Mathematics in Lean

Release 0.1

Jeremy Avigad
Kevin Buzzard
Robert Y. Lewis
Patrick Massot

May 12, 2020

CONTENTS

CHAPTER

ONE

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. Or 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, or 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 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,
```

(continues on next page)

(continued from previous page)

```
  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.

CHAPTER

TWO

BASIC SKILLS

This chapter is designed to introduce you to the nuts and bolts of mathematical reasoning in Lean: calculating, applying lemmas and theorems, and carrying out proof by induction.

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 ⊢ 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,
```

(continues on next page)

(continued from previous page)

```
  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
```

(continues on next page)

(continued from previous page)

```
#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 exactly.

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:

```
variables (R : Type*) [comm_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 :=
begin
  rw [hyp, hyp'],
```

(continues on next page)

(continued from previous page)

```
  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.

```
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:

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

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

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

```
variables (G : Type*) [group G]

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

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

```
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)
```

(continues on next page)

(continued from previous page)

```
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 :=
```

(continues on next page)

(continued from previous page)

```
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
```

(continues on next page)

(continued from previous page)

```
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,
```

(continues on next page)

(continued from previous page)

```
  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,
```

(continues on next page)

(continued from previous page)

```
    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 ∀ 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.

For another challenge, mathlib's manic naming convention is on proud display in the library's name for the triangle inequality:

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

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

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

In this example, the symbol $\wedge$ stands for "and," the symbol $\neg$ stands for "not," and `x ≠ y` abbreviates `¬ (x = y)`. In 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 $\sqcap$ and $\sqcup$ 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 $\sqcap$ and $\sqcup$ 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:

- $\sqcap$ is the *greatest lower bound*, *infimum*, or *meet*.
- $\sqcup$ 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 $\leq$
- $\cap$ and $\cup$ on the collection of subsets of some domain, with the ordering $\subseteq$
- $\wedge$ and $\vee$ 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:

```
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 (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.

CHAPTER

THREE

THE NATURAL NUMBERS

Mathematically speaking, the natural numbers are characterized up to isomorphism as a set with a zero element and an injective successor function such that zero is not a successor and the entire structure satisfies the *induction principle*: any property that holds of zero and is maintained by successors holds of every natural number. To introduce the natural numbers formally, Lean's core library makes use of Lean's foundational framework, which allows us to declare a new type as follows:

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

This declares a type `nat` with constants `nat.zero : nat` and `nat.succ : nat → nat`, and provides the associated principles of induction and recursion.

In this chapter, we will explain how the Lean library establishes the familiar properties of arithmetic on this foundation, and then we will show you how to use the natural numbers to prove some interesting theorems.

3.1 Defining Arithmetic

The principle of recursive definition means that, once we introduce notation ℕ for `nat` and `0` for `nat.zero`, and once we open the `nat` namespace so that we can write `succ` instead of `nat.succ`, we can define addition recursively as follows:

```
def add : ℕ → ℕ → ℕ
| m 0 := m
| m (succ n) := succ (add m n)
```

Once we introduce the usual notation for addition, the definition means that we can use the associated defining equations:

```
#check (add_zero m   : m + 0 = m)
#check (add_succ m n : m + (succ n) = succ (m + n))
```

But the computational nature of Lean's foundation gives us more, namely, that Lean will carry out these reductions internally in order to make expressions match, so the equations hold by the reflexivity axiom:

```
example : m + 0 = m := rfl
example : m + (succ n) = succ (m + n) := rfl
```

Here, `rfl` is a proof term that corresponds to the `refl` tactic. Numerals are also defined in a clever way in Lean so that, for example, `1` reduces to `succ 0`. This means in many contexts we can use `n + 1` and `succ n` interchangeably:

```
example : succ n = n + 1 := rfl
```

Suppose we want to prove the commutativity of addition, `m + n = n + m`. We don't have much to work with: we have the defining equations for addition, but no other facts about it. But we do have the principle of induction, which we can invoke with the `induction` tactic:

```
theorem add_comm : m + n = n + m :=
begin
  induction n,
  { sorry },
  sorry
end
```

In this section, we will continue the strategy of stating theorems with the same names that are used in the library but hiding them in a namespace to avoid a naming conflict. If you move your cursor through the proof, you will see that the induction tactic leaves two goals: in the base case, we need to prove `m + 0 = 0 + m`, and in the induction step, we need to prove `m + succ n = succ n + m` using the inductive hypothesis `m + n = n + m`. You will also see that Lean chose names automatically for the inductive hypothesis and the variable in the induction step. We can tell Lean to use `n` for the variable name and `ih` for the name of the inductive hypothesis by appending `with n ih` to the induction command.

How can we prove the base case? It turns out that this requires another instance of induction. We could call the induction tactic again in that subproof, but since the fact that we need, `0 + m = m`, is independently useful, we may as well make it a separate theorem. Similarly, in the inductive hypothesis, we need `succ m + n = succ (m + n)`, so we break that out as a separate theorem as well.

```
theorem zero_add : 0 + m = m :=
begin
  induction m with m ih,
  { refl },
  rw [add_succ, ih]
end

theorem succ_add : succ m + n = succ (m + n) :=
begin
  induction n with n ih,
  { refl },
  rw [add_succ, ih]
end

theorem add_comm : m + n = n + m :=
begin
  induction n with n ih,
  { rw zero_add, refl },
  rw [succ_add, ←ih]
end
```

We can similarly make quick work of associativity:

```
theorem add_assoc : m + n + k = m + (n + k) :=
begin
  induction k with k ih,
  { refl },
  rw [add_succ, ih],
  refl
end
```

Because addition is defined by recursion on the second argument, doing induction on `k` will allow us to use the defining equations for addition in the base case and induction step. This is a good heuristic when it comes to deciding which variable to use. We can do on to define multiplication in the expected way:

```
def mul : ℕ → ℕ → ℕ
| m 0     := 0
| m (n+1) := mul m n + m
```

This gives us the defining equations for multiplication:

```
#check (mul_zero m   : m * 0 = 0)
#check (mul_succ m n : m * (succ n) = m * n + m)

example : m * 0 = 0 := rfl
example : m * (n + 1) = m * n + m := rfl
```

We now challenge you to use nothing more than these defining equations and the properties of addition we have already established to prove all of the following:

```
theorem mul_add : m * (n + k) = m * n + m * k := sorry

theorem zero_mul : 0 * n = 0 := sorry

theorem one_mul : 1 * n = n := sorry

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

theorem mul_comm : m * n= n * m := sorry
```

3.2 Using the Natural Numbers

The command `#eval` cannot be used to prove theorems: Lean extracts bytecode from the definitions and executes them efficiently, without justifying the computation axiomatically. But it can be enjoyable to define a function, prove things about it, and then see it run. It is also sometimes helpful to get a sense of what the function does.

But how can we prove `2 + 2 = 4`?

3.3 Sums and Products

3.4 Fibonacci Numbers

[Watch this space.]

3.5 The AM-GM Inequality

[Watch this space.]