Mathematics in Lean

Release 0.1

Jeremy Avigad
Kevin Buzzard
Robert Y. Lewis
Patrick Massot

May 05, 2020

CONTENTS

1 Introduction 1

1.1 Getting Started 1

1.2 Overview 2

2 Basic Skills 5

2.1 Calculating 5

2.2 Proving Identities in Algebraic Structures 10

2.3 Using Theorems and Lemmas 14

2.4 Proving Facts about Algebraic Structures 14

2.5 Induction 14

2.6 The AM-QM Inequality 15

3 Logic 17

3.1 Implication and the Universal Quantifier 18

3.2 Conjunction and Negation 20

3.3 Disjunction 23

3.4 The Existential Quantifier 23

3.5 Logical Equivalence 24

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, are you have some time to spare. If you are in any of these situations, this book is for you.

[The rest of this section is science fiction right now]

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. Open that directory in `VS Code`.
4. Type `Ctrl-shift-P` and then enter the command `Lean: Open Documentation View`.
5. Click on `Mathematics in Lean`.

At that point, you will be reading 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 make a local copy of the snippet in your working folder and open it in a window so that you can experiment with it. This book also provides lots of exercises in that format. You can save your changes from VS Code in the usual way, and come back to the same file by pressing the corresponding `try it!` button again. If you want to reset the snippet or exercise to the version in the book, simply delete or rename the file with the changes you have made, and then press `try it!` once again.

Sometimes in the text we will quote from a longer example, like so:

```
-- Give an example here
-- Instead of a ``try it!'' button,
-- there should be a ``see more!`` button.
```

In that case, clicking on the `see more!` button opens a longer Lean file and takes you to that line. These displays are read only, and you should think of them as part of the main text. This allows us to describe a long development one piece at a time, leaving you free to survey the whole development as you please.

Of course, you can create other Lean files to experiment. We have therefore set up the main folder with four subdirectories:

- *snippets* contains your edited copies of the snippets in the text.
- *exercises* contains your edited copies of the exercises.
- *examples* contains the read-only examples we make use of in the text.
- *user* is a folder for you use any way you please.

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 ℕ or ℕ → 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
```

(continues on next page)

(continued from previous page)

```
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 prover will challenge you to think about mathematics and mathematical reasoning in fundamentally new ways. Your life may never be the same.

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 me!` button displays all the example as it is meant to be processed and checked by Lean.

Incidentally, 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 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 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 means of checking and expression and determining its type:

```
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` command 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 (a b c d : ℝ) (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, with 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 for 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 = 0 :=
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 the 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's 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: Here the curly brackets could be omitted, 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 a group the 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 one_mul (a : G) : 1 * a = a :=
sorry

theorem one_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

This section will introduce `apply` and friends, and prove some theorems that go beyond equations.

Examples will use $\leq$, `dvd`, `gcd` and `lcm` on `nat`, $\subseteq$ on sets, and so on.

Basically, we'll theorems with $\forall$ and $\to$ to prove atomic statements. Even with these restrictions, we can prove some nontrivial theorems, like the fact that `gcd` distributes over `lcm`, or `a * b ≤ (a^2 + b^2) / 2`.

We'll also demonstrate the use of calc for inequalities.

2.4 Proving Facts about Algebraic Structures

As we did in section Section 2.2, this section will apply skills from the last section in particular algebraic structures.

Examples may include:

- lattices: lubs, glbs, then theorems about distributivity, etc.
- ordered rings
- simple facts about metric spaces from axioms
- derive different versions of the triangle inequality from axioms for norms

2.5 Induction

We'll run through addition, multiplication, and exponentiation, à la the natural numbers game.

Then we'll do fibonacci numbers or something like that. By now, students should be able to prove interesting identities.

2.6 The AM-QM Inequality

We'll try taking the students through a proof of this, or something else that puts everything together: induction, inequalities, calculations, and theorems from the library.

CHAPTER THREE

LOGIC

Logic provides the means by which complex mathematical statements are built up from more simple ones, using linguistic constructs such as "and," "or," "not," and "if … then," "every," and "some." This chapter explains the rules that govern the use of these expressions in Lean.

The syntax for the logical connectives is as follows:

```
variables A B : Prop
variable  α : Type*
variable  P : α → Prop

-- if A then B
#check A → B

-- A and B
#check A ∧ B

-- A or B
#check A ∨ B

-- not A
#check ¬ A

-- A if and only if B
#check A ↔ B

#check true
#check false

-- for every x, P x
#check ∀ x, P x

-- for some x, P x
#check ∃ x, P x
```

In VS Code, you can type the symbols →, ∧, ∨, ¬, ↔, ∀, ∃ as `\r`, `\and`, `\or`, `\not`, `\iff`, `\all`, and `\ex` respectively.

The first line, `variables A B : Prop`, declares some variables ranging over propositions. In the next two lines, α is declared to be an arbitrary type, and `P` is declared as a general predicate on this type: for any `x` of type α, `P x` is the statement that `P` holds of type α. For example, the property of natural numbers of being `even` is represented as an object of type `ℕ → Prop`. The binary relation `m ∣ n` of divisibility on the natural numbers is represented as an object of type `ℕ → ℕ → Prop`. Since we can declare variables ranging over types like these, Lean allows us to make general schematic statements about propositions, predicates, and relations. In this chapter, we will use such schematic statements to describe logical rules in full generality, but then illustrate their uses in particular mathematical settings.

[Note: everything in this chapter will be rewritten. The idea is to introduce the rules for the logical connectives, but in the context of interesting mathematical examples and exercises. Some ideas are listed at the beginning of each section.]

3.1 Implication and the Universal Quantifier

[Examples: use monotone functions, and properies of lubs and glbs. For example, show that in any complete lattice, or maybe just on the reals, the glb can be defined as the lub of all the lower bounds.]

To prove an implication, we introduce it with a label. This moves it to the context, where we can use it. To *use* an element in the context, whether it is an implication or an atomic fact, we can apply it to the goal.

```
example : A → A :=
begin
  intro h,
  apply h
end
```

When a tactic proof is that short, one can put it on one line.

```
example : A → A :=
by { intro h, apply h }

example : A → A :=
by intro h; apply h
```

The `by` command uses a single tactic to prove a goal. The curly brackets are notation for a `begin ... end` pair, which condense a sequence of tactic instructions to a single one. We will see later that the semantics of the semicolon is slightly different; `t1; t2` also combines two tactics into one, but if applying `t1` results in more than one goal, `t2` is applied to all of them. If you put the cursor after the comma in `{ intro h, apply h }`, Lean will still show you the proof state at that point. A disadvantage of using the semicolon is that in this case there is no intermediate state; `intro h; apply h` is really a single compound step.

Here is a more interesting example.

```
example : (A → B) → (B → C) → A → C :=
begin
  intros h₁ h₂ h₃,
  apply h₂,
  apply h₁,
  apply h₃
end
```

This illustrates an important feature of the implication notation, namely, that iterated implication associates *to the right*. This means that the example above is parsed as `(A → B) → ((B → C) → (A → C))`. This convention supports the fact that it is common to state a theorem as an implication from hypotheses to a conclusion. Thus the example above can be read as the theorem that states that `C` follows from hypotheses `(A → B)`, `(B → C)`, and `A`. Of course, to prove such a theorem, the first thing you will do is introduce the hypotheses and name them. Because this pattern is so common, Lean also offers syntax to state a goal with the hypotheses already introduced:

```
example (h₁ : A → B) (h₂ : B → C) : A → C :=
begin
  intro h₃,
  apply h₂,
  apply h₁,
  apply h₃
end
```

Because introduction and application are so fundamental, it is often useful to replace tactic steps by the corresponding proof term. The notation for introduction is *lambda abstraction*: if `A` is any `Prop` and `t` is a proof of `B` in a context that includes `h : A`, then `λ h : A, t` is a proof of `A → B`. The label `h : A` can be simplified to `h` when Lean can infer `A` from the current context and goal. The notation for application of an implication to a hypothesis is simply to write one term next to the other: given `h₁ : A → B` and `h₂ : A`, the expression `h₁ h₂` denotes a proof of `B`. Thus all of the following work:

```
example : A → A :=
λ h : A, h

example : A → A :=
λ h, h

example (h₁ : A → B) (h₂ : B → C) : A → C :=
begin
  intro h₃,
  apply h₂ (h₁ h₃)
end

example (h₁ : A → B) (h₂ : B → C) : A → C :=
begin
  intro h₃,
  exact h₂ (h₁ h₃)
end

example (h₁ : A → B) (h₂ : B → C) : A → C :=
λ h₃, h₂ (h₁ h₃)
```

The `exact` tactic is like the `apply` tactic, except that it is expected to solve the current goal exactly, rather than reduce it to other subgoals, which can happen when one applies an implication. Using `exact` provides structure to a tactic proof, since it signals to Lean and to the reader that the command finishes off the current goal.

Lean provides additional mechanisms to structure a tactic proof. The `have` tactic introduces an intermediate subgoal: if you type `have h : A` in a context in which the target is `B`, you are left with two subgoals: first, you are required to prove `A` in the current context, and then you are required to prove `B` in a context that includes `A`.

```
example (h₁ : A → B) (h₂ : B → C) : A → C :=
begin
  intro h₃,
  have h₄ : B,
  { apply h₁, apply h₃ },
  show C,
  apply h₂, apply h₄
end
```

In this example, the `show` command does nothing substantial. It only serves to confirm to Lean, and to the reader of the proof, that at that stage the goal is to prove `C`. (Later we will see that `show` is syntactic sugar for the `change` tactic, and can often be used to re-express the target in an equivalent form.)

3.2 Conjunction and Negation

[This section will provide mathematical examples that require conjunction and negation.]

[Here is one: we can prove that if ≤ is a partial order and `a < b` is defined to be `a ≤ b ∧ a ≠ b`, then `a < b` is a strict order. Moreover, if ≤ is total, so is <. This proofs are just a lot of messing around with ∧ and ¬, so they are good exercises.]

[If you can think of other good examples, please let me know.]

Let's move on to "and," otherwise known as *conjunction*. Given a target of `A ∧ B,` the `split` tactic reduces the current goal to the two goals of proving `A` and `B`, respectively, each in the same context. On the other hand, given `h : A ∧ B` as a *hypothesis*, the expressions `h.1` and `h.2` provide proofs of `A` and `B`, respectively.

```
example : A ∧ B → B ∧ A :=
begin
  intro h,
  split,
  apply h.2,
  apply h.1
end
```

The notations `h.1` and `h.2` are instances of Lean's general projection notation. As we will see, it can be used in lots of situations where an object or hypothesis represent and amalgamation.

Instead of using the `split` tactic, we can use Lean's *anonymous constructor notation* `⟨..., ..., ...⟩` to tell Lean to put together the object we want. You can type the corner brackets with `\<` and `\>`.

```
example : A ∧ B → B ∧ A :=
begin
  intro h,
  exact ⟨h.2, h.1⟩
end
```

Just as anonymous constructors provide a general swiss-army-knife for putting together proofs and data, the `cases` tactic provides a general methods of *decomposing* proofs and data. In the next example, it decomposes `h : A ∧ B` into the two hypotheses `h₁: A` and `h₂ : B`.

```
example : A ∧ B → B ∧ A :=
begin
  intro h,
  cases h with h₁ h₂,
  exact ⟨h₂, h₁⟩
end
```

Mathlib provides a tactic, `rintros`, that combines the `intro` and `cases` steps into one. Because it is not a core Lean tactic, we need to add the line `import tactic` to the top of the file. The *pattern* `⟨h₁, h₂⟩` provides names for the hypotheses that are introduced.

```
import tactic

variables A B : Prop

example : A ∧ B → B ∧ A :=
begin
  rintros ⟨h₁, h₂⟩,
  exact ⟨h₂, h₁⟩
end
```

In fact, the use of lambda abstraction in a Lean expression also supports this sort of pattern matching,

```
example : A ∧ B → B ∧ A :=
λ ⟨h₁, h₂⟩, ⟨h₂, h₁⟩
```

Even when writing tactic proofs, it is often useful to use short proof terms like this to finish off a subgoal, for example, using the `exact` tactic.

According to Lean's parsing rules, conjunction associates to the right, so `A ∧ B ∧ C` is the same as `A ∧ (B ∧ C)`. The `rintros` tactic allows for more complex nested patterns to decompose a hypothesis like this. (The "r" stands for "recursive.") Similarly, the `rcases` tactic, like the `cases` tactic, can be used to decompose a hypothesis that is already introduced.

```
example : A ∧ (B ∧ C) ∧ D → (B ∧ D) ∧ A :=
begin
  rintros ⟨h₁, ⟨h₂, _⟩, h₃⟩,
  exact ⟨⟨h₂, h₃⟩, h₁⟩
end

example (h : A ∧ (B ∧ C) ∧ D) : (B ∧ D) ∧ A :=
begin
  rcases h with ⟨h₁, ⟨h₂, _⟩, h₃⟩,
  exact ⟨⟨h₂, h₃⟩, h₁⟩
end
```

This example illustrates another nice bit of Lean syntax: you can use the underscore symbol as an *anonymous label* to avoid naming a hypothesis or piece of data that you do not need to refer to later on. (We will see that the underscore has multiple uses and meanings in Lean.)

We will close this section with a discussion of *negation* and *falsity*. In Lean, `¬ A` is defined to be `A → false`. This makes sense if you think of `¬ A` as equivalent to the statement "if `A` is true, then `2 + 2 = 5`," where `2 + 2 = 5` is a prototypical falsehood. An advantage to this definition is that Lean can unfold the definition when necessary, so that introduction and application work the same way for negation as they do for implication.

```
example : (A → B) → ¬ B → ¬ A :=
begin
  intros h₁ h₂ h₃,
  apply h₂,
  apply h₁,
  apply h₃
end
```

This proof may look familiar: it is exactly the same proof we used to establish `(A → B) → (B → C) → A → C`. We can see that the example above is an instance of the general result by naming the general result and then applying it:

```
theorem impl_compose : (A → B) → (B → C) → A → C :=
λ h₁ h₂ h₃, h₂ (h₁ h₃)

example : (A → B) → ¬ B → ¬ A :=
by apply impl_compose

example : (A → B) → ¬ B → ¬ A :=
impl_compose A B false

example (h₁ : A → B) (h₂ : ¬ B) : ¬ A :=
impl_compose A B false h₁ h₂
```

The fact that the arguments `A`, `B`, and `false` have to be provided in the last two examples give us an opportunity to introduce another important feature of Lean, namely, the ability to declare arguments as *implicit*. In the first example, the `apply` command works because Lean is able to infer the arguments from the target of the goal. For the same reason, we can use an underscore character to leave the arguments implicit in the proof-term representation:

```
example : (A → B) → ¬ B → ¬ A :=
impl_compose _ _ _

example (h₁ : A → B) (h₂ : ¬ B) : ¬ A :=
impl_compose _ _ _ h₁ h₂
```

But typing underscores can be tedious, and so Lean allows us to use curly braces to specify that the arguments will be suppressed by default:

```
theorem impl_compose {A B C : Prop} : (A → B) → (B → C) → A → C :=
λ h₁ h₂ h₃, h₂ (h₁ h₃)

example : (A → B) → ¬ B → ¬ A :=
impl_compose

example (h₁ : A → B) (h₂ : ¬ B) : ¬ A :=
impl_compose h₁ h₂
```

You needn't worry about the details right now. We will have more to say about the use of implicit arguments the next time they come up.

Given that `¬ A` is defined to be `A → false`, what can we say about `false`? One we have `false` in our context, our swiss-army knife, the `cases` tactic, can use it to establish any conclusion. The intuition is that if we try to split on all the ways a contradiction can come about, there aren't any, and so the proof is done. Alternatively, Lean has a `contradiction` tactic, which tries to close a goal by finding any of a number of types of overt contradiction in the context.

```
example : false → A :=
by { intro h, cases h }

example : false → A :=
by { intro h, contradiction }

example (h₁ : B) (h₂ : ¬ B) : A :=
by contradiction
```

```
import tactic

variables A B C : Prop

example : A ∧ (A → B) → A ∧ B :=
sorry

example : B → (A → B) :=
sorry

example (h : A ∧ B → C) : A → B → C :=
sorry

example (h : A → B → C) : A ∧ B → C :=
sorry

example : (A → B) ∧ (B → C) ∧ A → C :=
sorry
```

(continues on next page)

(continued from previous page)

```
example : A → (A → B) → (A ∧ B → C) → C :=
sorry

-- use rcases
example (h : A ∧ (A → B) ∧ (A ∧ B → C)) : C :=
sorry

example : A → ¬ (¬ A ∧ B) :=
sorry

example : ¬ (A ∧ B) → A → ¬ B :=
sorry

example : A ∧ ¬ A → B :=
sorry
```

3.3 Disjunction

[We'll present mathematical examples where case splits are needed, and also reasoning by cases and proof by contradiction.]

[decidability: explain why Lean cares (we can evaluate `if x > 7 then 3 else 9`), but then show how to `open_local_classical`.]

3.4 The Existential Quantifier

[Do some fun examples here, like divisibility and surjectivity.]

A nice example, illustrating the `ring` tactic:

```
import algebra.group_power tactic.ring

variables {α : Type*} [comm_ring α]

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

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

Add exercises for all of these.

3.5 Logical Equivalence

Show how to prove `A ↔ B`, how to use both directions, how to use it with rewrite.