Σ n : ℕ, standard_simplex n

section
variable s : std_simplex

#check s.fst
#check s.snd

#check s.1
#check s.2

end
```

Given `s : std_simplex`, the first component `s.fst` is a natural number, and the second component is an element of the corresponding simplex `standard_simplex s.fst`. The difference between a Sigma type and a subtype is

that the second component of a Sigma type is data rather than a proposition.

But even though we can use products, subtypes, and Sigma types instead of structures, using structures has a number of advantages. Defining a structure abstracts away the underlying representation and provides custom names for the functions that access the components. This makes proofs more robust: proofs that rely only on the interface to a structure will generally continue to work when we change the definition, as long as we redefine the old accessors in terms of the new definition. Moreover, as we are about to see, Lean provides support for weaving structures together into a rich, interconnected hierarchy, and for managing the interactions between them.

6.2 Algebraic Structures

To clarify what we mean by the phrase *algebraic structure*, it will help to consider some examples.

1. A *partially ordered set* consists of a set P and a binary relation $\leq$ on P that is transitive and antireflexive.
2. A *group* consists of a set G with an associative binary operation, an identity element 1, and a function $g \mapsto g^{-1}$ that returns an inverse for each g in G. A group is *abelian* or *commutative* if the operation is commutative.
3. A *lattice* is a partially ordered set with meets and joins.
4. A *ring* consists of an (addively written) abelian group $(R, +, 0, x \mapsto -x)$ together with an associative multiplication operation $\cdot$ and an identity 1, such that multiplication distributes over addition. A ring is *commutative* if the multiplication is commutative.
5. An *ordered ring* $(R, +, 0, -, \cdot, 1, \leq)$ consists of a ring together with a partial order on its elements, such that *a le b* implies $a + c \leq b + c$ for every a, b, and c in R, and $0 \leq a$ and $0 \leq b$ implies $0 \leq ab$ for every a and b in R.
6. A *metric space* consists of a set X and a function $d : X \times X \to \mathbb{R}$ such that the following hold:
   - $d(x, y) \geq 0$ for every x and y in X.
   - $d(x, y) = 0$ if and only if $x = y$.
   - $d(x, y) = d(y, x)$ for every x and y in X.
   - $d(x, z) \leq d(x, y) + d(y, z)$ for every x, y, and z in X.
7. A *topological space* consists of a set X and a collection $\mathcal{T}$ of subsets of X, called the *open subsets of* X, such that the following hold:
   - The empty set is open.
   - The intersection of two open sets is open.
   - An arbitrary union of open sets is open.

In each of these examples, the elements of the structure belong to a set, the *carrier set*, that sometimes stands proxy for the entire structure. For example, when we say "let G be a group" and then "let $g \in G$," we are using G to stand for both the structure and its carrier. Not every algebraic structure is associated with a single carrier set in this way. For example, a *bipartite graph* involves a relation between two sets, as does a *Galois connection*, A *category* also involves two sets of interest, commonly called the *objects* and the *morphisms*.

The examples indicate some of the things that a proof assistant has to do in order to support algebraic reasoning. First, it needs to recognize concrete instances of structures. The number systems $\mathbb{Z}$, $\mathbb{Q}$, and $\mathbb{R}$ are all ordered rings, and we should be able to apply a generic theorem about ordered rings in any of these instances. Sometimes a concrete set may the an instance of a structure in more than one way. For example, in addition to the usual topology on $\mathbb{R}$, which forms the basis for real analysis, we can also consider the *discrete* topology on $\mathbb{R}$, in which every set is open.

Second, a proof assistant needs to support generic notation on structures. In Lean, the notation $*$ is used for multiplication in all the usual number systems, as well as for multiplication in generic groups and rings. When we use an expression like

`f x * y`, Lean has to use information about the types of `f`, `x`, and `y` to determine which multiplication we have in mind.

Third, it needs to deal with the fact that structures can inherit definitions, theorems, and notation from other structures in various ways. Some structures extend others by adding more axioms. A commutative ring is still a ring, so any definition that makes sense in a ring also makes sense in a commutative ring, and any theorem that holds in a ring also holds in a commutative ring. Some structures extend others by adding more data. For example, the additive part of any ring is an additive group. The ring structure adds a multiplication and an identity, as well as axioms that govern them and relate them to the additive part. Sometimes we can define one structure in terms of another. Any metric space has a canonical topology associated with it, the *metric space topology*, and there are various topologies that can be associated with any linear ordering.

Finally, it is important to keep in mind that mathematics allows us to use functions and operations to define structures in the same way we use functions and operations to define numbers. Products and powers of groups are again groups. For every n, the integers modulo n form a ring, and for every $k > 0$, the $k \times k$ matrices of polynomials with coefficients in that ring again form a ring. Thus we can calculate with structures just as easily as we can calculate with their elements. This means that algebraic structures lead dual lives in mathematics, as containers for collections of objects and as objects in their own right. A proof assistant has to accommodate this dual role.

When dealing with elements of a type that has an algebraic structure associated with it, a proof assistant needs to recognize the structure and find the relevant definitions, theorems, and notation. All this should sound like a lot of work, and it is. But Lean uses a small collection of fundamental mechanisms to carry out these tasks. The goal of this section is to explain these mechanisms and show you how to use them.

The first ingredient is almost too obvious to mention: formally speaking, algebraic structures are structures in the sense of the Section 6.1. An algebraic structure is a specification of a bundle of data satisfying some axiomatic hypotheses, and we saw in the Section 6.1 that this is exactly what the `structure` command is designed to accommodate. It's a marriage made in heaven!

Given a data type α, we can define the group structure on α as follows.

```
structure group₁ (α : Type*) :=
(mul: α → α → α)
(one: α)
(inv: α → α)
(mul_assoc : ∀ x y z : α, mul (mul x y) z = mul x (mul y z))
(mul_one: ∀ x : α, mul x one = x)
(one_mul: ∀ x : α, mul x one = x)
(mul_left_inv : ∀ x : α, mul (inv x) x = one)
```

Notice that the type α is a *parameter* in the definition of `group₁`. So you should think of an object `struc : group₁ α` as being a group structure on α. We saw in Section 2.2 that the counterpart `mul_right_inv` to `mul_left_inv` follows from the other group axioms, so there is no need to add it to the definition.

This definition of a group is similar to the definition of `group` in mathlib, and we have chosen the name `group₁` to distinguish our version. If you write `#check group` and ctrl-click on the definition, you will see that the mathlib version of `group` is defined to extend another structure; we will explain how to do that later. If you type `#print group` you will also see that the mathlib version of `group` has a number of extra fields. For reasons we will explain later, sometimes it is useful to add redundant information to a structure, so that there are additional fields for objects and functions that can be defined from the core data. Don't worry about that for now. Rest assured that our simplified version `group₁` is morally the same as the definition of a group that mathlib uses.

It is sometimes useful to bundle the type together with the structure, and mathlib also contains a definition of a `Group` structure that is equivalent to the following:

```
structure Group₁ :=
(α : Type*)
(str : group₁ α)
```

The mathlib version is found in `algebra.category.Group.basic`, and you can `#check` it if you add this to the imports at the beginning of the examples file.

For reasons that will become clearer below, it is more often useful to keep the type α separate from the structure `group` α. We refer to the two objects together as a *partially bundled structure*, since the representation combines most, but not all, of the components into one structure. It is common in mathlib to use capital roman letters like `G` for a type when it is used as the carrier type for a group.

Let's construct a group, which is to say, an element of the `group₁` type. For any pair of types α and β, Mathlib defines the type `equiv` α β of *equivalences* between α and β. Mathlib also defines the suggestive notation $\alpha \simeq \beta$ for this type. An element `f :` $\alpha \simeq \beta$ is a bijection between α and β represented by four components: a function `f.to_fun` from α to β, the inverse function `f.inv` from β to α, and two properties that specify these functions are indeed inverse to one another.

```
variables (α β γ : Type*)
variables (f : α ≃ β) (g : β ≃ γ)

#check equiv α β
#check (f.to_fun : α → β)
#check (f.inv_fun : β → α)
#check (f.right_inv: ∀ x : β, f (f.inv_fun x) = x)
#check (f.left_inv: ∀ x : α, f.inv_fun (f x) = x)

#check (equiv.refl α : α ≃ α)
#check (f.symm : β ≃ α)
#check (f.trans g : α ≃ γ)
```

Notice the creative naming of the last three constructions. We think of the identity function `equiv.refl`, the inverse operation `equiv.symm`, and the composition operation `equiv.trans` as explicit evidence that the property of being in bijective correspondence is an equivalence relation.

Notice also that `f.trans g` requires composing the forward functions in reverse order. Mathlib has declared a *coercion* from `equiv` α β to the function type $\alpha \to \beta$, so we can omit writing `.to_fun` and have Lean insert it for us.

```
example (x : α) : (f.trans g).to_fun x = g.to_fun (f.to_fun x) := rfl

example (x : α) : (f.trans g) x = g (f x) := rfl

example : (f.trans g : α → γ) = g ∘ f := rfl
```

Mathlib also defines the type `perm` α of equivalences between α and itself.

```
example (α : Type*) : equiv.perm α = (α ≃ α) := rfl
```

It should be clear that `perm` α forms a group under composition of equivalences. We orient things so that `mul f g` is equal to `g.trans f`, whose forward function is `f ∘ g`. In other words, multiplication is what we ordinarily think of as composition of the bijections. Here we define this group:

```
def perm_group {α : Type*} : group₁ (equiv.perm α) :=
{ mul          := λ f g, equiv.trans g f,
  one          := equiv.refl α,
  inv          := equiv.symm,
  mul_assoc    := λ f g h, (equiv.trans_assoc _ _ _).symm,
  one_mul      := equiv.trans_refl,
  mul_one      := equiv.refl_trans,
  mul_left_inv := equiv.self_trans_symm }
```

In fact, mathlib defines exactly this `group` structure on `equiv.perm` α in the file `group_theory.perm.basic`.

As always, you can hover over the theorems used in the definition of `perm_group` to see their statements, and you can jump to their definitions in the original file to learn more about how they are implemented.

In ordinary mathematics, we generally think of notation as independent of structure. For example, we can consider groups $(G_1, \cdot, 1, \cdot^{-1})$, $(G_2, \circ, e, i(\cdot))$, and $(G_3, +, 0, -)$. In the first case, we write the binary operation as $\cdot$, the identity at 1, and the inverse function as $x \mapsto x^{-1}$. In the second and third cases, we use the notational alternatives shown. When we formalize the notion of a group in Lean, however, the notation is more tightly linked to the structure. In Lean, the components of any `group` are named `mul`, `one`, and `inv`, and in a moment we will see how multplicative notation is set up to refer to them. If we want to use additive notation, we instead use an isomorphic structure `additive_group`. Its components are named `add`, `zero`, and `neg`, and the associated notation is what you would expect it to be.

Recall the type `point` that we defined in Section 6.1, and the addition function that we defined there. These definitions are reproduced in the examples file that accompanies this section. As an exercise, define an $\texttt{add\_group}_1$ structure that is similar to the $\texttt{group}_1$ structure we defined above, except that it uses the additive naming scheme just described. Define negation and a zero on the `point` data type, and define the $\texttt{add\_group}_1$ structure on `point`.

```
structure add_group₁ (α : Type*) :=
(add : α → α → α)
-- fill in the rest

@[ext] structure point := (x : ℝ) (y : ℝ) (z : ℝ)

namespace point

def add (a b : point) : point := ⟨a.x + b.x, a.y + b.y, a.z + b.z⟩

def neg (a b : point) : point := sorry

def zero : point := sorry

def add_group_point : add_group point := sorry

end point
```

We are making progress. Now we know how to define algebraic structures in Lean, and we know how to define instances of those structures. But we also want to associate notation with structures so that we can use it with each instance. Moreover, we want to arrange it so that we can define an operation on a structure and use it with any particular instance, and we want to arrange it so that we can prove a theorem about a structure and use it with any instance.

In fact, mathlib is already set up to use generic group notation, definitions, and theorems for `equiv.perm` α.

```
variables {α : Type*} (f g : equiv.perm α) (n : ℕ)

#check f * g
#check mul_assoc f g g⁻¹

-- group power, defined for any group
#check g^n

example : f * g * (g⁻¹) = f :=
by { rw [mul_assoc, mul_right_inv, mul_one] }

example : f * g * (g⁻¹) = f := mul_inv_cancel_right f g

example {α : Type*} (f g : equiv.perm α) : g.symm.trans (g.trans f) = f :=
mul_inv_cancel_right f g
```

You can check that this is not the case for the additive group structure on `point` that we asked you to define above. Our task now is to understand that magic that goes on under the hood in order to make the examples for `equiv.perm` α

work the way they do.

The issue is that Lean needs to be able to *find* the relevant notation and the implicit group structure, using the information that is found in the expressions that we type. Simiarly, when we write `x + y` with expressions `x` and `y` that have type $\mathbb{R}$, Lean needs to interpret the + symbol as the relevant addition function on the reals. It also has to recognize the type $\mathbb{R}$ as an instance of a commutative ring, so that all the definitions and theorems for a commutative ring are available. For another example, continuity is defined in Lean relative to any two topological spaces. When we have `f : ℝ → ℂ` and we write `continuous f`, Lean has to find the relevant topologies on $\mathbb{R}$ and $\mathbb{C}$.

The magic is achieved with a combination of three things.

1. *Logic.* A definition that should be interpreted in any group takes, as arguments, the type of the group and the group structure as arguments. Similarly, a theorem about the elements of an arbitrary group begins with universal quantifiers over the type of the group and the group structure.
2. *Implicit arguments.* The arguments for the type and the structure are generally left implicit, so that we do not have to write them or see them in the Lean information window. Lean fills the information in for us silently.
3. *Type class inference.* Also known as *class inference*, this is a simple but powerful mechanism that enables us to register information for Lean to use later on. When Lean is called on to fill in implicit arguments to a definition, theorem, or piece of notation, it can make use of information that has been registered.

Whereas an annotation `(grp : group G)` tells Lean that it should expect to be given that argument explicitly and the annotation `{grp : group G}` tells Lean that it should try to figure it out from contextual cues in the expression, the annotation `[grp : group G]` tells Lean that the corresponding argument should be synthesized using type class inference. Since the whole point to the use of such arguments is that we generally do not need to refer to them explicitly, Lean allows us to write `[group G]` and leave the name anonymous. You have probably already noticed that Lean chooses names like `_inst_1` automatically. When we use the anonymous square-bracket annotation with the `variables` command, then as long as the variables are still in scope, Lean automatically adds the argument `[group G]` to any definition or theorem that mentions `G`.

How do we register the information that Lean needs to use to carry out the search? Returning to our group example, we need only make two changes. First, instead of using the `structure` command to define the group structure, we use the keyword `class` to indicate that it is a candidate for class inference. Second, instead of defining particular instances with `def`, we use the keyword `instance` to register the particular instance with Lean. As with the names of class variables, we are allowed to leave the name of an instance definition anonymous, since in general we intend Lean to find it and put it to use without troubling us with the details.

```
class group₂ (α : Type*) :=
(mul: α → α → α)
(one: α)
(inv: α → α)
(mul_assoc : ∀ x y z : α, mul (mul x y) z = mul x (mul y z))
(mul_one: ∀ x : α, mul x one = x)
(one_mul: ∀ x : α, mul x one = x)
(mul_left_inv : ∀ x : α, mul (inv x) x = one)

instance {α : Type*} : group₂ (equiv.perm α) :=
{ mul          := λ f g, equiv.trans g f,
  one          := equiv.refl α,
  inv          := equiv.symm,
  mul_assoc    := λ f g h, (equiv.trans_assoc _ _ _).symm,
  one_mul      := equiv.trans_refl,
  mul_one      := equiv.refl_trans,
  mul_left_inv := equiv.self_trans_symm }
```

The following illustrates their use.

```
#check @group₂.mul

def my_square {α : Type*} [group₂ α] (x : α) := group₂.mul x x

#check @my_square

section
variables {β : Type*} (f g : equiv.perm β)

example : group₂.mul f g = g.trans f := rfl

example : my_square f = f.trans f := rfl

end
```

The `#check` command shows that `group₂.mul` has an implicit argument `[group₂ α]` that we expect to be found by class inference, where α is the type of the arguments to `group₂.mul`. In other words, `{α : Type*}` is the implicit argument for the type of the group elements and `[group₂ α]` is the implicit argument for the group structure on α. Similarly, when we define a generic squaring function `my_square` for `group₂`, we use an implicit argument `{α : Type*}` for the type of the elements and an implicit argument `[group₂ α]` for the `group₂` structure.

In the first example, when we write `group₂.mul f g`, the type of `f` and `g` tells Lean that in the argument α to `group₂.mul` has to be instantiated to `equiv.perm β`. That means that Lean has to find an element of `group₂ (equiv.perm β)`. The previous `instance` declaration tells Lean exactly how to do that. Problem solved!

This simple mechanism for registering information so that Lean can find it when it needs it is remarkably useful. Here is one way it comes up. In Lean's foundation, a data type α may be empty. In a number of applications, however, it is useful to know that a type has at least one element. For example, the function `list.head`, which returns the first element of a list, can return the default value when the list is empty. To make that work, the Lean library defines a class `inhabited α`, which does nothing more than store a default value. We can show that the `point` type is an instance:

```
instance : inhabited point := { default := ⟨0, 0, 0⟩ }

#check (default : point)

example : ([] : list point).head = default := rfl
```

The class inference mechanism is also used for generic notation. The expression `x + y` is an abbreviation for `has_add.add x y` where—you guessed it—`has_add α` is a class that stores a binary function on α. Writing `x + y` tells Lean to find a registered instance of `[has_add.add α]` and use the corresponding function. Below, we register the addition function for `point`.

```
instance : has_add point := { add := point.add }

section
variables x y : point

#check x + y

example : x + y = point.add x y := rfl

end
```

In this way, we can assign the notation + to binary operations on other types as well.

But we can do even better. We have seen that * can be used in any group, + can be used in any additive group, and both can be used in any ring. When we define a new instance of a ring in Lean, we don't have to define + and * for that

instance, because Lean knows that these are defined for every ring. We can use this method to specify notation for our `group₂` class:

```
instance has_mul_group₂ {α : Type*} [group₂ α] : has_mul α := ⟨group₂.mul⟩

instance has_one_group₂ {α : Type*} [group₂ α] : has_one α := ⟨group₂.one⟩

instance has_inv_group₂ {α : Type*} [group₂ α] : has_inv α := ⟨group₂.inv⟩

section
variables {α : Type*} (f g : equiv.perm α)

#check f * 1 * g⁻¹

def foo: f * 1 * g⁻¹ = g.symm.trans ((equiv.refl α).trans f) := rfl

end
```

In this case, we have to supply names for the instances, because Lean has a hard time coming up with good defaults. What makes this approach work is that Lean carries out a recursive search. According to the instances we have declared, Lean can find an instance of `has_mul (equiv.perm α)` by finding an instance of `group₂ (equiv.perm α)`, and it can find an instance of `group₂ (equiv.perm α)` because we have provided one. Lean is capable of finding these two facts and chaining them together.

The example we have just given is dangerous, because Lean's library also has an instance of `group (equiv.perm α)`, and multiplication is defined on any group. So it is ambiguous as to which instance is found. In fact, Lean favors more recent declarations unless you explicitly specify a different priority. Also, there is another way to tell Lean that one structure is an instance of another, using the `extends` keyword. This is how `mathlib` specifies that, for example, every commutative ring is a ring. You can find more information in a section on class inference in *Theorem Proving in Lean*.

In general, it is a bad idea to specify a value of `*` for an instance of an algebraic structure that already has the notation defined. Redefining the notion of `group` in Lean is an artificial example. In this case, however, both interpretations of the group notation unfold to `equiv.trans`, `equiv.refl`, and `equiv.symm`, in the same way.

As a similarly artificial exercise, define a class `add_group₂` in analogy to `group₂`. Define the usual notation for addition, negation, and zero on any `add_group₂` using the classes `has_add`, `has_neg`, and `has_zero`. Then show `point` is an instance of `add_group₂`. Try it out and make sure that the additive group notation works for elements of `point`.

```
class add_group₂ (α : Type*) :=
(add : α → α → α)
-- fill in the rest
```

It is not a big problem that we have already declared instances `has_add`, `has_neg`, and `has_zero` for `point` above. Once again, the two ways of synthesizing the notation should come up with the same answer.

Class inference is subtle, and you have to be careful when using it, because it configures automation that invisibly governs the interpretation of the expressions we type. When used wisely, however, class inference is a powerful tool. It is what makes algebraic reasoning possible in Lean.

CHAPTER

SEVEN

TOPOLOGY

Calculus is based on the concept of a function, which is used to model quantities that depend on one another. For example, it is common to study quantities that change over time. The notion of a *limit* is also fundamental. We may say that the limit of a function $f(x)$ is a value b as x approaches a value a, or that $f(x)$ *converges to* b as x approaches a. Equivalently, we may say that a $f(x)$ approaches a as x approaches a value b, or that it *tends to* b as x tends to a. We have already begun to consider such notions in Section 3.6.

Topology is the abstract study of limits and continuity. In this chapter, we will see how topological notions are formalized in mathlib. Not only do topological abstractions apply in much greater generality, but that also, somewhat paradoxically, make it easier to reason about limits and continuity in concrete instances.

Topological notions build on quite a few layers of mathematical structure. The first layer is naive set theory, as described in Chapter 4. The next layer is the theory of *filters*, which we will describe in Section 7.1. On top of that, we layer the theories of *topological spaces*, *metric spaces*, and a slightly more exotic intermediate notion called a *uniform space*.

Whereas previous chapters relied on mathematical notions that were likely familiar to you, the notion of a filter less well known, even to many working mathematicians. The notion is essential, however, for formalizing mathematics effectively. Let us explain why. Let `f : ℝ → ℝ` be any function. We can consider the limit of `f x` as `x` approaches some value x_0, but we can also consider the limit of `f x` as `x` approaches infinity or negative infinity. We can moreover consider the limit of `f x` as `x` approaches x_0 from the right, conventionally written x_0^+, or from the left, written x_0^-. There are variations where `x` approaches x_0 or x_0^+ or x_0^- but is not allowed to take on the value x_0 itself. This results in at least eight ways that `x` can approach something. We can also restrict to rational values of `x` or place other constraints on the domain, but let's stick to those 8 cases.

We have a similar variety of options on the codomain: we can specify that `f x` approaches a value from the left or right, or that it approaches positive or negative infinity, and so on. For example, we may wish to say that `f x` tends to $+\infty$ when `x` tends to x_0 from the right without being equal to x_0. This results in 64 different kinds of limit statements, and we haven't even begun to deal with limits of sequences, as we did in Section 3.6.

The problem is compounded even further when it comes to the supporting lemmas. For instance, limits compose: if `f x` tends to y_0 when `x` tends to x_0 and `g y` tends to z_0 when `y` tends to y_0 then `g ∘ f x` tends to z_0 when `x` tends to x_0. There are three notions of "tends to" at play here, each of which can be instantiated in any of the eight ways described in the previous paragraph. This results in 512 lemmas, a lot to have to add to a library! Informally, mathematicians generally prove two or three of these and simply note that the rest can be proved "in the same way." Formalizing mathematics requires making the relevant notion of "sameness" fully explicit, and that is exactly what Bourbaki's theory of filters manages to do.

7.1 Filters

A *filter* on a type `X` is a collection of sets of `X` that satisfies three conditions that we will spell out below. The notion supports two related ideas:

- *limits*, including all the kinds of limits discussed above: finite and infinite limits of sequences, finite and infinite limits of functions at a point or at infinity, and so on.
- *things happening eventually*, including things happening for large enough `n : ℕ`, or sufficiently near a point `x`, or for sufficiently close pairs of points, or almost everywhere in the sense of measure theory. Dually, filters can also express the idea of *things happening often*: for arbitrarily large `n`, at a point in any neighborhood of given a point, etc.

The filters that correspond to these descriptions will be defined later in this section, but we can already name them:

- `(at_top : filter ℕ)`, made of sets of ℕ containing `{n | n ≥ N}` for some `N`
- $\mathcal{N}$ `x`, made of neighborhoods of `x` in a topological space
- $\mathcal{U}$ `X`, made of entourages of a uniform space (uniform spaces generalize metric spaces and topological groups)
- μ`.a_e`, made of sets whose complement has zero measure with respect to a measure μ.

The general definition is as follows: a filter `F : filter X` is a collection of sets `F.sets : set (set X)` satisfying the following:

- `F.univ_sets : univ ∈ F.sets`
- `F.sets_of_superset : ∀ {U V}, U ∈ F.sets → U ⊆ V → V ∈ F.sets`
- `F.inter_sets : ∀ {U V}, U ∈ sets → V ∈ sets → U ∩ V ∈ sets.`

The first condition says that the set of all elements of `X` belongs to `F.sets`. The second condition says that if `U` belongs to `F.sets` then anything containing `U` also belongs to `F.sets`. The third condition says that `F.sets` is closed under finite intersections. In mathlib, a filter `F` is defined to be a structure bundling `F.sets` and its three properties, but the properties carry no additional data, and it is convenient to blur the distinction between `F` and `F.sets`. We therefore define `U ∈ F` to mean `U ∈ F.sets`. This explains why the word `sets` appears in the names of some lemmas that that mention `U ∈ F`.

It may help to think of a filter as defining a notion of a "sufficiently large" set. The first condition then says that `univ` is sufficiently large, the second one says that a set containing a sufficiently large set is sufficiently large and the third one says that the intersection of two sufficiently large sets is sufficiently large.

It may be even more useful to think of a filter on a type `X` as a generalized element of `set X`. For instance, `at_top` is the "set of very large numbers" and $\mathcal{N}$ x_0 is the "set of points very close to x_0." One manifestation of this view is that we can associate to any `s : set X` the so-called *principal filter* consisting of all sets that contain `s`. This definition is already in mathlib and has a notation $\mathcal{P}$ (localized in the `filter` namespace). For the purpose of demonstration, we ask you to take this opportunity to work out the definition here.

```
def principal {α : Type*} (s : set α) : filter α :=
{ sets := {t | s ⊆ t},
  univ_sets := sorry,
  sets_of_superset := sorry,
  inter_sets := sorry}
```

For our second example, we ask you to define the filter `at_top : filter ℕ`. (We could use any type with a preorder instead of ℕ.)

```
example : filter ℕ :=
{ sets := {s | ∃ a, ∀ b, a ≤ b → b ∈ s},
```

(continues on next page)

(continued from previous page)

```
  univ_sets := sorry,
  sets_of_superset := sorry,
  inter_sets := sorry }
```

We can also directly define the filter $\mathcal{N}$ `x` of neighborhoods of any `x : ℝ`. In the real numbers, a neighborhood of `x` is a set containing an open interval $(x_0 - \varepsilon, x_0 + \varepsilon)$, defined in mathlib as `Ioo (x₀ - ε) (x₀ + ε)`. (This is notion of a neighborhood is only a special case of a more general construction in mathlib.)

With these examples, we can already define what is means for a function `f : X → Y` to converge to some `G : filter Y` along some `F : filter X`, as follows:

```
def tendsto₁ {X Y : Type*} (f : X → Y) (F : filter X) (G : filter Y) :=
∀ V ∈ G, f ⁻¹' V ∈ F
```

When `X` is $\mathbb{N}$ and `Y` is $\mathbb{R}$, `tendsto₁ u at_top (𝓝 x)` is equivalent to saying that the sequence `u : ℕ → ℝ` converges to the real number `x`. When both `X` and `Y` are $\mathbb{R}$, `tendsto f (𝓝 x₀) (𝓝 y₀)` is equivalent to the familiar notion $\lim_{x \to x_0} f(x) = y_0$. All of the other kinds of limits mentioned in the introduction are also equivalent to instances of `tendsto₁` for suitable choices of filters on the source and target.

The notion `tendsto₁` above is definitionally equivalent to the notion `tendsto` that is defined in mathlib, but the latter is defined more abstractly. The problem with the definition of `tendsto₁` is that it exposes a quantifier and elements of `G`, and it hides the intuition that we get by viewing filters as generalized sets. We can hide the quantifier `∀ V` and make the intuition more salient by using more algebraic and set-theoretic machinery. The first ingredient is the *pushforward* operation f_* associated to any map `f : X → Y`, denoted `filter.map f` in mathlib. Given a filter `F` on `X`, `filter.map f F : filter Y` is defined so that `V ∈ filter.map f F ↔ f ⁻¹' V ∈ F` holds definitionally. In this examples file we've opened the `filter` namespace so that `filter.map` can be written as `map`. This means that we can rewrite the definition of `tendsto` using the order relation on `filter Y`, which is reversed inclusion of the set of members. In other words, given `G H : filter Y`, we have `G ≤ H ↔ ∀ V : set Y, V ∈ H → V ∈ G`.

```
def tendsto₂ {X Y : Type*} (f : X → Y) (F : filter X) (G : filter Y) :=
map f F ≤ G

example {X Y : Type*} (f : X → Y) (F : filter X) (G : filter Y) :
  tendsto₂ f F G ↔ tendsto₁ f F G := iff.rfl
```

It may seem that the order relation on filters is backward. But recall that we can view filters on `X` as generalized elements of `set X`, via the inclusion of `𝒫 : set X → filter X` which maps any set `s` to the corresponding principal filter. This inclusion is order preserving, so the order relation on `filter` can indeed be seen as the natural inclusion relation between generalized sets. In this analogy, pushforward is analogous to the direct image. And, indeed, `map f (𝒫 s) = 𝒫 (f '' s)`.

We can now understand intuitively why a sequence `u : ℕ → ℝ` converges to a point x_0 if and only if we have `map u at_top ≤ 𝓝 x₀`. The inequality means the "direct image under `u`" of "the set of very big natural numbers" is "included" in "the set of points very close to x_0."

As promised, the definition of `tendsto₂` does not exhibit any quantifiers or sets. It also leverages the algebraic properties of the pushforward operation. First, each `filter.map f` is monotone. And, second, `filter.map` is compatible with composition.

```
#check (@filter.map_mono : ∀ {α β} {m : α → β}, monotone (map m))
#check (@filter.map_map : ∀ {α β γ} {f : filter α} {m : α → β} {m' : β → γ},
                          map m' (map m f) = map (m' ∘ m) f)
```

Together these two properties allow us to prove that limits compose, yielding in one shot all 256 variants of the composition lemma described in the introduction, and lots more. You can practice proving the following statement using either the

definition of `tendsto₁` in terms of the universal quantifier or the algebraic definition, together with the two lemmas above.

```
example {X Y Z : Type*} {F : filter X} {G : filter Y} {H : filter Z} {f : X → Y} {g :␣
↪Y → Z}
  (hf : tendsto₁ f F G) (hg : tendsto₁ g G H) : tendsto₁ (g ∘ f) F H :=
sorry
```

The pushforward construction uses a map to push filters from the map source to the map target. There also a *pullback* operation, `filter.comap`, going in the other direction. This generalizes the preimage operation on sets. For any map `f`, `filter.map f` and `filter.comap f` form what is known as a *Galois connection*, which is to say, they satisfy

```
filter.map_le_iff_le_comap : filter.map f F ≤ G ↔ F ≤ filter.comap f
G
```

for every `F` and `G`. This operation could be used to provided another formulation of `tendsto` that would be provably (but not definitionaly) equivalent to the one in mathlib.

The `comap` operation can be used to restrict filters to a subtype. For instance, suppose we have `f : ℝ → ℝ`, `x₀ : ℝ` and `y₀ : ℝ`, and suppose we want to state that `f x` approaches `y₀` when `x` approaches `x₀` within the rational numbers. We can pull the filter `𝓝 x₀` back to `ℚ` using the coercion map `coe : ℚ → ℝ` and state `tendsto (f ∘ coe : ℚ → ℝ) (comap coe (𝓝 x₀)) (𝓝 y₀)`.

```
variables (f : ℝ → ℝ) (x₀ y₀ : ℝ)

#check comap (coe : ℚ → ℝ) (𝓝 x₀)
#check tendsto (f ∘ coe) (comap (coe : ℚ → ℝ) (𝓝 x₀)) (𝓝 y₀)
```

The pullback operation is also compatible with composition, but it *contravariant*, which is to say, it reverses the order of the arguments.

```
section
variables {α β γ : Type*} (F : filter α) {m : γ → β} {n : β → α}

#check (comap_comap : comap m (comap n F) = comap (n ∘ m) F)
end
```

Let's now shift attention to the plane `ℝ × ℝ` and try to understand how the neighborhoods of a point `(x₀, y₀)` are related to `𝓝 x₀` and `𝓝 y₀`. There is a product operation `filter.prod : filter X → filter Y → filter (X × Y)`, denoted by `×ᶠ`, which answers this question:

```
example : 𝓝 (x₀, y₀) = 𝓝 x₀ ×ᶠ 𝓝 y₀ := nhds_prod_eq
```

The product operation is defined in terms of the pullback operation and the `inf` operation:

```
F ×ᶠ G = (comap prod.fst F) ⊓ (comap prod.snd G).
```

Here the `inf` operation refers to the lattice structure on `filter X` for any type `X`, whereby `F ⊓ G` is the greatest filter that is smaller than both `F` and `G`. Thus the `inf` operation generalizes the notion of the intersection of sets.

A lot of proofs in mathlib use all of the aforementioned structure (`map`, `comap`, `inf`, `sup`, and `prod`) to give algebraic proofs about convergence without ever referring to members of filters. You can practice doing this in a proof of the following lemma, unfolding the definition of `tendsto` and `filter.prod` if needed.

```
#check le_inf_iff

example (f : ℕ → ℝ × ℝ) (x₀ y₀ : ℝ) :
  tendsto f at_top (𝓝 (x₀, y₀)) ↔
```

(continues on next page)

(continued from previous page)

```
    tendsto (prod.fst ∘ f) at_top (𝓝 x₀) ∧ tendsto (prod.snd ∘ f) at_top (𝓝 y₀) :=
sorry
```

The ordered type `filter X` is actually a *complete* lattice, which is to say, there is a bottom element, there is a top element, and every set of filters on `X` has an `Inf` and a `Sup`.

Note that given the second property in the definition of a filter (if `U` belongs to `F` then anything larger than `U` also belongs to `F`), the first property (the set of all inhabitants of `X` belongs to `F`) is equivalent to the property that `F` is not the empty collection of sets. This shouldn't be confused with the more subtle question as to whether the empty set is an *element* of `F`. The definition of a filter does not prohibit `∅ ∈ F`, but if the empty set is in `F` then every set is in `F`, which is to say, `∀ U : set X, U ∈ F`. In this case, `F` is a rather trivial filter, which is precisely the bottom element of the complete lattice `filter X`. This contrasts with the definition of filters in Bourbaki, which doesn't allow filters containing the empty set.

Because we include the trivial filter in our definition, we sometimes need to explicitly assume nontriviality in some lemmas. In return, however, the theory has nicer global properties. We have already seen that including the trivial filter gives us a bottom element. It also allows us to define `principal : set X → filter X`, which maps `∅` to `⊥`, without additing a precondition to rule out the empty set. And it allows us to define the pullback operation without a precondition as well. Indeed, it can happen that `comap f F = ⊥` although `F ≠ ⊥`. For instance, given `x₀ : ℝ` and `s : set ℝ`, the pullback of `𝓝 x₀` under the coercion from the subtype corresponding to `s` is nontrivial if and only if `x₀` belongs to the closure of `s`.

In order to manage lemmas that do need to assume some filter is nontrivial, mathlib has a type class `filter.ne_bot`, and the library has lemmas that assume `(F : filter X) [F.ne_bot]`. The instance database knows, for example, that `(at_top : filter ℕ).ne_bot`, and it knows that pushing forward a nontrivial filter gives a nontrivial filter. As a result, a lemma assuming `[F.ne_bot]` will automatically apply to `map u at_top` for any sequence `u`.

Our tour of the algebraic properties of filters and their relation to limits is essentially done, but we have not yet justified our claim to have recaptured the usual limit notions. Superficially, it may seem that `tendsto u at_top (𝓝 x₀)` is stronger than the notion of convergence defined in Section 3.6 because we ask that *every* neighborhood of `x₀` has a preimage belonging to `at_top`, whereas the usual definition only requires this for the standard neighborhoods `Ioo (x₀ - ε) (x₀ + ε)`. The key is that, by definition, every neighborhood contains such a standard one. This observation leads to the notion of a *filter basis*.

Given `F : filter X`, a family of sets *`s : ι → set X`* is a basis for `F` if for every set `U`, we have `s i ⊆ U` for some `i`. In other words, formally speaking, `s` is a basis if it satisfies `∀ U : set X, U ∈ F ↔ ∃ i, s i ⊆ U`. It is even more flexible to consider a predicate on `ι` that selects only some of the values `i` in the indexing type. In the case of `𝓝 x₀`, we want `ι` to be `ℝ`, we write `ε` for `i`, and the predicate should select the positive values of `ε`. So the fact that the sets `Ioo (x₀ - ε) (x₀ + ε)` form a basis for the neighborhood topology on `ℝ` is stated as follows:

```
example (x₀ : ℝ) : has_basis (𝓝 x₀) (λ ε : ℝ, 0 < ε) (λ ε, Ioo (x₀ - ε) (x₀ + ε)) :=
nhds_basis_Ioo_pos x₀
```

There is also a nice basis for the filter `at_top`. The lemma `filter.has_basis.tendsto_iff` allows us to reformulate a statement of the form `tendsto f F G` given bases for `F` and `G`. Putting these pieces together gives us essentially the notion of convergence that we used in Section 3.6.

```
example (u : ℕ → ℝ) (x₀ : ℝ) :
  tendsto u at_top (𝓝 x₀) ↔ ∀ ε > 0, ∃ N, ∀ n ≥ N, u n ∈ Ioo (x₀ - ε) (x₀ + ε) :=
begin
  have : at_top.has_basis (λ n : ℕ, true) Ici := at_top_basis,
  rw this.tendsto_iff (nhds_basis_Ioo_pos x₀),
  simp
end
```

We now show how filters facilitate working with properties that hold for sufficiently large numbers or for points that are sufficiently close to a given point. In Section 3.6, we were often faced with the situation where we knew that some property `P n` holds for sufficiently large `n` and that some other property `Q n` holds for sufficiently large `n`. Using `cases` twice

gave us N_P and N_Q satisfying ∀ n ≥ N_P, P n and ∀ n ≥ N_Q, Q n. Using set N := max N_P N_Q, we could eventually prove ∀ n ≥ N, P n ∧ Q n. Doing this repeatedly becomes tiresome.

We can do better by noting that the statement "P n and Q n hold for large enough n" means that we have {n | P n} ∈ at_top and {n | Q n} ∈ at_top. The fact that at_top is a filter implies that the intersection of two elements of at_top is again in at_top, so we have {n | P n ∧ Q n} ∈ at_top. Writing {n | P n} ∈ at_top is unpleasant, but we can use the more suggestive notation ∀ᶠ n in at_top, P n. Here the superscripted f stands for "filter." You can think of the notation as saying that for all n in the "set of very large numbers," P n holds. The ∀ᶠ'' notation stands for ``filter.eventually, and the lemma filter.eventually.and uses the intersection property of filters to do what we just described:

```
example (P Q : ℕ → Prop) (hP : ∀ᶠ n in at_top, P n) (hQ : ∀ᶠ n in at_top, Q n) :
  ∀ᶠ n in at_top, P n ∧ Q n := hP.and hQ
```

This notation is so convenient and intuitive that we also have specializations when P is an equality or inequality statement. For example, let u and v be two sequences of real numbers, and let us show that if u n and v n coincide for sufficiently large n then u tends to x_0 if and only if v tends to x_0. First we'll use the generic eventually and then the one specialized for the equality predicate, eventually_eq. The two statements are definitionaly equivalent so the same proof work in both cases.

```
example (u v : ℕ → ℝ) (h : ∀ᶠ n in at_top, u n = v n) (x₀ : ℝ) :
  tendsto u at_top (𝓝 x₀) ↔ tendsto v at_top (𝓝 x₀) :=
tendsto_congr' h

example (u v : ℕ → ℝ) (h : u =ᶠ[at_top] v) (x₀ : ℝ) :
  tendsto u at_top (𝓝 x₀) ↔ tendsto v at_top (𝓝 x₀) :=
tendsto_congr' h
```

It instructive to review the definition of filters in terms of eventually. Given F : filter X, for any predicates P and Q on X,

- the condition univ ∈ F ensures (∀ x, P x) → ∀ᶠ x in F, P x,
- the condition U ∈ F → U ⊆ V → V ∈ F ensures (∀ᶠ x in F, P x) → (∀ x, P x → Q x) → ∀ᶠ x in F, Q x, and
- the condition U ∈ F → V ∈ F → U ∩ V ∈ F ensures (∀ᶠ x in F, P x) → (∀ᶠ x in F, Q x) → ∀ᶠ x in F, P x ∧ Q x.

```
#check @eventually_of_forall
#check @eventually.mono
#check @eventually.and
```

The second item, corresponding to eventually.mono, supports nice ways of using filters, especially when combined with eventually.and. The filter_upwards tactic allows us to combine them. Compare:

```
example (P Q R : ℕ → Prop) (hP : ∀ᶠ n in at_top, P n) (hQ : ∀ᶠ n in at_top, Q n)
  (hR : ∀ᶠ n in at_top, P n ∧ Q n → R n) :
  ∀ᶠ n in at_top, R n :=
begin
  apply (hP.and (hQ.and hR)).mono,
  rintros n ⟨h, h', h''⟩,
  exact h'' ⟨h, h'⟩
end

example (P Q R : ℕ → Prop) (hP : ∀ᶠ n in at_top, P n) (hQ : ∀ᶠ n in at_top, Q n)
  (hR : ∀ᶠ n in at_top, P n ∧ Q n → R n) :
  ∀ᶠ n in at_top, R n :=
```

(continues on next page)

(continued from previous page)

```
begin
  filter_upwards [hP, hQ, hR],
  intros n h h' h'',
  exact h'' ⟨h, h'⟩
end
```

Readers who know about measure theory will note that the filter `μ.ae` of sets whose complement has measure zero (aka "the set consisting of almost every point") is not very useful as the source or target of `tendsto`, but it can be conveniently used with `eventually` to say that a property holds for almost every point.

There is a dual version of `∀ᶠ x in F, P x`, which is occasionally useful: `∃ᶠ x in F, P x` means `{x | ¬P x} ∉ f`. For example, `∃ᶠ n in at_top, P n` means there are arbitrarily large `n` such that `P n` holds. The `∃ᶠ` notation stands for `filter.frequently`.

For a more sophisticated example, consider the following statement about a sequence `u`, a set `M`, and a value `x`:

> If `u` converges to `x` and `u n` belongs to `M` for sufficiently large `n` then `x` is in the closure of `M`.

This can be formalized as follows:

```
tendsto u at_top (𝓝 x) → (∀ᶠ n in at_top, u n ∈ M) → x ∈ closure M.
```

This is a special case of the theorem `mem_closure_of_tendsto` from the topology library. See if you can prove it using the quoted lemmas, using the fact that `cluster_pt x F` means `(𝓝 x ⊓ F).ne_bot`.

```
#check mem_closure_iff_cluster_pt
#check le_principal_iff
#check ne_bot_of_le

example (u : ℕ → ℝ) (M : set ℝ) (x : ℝ)
  (hux : tendsto u at_top (𝓝 x)) (huM : ∀ᶠ n in at_top, u n ∈ M) : x ∈ closure M :=
sorry
```

CHAPTER

EIGHT

INDEX

INDEX