Mathematics in Lean

Release 0.1

Jeremy Avigad
Kevin Buzzard
Robert Y. Lewis
Patrick Massot

May 29, 2020

CONTENTS

INTRODUCTION

1.1 Getting Started

So, you are ready to formalize some mathematics. Maybe you have heard that formalization is the future (say, from the article The Mechanization of Mathematics, or the talk The Future of Mathematics), and you want in. Maybe you have played the Natural Number Game and you are hooked. Maybe you have heard about Lean and its library mathlib through online chatter and you want to know what the fuss is about. Or maybe you like mathematics and you like computers, and you have some time to spare. If you are in any of these situations, this book is for you.

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 in the mathlib repository.

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 the file `welcome.lean` will simultaneously open this tutorial in a VS Code window.

Every once in a while, you will see a code snippet like this:

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

Clicking on the `try it!` button in the upper right corner will open a copy in a window so that you can edit it, and Lean provides feedback in the `Lean Goal` window. This book provides lots of challenging exercises for you to do that way.

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*. 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 were 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`:

```
import data.nat.parity
open nat

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

```
import data.nat.parity tactic
open nat

example : ∀ m n, even n → even (m * n) :=
begin
  rintros m n ⟨k, hk⟩,
  use m * k,
```

```
  rw [hk, mul_left_comm]
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 m = 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 last line uses the rewrite tactic to replace n by 2 * k in the goal and then swap the m and the 2 to show that the two sides of the equality are the same.

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, even n → even (m * n) :=
by rintros m n ⟨k, hk⟩; use m * k; rw [hk, mul_left_comm]
```

Here were 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, 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 mathlib 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.

Acknowledgment. We are grateful to Gabriel Ebner for setting up the infrastructure for running this tutorial in VS Code.

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`. 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`.

```
import data.real.basic

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

As you move your cursor past each step of the proof, you can see the goal of the proof change. 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 all the example as it is meant to be processed and checked by Lean.

You can type the ℝ character as `\R` or `\real` in the VS Code editor. 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 a backslash, you can change the leading character by changing the `lean.input.leader` setting in VS Code.

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
```

In the Lean editor mode, when a cursor is in the middle of a tactic proof, Lean reports on the current *proof state*. A typical proof state in Lean might look as follows:

```
1 goal
x y : ℕ,
h₁ : prime x,
h₂ : ¬even x,
h₃ : y > x
⊢ y ≥ 4
```

The lines before the one that begins with $\vdash$ 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 h_1, h_2, and h_3. 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.

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,
  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) :=
begin
  rw [h', ←mul_assoc, h, mul_assoc]
end

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 forall 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 : ℝ

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

We can delimit the scope of the declaration by putting it in a `section ... end` block. Finally, 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 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 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 `a + a = 2 * 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
  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 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`.

2.2 Proving Identities in Algebraic Structures

Mathematically, a ring consists of a set, 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, we base our algebraic structures on *types* rather than sets. Modulo this difference, we can take the ring axioms to be as follows:

```
import algebra.ring

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 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`.

```
import tactic

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 :=
```

(continues on next page)

```
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. To avoid error messages from Lean, we will put our versions 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 algebra.ring

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]

end my_ring

#check @my_ring.add_zero
#check @add_zero
```

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 neg_add_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
```

If you are clever, 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. We will discuss apply and exact in the next section.

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.

In Lean, subtraction in a ring is defined to be addition of the additive inverse.

```
theorem sub_eq_add_neg (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 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
```

Extra points if you do it two different ways: once using `rw`, and once 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 cleverness, 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:

```
import algebra.group

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`).

```
import algebra.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.

```
import algebra.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)

namespace my_group

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

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

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

end my_group
```

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`:

```
import data.real.basic

variables a b c : ℝ

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

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 (a b c d e : ℝ) (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 (a b c d e : ℝ) (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.

```
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.

```
import analysis.special_functions.exp_log

open real

variables  a b c d : ℝ

#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_lt_add_of_le_of_lt : a ≤ b → c < d → a + c < b + d)
#check (add_lt_add_of_le_of_lt : a ≤ b → c < d → a + c < b + d)
#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)
```

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

In this section, however, we will use that 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) :=
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 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.

```
import data.real.basic
import tactic

example (a : ℝ) : 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. 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 :=
begin
  sorry
end
```

Also, confirm that `linarith` can do it with a bit of help.

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 in this example. 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 `≤` over `≥`. 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_of_le_of_neg_le`.

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

#check abs_le_of_le_of_neg_le
```

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?

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,
      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 a later chapter, 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 :=
begin
  sorry
end

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 ≤ and the characterizing properties of `min` and `max` enumerated above. We need to use the fact that ≤ on the real numbers is a *total order*, which is to say, it satisfies $\forall$ `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 a later chapter, so 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) :=
begin
  sorry
end

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

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) :=
begin
  sorry
end
```

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.

```
import data.nat.gcd

variables x y z : ℕ

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 :=
begin
  rw nat.pow_two,
  apply dvd_mul_left
end
```

You can also use `nat.pow_succ` instead of `nat.pow_two` to expand `x^2` into a product, with slightly different effect. (In the context of the natural numbers, `succ` refers to the successor function; in Lean, `2` is definitionally equal to `succ 1`.) See if you can guess the names of the theorems you need to prove the following:

```
import data.nat.gcd

variables w x y z : ℕ

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

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:

```
import data.nat.gcd

open nat

variables n : ℕ

#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 :=
begin
  sorry
end
```

2.5 Proving Facts about Algebraic Structures

In Section 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 likes 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, ≤, 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 ∧ stands for "and," the symbol ¬ stands for "not," and x ≠ y abbreviates ¬ (x = y). In a later chapter, 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 ⊓ and ⊔ that are analogous to min and max on the real numbers:

```
import order.lattice

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 mind:

- ⊓ is the *greatest lower bound*, *infimum*, or *meet*.

- ⊔ is the *least upper bound*, *supremum*, or *join*.

Some instances of lattices include:

- min 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 min / 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:

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

```
import order.lattice

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:

```
import order.lattice

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

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

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

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

```
import algebra.ordered_ring

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)
#check (zero_ne_one : (0 : R) ≠ 1)
```

In a later chapter, we will see how 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:

```
import topology.metric_space.basic

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.

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 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_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 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_lemma : ∀ {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, 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_lemma : ∀ {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, remember that `λ x, f x + g x` is the function that maps x to `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 predicate `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.

```
import algebra.ordered_group

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. The `intros` command corresponds to a lambda, and the remaining term consists of applications.

```
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 even (f : ℝ → ℝ) : Prop := ∀ x, f x = f (-x)
def odd (f : ℝ → ℝ) : Prop := ∀ x, f x = - f (-x)

example (ef : even f) (eg : even g) : even (λ x, f x + g x) :=
begin
  intro x,
  calc
    (λ x, f x + g x) x = f x + g x        : rfl
                   ... = f (-x) + g (-x) : by rw [ef, eg]
end
```

(continues on next page)

```
example (of : odd f) (og : odd g) : even (λ x, f x * g x) :=
sorry

example (ef : even f) (og : odd g) : odd (λ x, f x * g x) :=
sorry

example (ef : even f) (og : odd g) : 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.

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 $\in$ 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 $\subseteq$ t is defined to mean $\forall$ {x : α}, x $\in$ s $\to$ x $\in$ t. The variable in the quantifier is marked implicit so that given h : s $\subseteq$ t and h' : x $\in$ s, we can write h h' as justification for x $\in$ 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 }

example : s ⊆ s := λ x xs, xs

example : r ⊆ s → s ⊆ t → r ⊆ t :=
begin
  sorry
end

example : 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 $\leq$ 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 x_1 and x_2 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
  intros x₁ x₂ h',
  apply eq_of_add_eq_add_right 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)) :=
begin
  intros x₁ x₂ h,
  apply injf,
  apply injg,
  apply h
end
```

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 $\exists$ 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, ∧, below.) 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.

```
import data.real.basic

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 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` if 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`.

```
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⟩,
  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.

```
import tactic

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 a 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.

```
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
```

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.

```
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 ≤. 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:

```
variables {α : Type*} (P : α → 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)

```
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. (Due to an annoying quirk of Lean, this cannot appear right after the `import` lines in your file, but it can appear later on.) With that command, we can use proof by contradiction to prove the third implication as follows.

```
import tactic

variables {α : Type*} (P : α → Prop)

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 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 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, $\wedge$, 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 $\leq$. 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',
  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 ∧ prime m ∧ 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.

```
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 ≤, Lean axiomatizes the associated strict pre-order by `a < b ↔ a ≤ b ∧ ¬ b ≤ a`. Show that if ≤ 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 ≤ 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 ≤ and ¬ three times.

```
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`.

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

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 "insert left" and `inr` is short for "insert 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.

```
theorem le_abs_self : x ≤ abs x :=
sorry

theorem neg_le_abs_self : -x ≤ abs x :=
sorry

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

example (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:

```
import algebra.group_power tactic

variables {R : Type*} [integral_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 classi-`
`cal`. 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.

```
import tactic

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 $s : \mathbb{N} \to \mathbb{R}$. 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 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 }
```

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 : 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.

```
variables {s t : ℕ → ℝ} {a b : ℝ}

theorem converges_to_add
  (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
    {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_of_ne_zero 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 (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 (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
    (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 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.