Mathematics in Lean

Release 0.1

Jeremy Avigad
Patrick Massot

May 24, 2025

CONTENTS

CHAPTER

ONE

INTRODUCTION

1.1 Getting Started

The goal of this book is to teach you to formalize mathematics using the Lean 4 interactive proof assistant. It assumes that you know some mathematics, but it does not require much. Although we will cover examples ranging from number theory to measure theory and analysis, we will focus on elementary aspects of those fields, in the hopes that if they are not familiar to you, you can pick them up as you go. We also don't presuppose any background with formal methods. Formalization can be seen as a kind of computer programming: we will write mathematical definitions, theorems, and proofs in a regimented language, like a programming language, that Lean can understand. In return, Lean provides feedback and information, interprets expressions and guarantees that they are well-formed, and ultimately certifies the correctness of our proofs.

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

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

1. Install Lean 4 and VS Code following these installation instructions.
2. Make sure you have git installed.
3. Follow these instructions to fetch the `mathematics_in_lean` repository and open it up in VS Code.
4. Each section in this book has an associated Lean file with examples and exercises. You can find them in the folder `MIL`, organized by chapter. We strongly recommend making a copy of that folder and experimenting and doing the exercises in that copy. This leaves the originals intact, and it also makes it easier to update the repository as it changes (see below). You can call the copy `my_files` or whatever you want and use it to create your own Lean files as well.

At that point, you can open the textbook in a side panel in VS Code as follows:

1. Type `ctrl-shift-P` (`command-shift-P` in macOS).
2. Type `Lean 4: Docs: Show Documentation Resources` in the bar that appears, and then press return. (You can press return to select it as soon as it is highlighted in the menu.)
3. In the window that opens, click on `Mathematics in Lean`.

Alternatively, you can run Lean and VS Code in the cloud, using Gitpod. You can find instructions as to how to do that on the Mathematics in Lean project page on Github. We still recommend working in a copy of the *MIL* folder, as described above.

This textbook and the associated repository are still a work in progress. You can update the repository by typing `git pull` followed by `lake exe cache get` inside the `mathematics_in_lean` folder. (This assumes that you have not changed the contents of the `MIL` folder, which is why we suggested making a copy.)

We intend for you to work on the exercises in the `MIL` folder while reading the textbook, which contains explanations, instructions, and hints. The text will often include examples, like this one:

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

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

This book moreover provides lots of challenging exercises for you to try. Don't rush past these! Lean is about *doing* mathematics interactively, not just reading about it. Working through the exercises is central to the experience. You don't have to do all of them; when you feel comfortable that you have mastered the relevant skills, feel free to move on. You can always compare your solutions to the ones in the `solutions` folder associated with each section.

1.2 Overview

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

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

```
#check 2 + 2

def f (x : ℕ) :=
  x + 3

#check f
```

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

```
#check 2 + 2 = 4

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

#check FermatLastTheorem
```

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 : FermatLastTheorem :=
  sorry

#check hard
```

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

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

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

```
example : ∀ m n : Nat, Even n → Even (m * n) := fun m n ⟨k, (hk : n = k + k)⟩ ↦
  have hmn : m * n = m * k + m * k := by rw [hk, mul_add]
  show ∃ l, m * n = l + l from ⟨_, hmn⟩
```

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

```
example : ∀ m n : Nat, Even n → Even (m * n) :=
fun m n ⟨k, hk⟩ ↦ ⟨m * k, by rw [hk, mul_add]⟩
```

The following is, instead, a *tactic-style* proof of the same theorem, where lines starting with `--` are comments, hence ignored by Lean:

```
example : ∀ m n : Nat, Even n → Even (m * n) := by
  -- Say `m` and `n` are natural numbers, and assume `n = 2 * k`.
  rintro m n ⟨k, hk⟩
  -- We need to prove `m * n` is twice a natural number. Let's show it's twice `m *
↪k`.
  use m * k
  -- Substitute for `n`,
  rw [hk]
  -- and now it's obvious.
  ring
```

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

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

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

```
example : ∀ m n : Nat, Even n → Even (m * n) := by
  rintro m n ⟨k, hk⟩; use m * k; rw [hk]; ring
```

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

```
example : ∀ m n : Nat, Even n → Even (m * n) := by
  intros; simp [*, 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 complete overview of the library; the community web pages contain extensive documentation. Rather, our goal is to introduce you to the style of thinking that underlies that formalization, and point out basic entry points so that you are comfortable browsing the library and finding things on your own.

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

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

Acknowledgments. We are grateful to Gabriel Ebner for setting up the infrastructure for running this tutorial in VS Code, and to Kim Morrison and Mario Carneiro for help porting it from Lean 4. We are also grateful for help and corrections from Takeshi Abe, Julian Berman, Alex Best, Thomas Browning, Bulwi Cha, Hanson Char, Bryan Gin-ge Chen, Steven Clontz, Mauricio Collaris, Johan Commelin, Mark Czubin, Alexandru Duca, Pierpaolo Frasa, Denis Gorbachev, Winston de Greef, Mathieu Guay-Paquet, Marc Huisinga, Benjamin Jones, Julian Külshammer, Victor Liu, Jimmy Lu, Martin C. Martin, Giovanni Mascellani, John McDowell, Joseph McKinsey, Bhavik Mehta, Isaiah Mindich, Kabelo Moiloa, Hunter Monroe, Pietro Monticone, Oliver Nash, Emanuelle Natale, Filippo A. E. Nuccio, Pim Otte, Bartosz Piotrowski, Nicolas Rolland, Keith Rush, Yannick Seurin, Guilherme Silva, Bernardo Subercaseaux, Pedro Sánchez Terraf, Matthew Toohey, Alistair Tucker, Floris van Doorn, Eric Wieser, and others. Our work has been partially supported by the Hoskinson Center for Formal Mathematics.

CHAPTER

TWO

BASICS

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

2.1 Calculating

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

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

Let's try out `rw`.

```
example (a b c : ℝ) : a * b * c = b * (a * c) := by
  rw [mul_comm a b]
  rw [mul_assoc b a c]
```

The `import` lines at the beginning of the associated examples file import the theory of the real numbers from Mathlib, as well as useful automation. For the sake of brevity, we generally suppress information like this in the textbook.

You are welcome to make changes to see what happens. You can type the ℝ character as `\R` or `\real` in VS Code. The symbol doesn't appear until you hit space or the tab key. If you hover over a symbol when reading a Lean file, VS Code will show you the syntax that can be used to enter it. If you are curious to see all available abbreviations, you can hit Ctrl-Shift-P and then type abbreviations to get access to the `Lean 4: Show Unicode Input Abbreviations` command. If your keyboard does not have an easily accessible backslash, you can change the leading character by changing the `lean4.input.leader` setting.

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

```
1 goal
x y : ℕ,
h₁ : Prime x,
h₂ : ¬Even x,
```

(continues on next page)

(continued from previous page)

```
h₃ : y > x
⊢ y ≥ 4
```

The lines before the one that begins with ⊢ denote the *context*: they are the objects and assumptions currently at play. In this example, these include two objects, `x` and `y`, each a natural number. They also include three assumptions, labelled 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.

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. Note that the left-pointing arrow refers to going from right to left in the identity provided by `mul_assoc`, it has nothing to do with the left or right side of the goal.

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

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

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 := by
  rw [mul_assoc]
  rw [mul_comm]
```

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) := by
  sorry

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

You can 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) := by
  rw [h']
  rw [← mul_assoc]
  rw [h]
  rw [mul_assoc]
```

Try these, using the theorem `sub_self` for the second one:

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

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

Multiple rewrite commands can be carried out with a single command, by listing the relevant identities separated by commas inside the square brackets.

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

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

Another trick is that we can declare variables once and for all outside an example or theorem. Lean then includes them automatically.

```
variable (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]
```

Inspection of the tactic state at the beginning of the above proof reveals that Lean indeed included all variables. We can delimit the scope of the declaration by putting it in a `section ... end` block. Finally, recall from the introduction that Lean provides us with a command to determine the type of an expression:

```
section
variable (a b c : ℝ)

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

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 `2 * a = a + a`. The theorems `add_mul` and `mul_add` express the distributivity of multiplication over addition, and the theorem `add_assoc` expresses the associativity of addition. Use the `#check` command to see the precise statements.

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

Whereas it is possible to figure out what is 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)]
```

(continues on next page)

(continued from previous page)

```
    _ = a * a + 2 * (a * b) + b * b := by
      rw [mul_comm b a, ← two_mul]
```

Notice that the proof does *not* begin with `by`: an expression that begins with `calc` is a *proof term*. A `calc` expression can also be used inside a tactic proof, but Lean interprets it as the instruction to use the resulting proof term to solve the goal. The `calc` syntax is finicky: the underscores and justification have to be in the format indicated above. Lean uses indentation to determine things like where a block of tactics or a `calc` block begins and ends; try changing the indentation in the proof above to see what happens.

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) := by
      sorry
    _ = 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 := by
  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 := by
  sorry

#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 := by
  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
```

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

We close this section by noting that Mathlib provides a useful bit of automation with a `ring` tactic, which is designed to prove identities in any commutative ring as long as they follow purely from the ring axioms, without using any local assumption.

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

(continues on next page)

(continued from previous page)

```
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 := by
  rw [hyp, hyp']
  ring
```

The `ring` tactic is imported indirectly when we import `Mathlib.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 Mathlib.Tactic`. We will see there are similar tactics for other common kind of algebraic structures.

There is a variation of `rw` called `nth_rw` that allows you to replace only particular instances of an expression in the goal. Possible matches are enumerated starting with 1, so in the following example, `nth_rw 2 [h]` replaces the second occurrence of `a + b` with `c`.

```
example (a b c : ℕ) (h : a + b = c) : (a + b) * (a + b) = a * c + b * c := by
  nth_rw 2 [h]
  rw [add_mul]
```

2.2 Proving Identities in Algebraic Structures

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

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

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

```
variable (R : Type*) [Ring R]

#check (add_assoc : ∀ a b c : R, a + b + c = a + (b + c))
#check (add_comm : ∀ a b : R, a + b = b + a)
#check (zero_add : ∀ a : R, 0 + a = a)
#check (neg_add_cancel : ∀ 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

recognize appropriate instances. So any theorem about rings can be applied to concrete rings like the integers, $\mathbb{Z}$, the rational numbers, $\mathbb{Q}$, and the complex numbers $\mathbb{C}$. It can also be applied to any instance of an abstract structure that extends rings, such as any ordered ring or any field.

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

```
variable (R : Type*) [CommRing R]
variable (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 := by
  rw [hyp, hyp']
  ring
```

We leave it to you to check that all the other proofs go through unchanged. Notice that when a proof is short, like `by ring` or `by linarith` or `by sorry`, it is common (and permissible) to put it on the same line as the `by`. Good proof-writing style should strike a balance between concision and readability.

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

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

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

```
namespace MyRing
variable {R : Type*} [Ring R]

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

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

#check MyRing.add_zero
#check add_zero

end MyRing
```

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, neg_add_cancel, zero_add]
```

Prove the companion version:

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

Use these to prove the following:

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

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

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

We will 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 := by
  have h : a * 0 + a * 0 = a * 0 + 0 := by
    rw [← mul_add, add_zero, add_zero]
  rw [add_left_cancel h]
```

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. The fact that the next line is indented indicates that Lean is expecting a block of tactics that serves to prove this new goal. The indentation therefore promotes a modular style of proof: the indented subproof 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`. The `exact` tactic takes as argument a proof term which completely proves the current goal, without creating any new goal. The `apply` tactic is a variant whose argument is not necessarily a complete proof. The missing pieces are either inferred automatically by Lean or become new goals to prove. While the `exact` tactic is technically redundant since it is strictly less powerful than `apply`, it makes proof scripts slightly clearer to human readers and easier to maintain when the library evolves.

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 := by
  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 := by
  sorry

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

theorem neg_zero : (-0 : R) = 0 := by
  apply neg_eq_of_add_eq_zero
  rw [add_zero]

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

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

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

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

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

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

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

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 `rfl` tactic does the same. This is an instance of what is known as a *definitional equality* in Lean's underlying logic. This means that not only can one rewrite with `sub_eq_add_neg` to replace `a - b = a + -b`, but in some contexts, when dealing with the real numbers, you can use the two sides of the equation interchangeably. For example, you now have enough information to prove the theorem `self_sub` from the last section:

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

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

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

```
theorem one_add_one_eq_two : 1 + 1 = (2 : R) := by
  norm_num

theorem two_mul (a : R) : 2 * a = a + a := by
  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:

```
variable (A : Type*) [AddGroup A]

#check (add_assoc : ∀ a b c : A, a + b + c = a + (b + c))
#check (zero_add : ∀ a : A, 0 + a = a)
#check (neg_add_cancel : ∀ 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, `AddCommGroup` and `CommGroup`).

```
variable {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 (inv_mul_cancel : ∀ a : G, a⁻¹ * a = 1)
```

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

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

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

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

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

2.3 Using Theorems and Lemmas

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

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

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

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

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

(continues on next page)

(continued from previous page)

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

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

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

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

In the first example, applying `le_trans` creates two goals, and we use the dots to indicate where the proof of each begins. The dots are optional, but they serve to *focus* the goal: within the block introduced by the dot, only one goal is visible, and it must be completed before the end of the block. Here we end the first block by starting a new one with another dot. We could just as well have decreased the indentation. In the third example and in the last example, we avoid going into tactic mode entirely: `le_trans h₀ h₁` and `le_refl x` are the proof terms we need.

Here are a few more library theorems:

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

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

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

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

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

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

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

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

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

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

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

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

```
example (h : a ≤ b) : exp a ≤ exp b := by
  rw [exp_le_exp]
  exact h
```

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

```
example (h₀ : a ≤ b) (h₁ : c < d) : a + exp c + e < b + exp d + e := by
  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
```

The first line, `apply add_lt_add_of_lt_of_le`, creates two goals, and once again we use a dot 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) := by sorry

example : (0 : ℝ) < 1 := by norm_num
```

(continues on next page)

(continued from previous page)

```
example (h : a ≤ b) : log (1 + exp a) ≤ log (1 + exp b) := by
  have h₀ : 0 < 1 + exp a := by sorry
  apply log_le_log h₀
  sorry
```

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 use *Loogle <https://loogle.lean-lang.org>* to search Lean and Mathlib definitions and theorems by patterns.
- You can rely on Mathlib naming conventions and Ctrl-space completion in the editor to guess a theorem name (or Cmd-space on a Mac keyboard). 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 Ctrl-space will give you some helpful choices. Note that hitting Ctrl-space twice displays more information about the available completions.
- If you right-click on an existing theorem name in VS Code, the editor will show a menu with the option to jump to the file where the theorem is defined, and you can find similar theorems nearby.
- You can use the `apply?` tactic, which tries to find the relevant theorem in the library.

```
example : 0 ≤ a ^ 2 := by
  -- apply?
  exact sq_nonneg a
```

To try out `apply?` 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 := by
  sorry
```

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

Here is another example of an inequality:

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

  calc
    2*a*b = 2*a*b + 0 := by ring
    _ ≤ 2*a*b + (a^2 - 2*a*b + b^2) := add_le_add (le_refl _) h
    _ = a^2 + b^2 := by ring
```

Mathlib tends to put spaces around binary operations like `*` and `^`, but in this example, the more compressed format increases readability. There are a number of things worth noticing. First, an expression `s ≥ t` is definitionally equivalent to `t ≤ s`. In principle, this means one should be able to use them interchangeably. But some of Lean's automation does not recognize the equivalence, so Mathlib tends to favor $\leq$ over $\geq$. Second, we have used the `ring` tactic extensively. It is a real timesaver! Finally, notice that in the second line of the second `calc` proof, instead of writing `by exact add_le_add (le_refl _) h`, we can simply write the proof term `add_le_add (le_refl _) h`.

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

```
example : 2*a*b ≤ a^2 + b^2 := by
  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
```

How nice! We challenge you to use these ideas to prove the following theorem. You can use the theorem `abs_le'.mpr`. You will also need the `constructor` tactic to split a conjunction to two goals; see Section 3.4.

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

#check abs_le'.mpr
```

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

2.4 More examples using apply and rw

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

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

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

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

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

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

```
example : min a b = min b a := by
  apply le_antisymm
  · show min a b ≤ min b a
    apply le_min
    · apply min_le_right
    apply min_le_left
  · show min b a ≤ min a b
    apply le_min
    · apply min_le_right
    apply min_le_left
```

Here we have used dots to separate proofs of different goals. Our usage is inconsistent: at the outer level, we use dots and indentation for both goals, whereas for the nested proofs, we use dots 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 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 := by
  have h : ∀ x y : ℝ, min x y ≤ min y x := by
    intro x y
    apply le_min
    apply min_le_right
    apply min_le_left
  apply le_antisymm
  apply h
  apply h
```

We will say more about the universal quantifier in Section 3.1, but suffice it to say here that the hypothesis `h` says that the desired inequality holds for any `x` and `y`, and the `intro` 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 := by
  apply le_antisymm
  repeat
    apply le_min
    apply min_le_right
    apply min_le_left
```

We encourage you to prove the following as exercises. You can use either of the tricks just described to shorten the first.

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

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

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

Here is one such example:

```
theorem aux : min a b + c ≤ min (a + c) (b + c) := by
  sorry
example : min a b + c = min (a + c) (b + c) := by
  sorry
```

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

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

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

Use it to prove the following variant, using also `add_sub_cancel_right`:

```
example : |a| - |b| ≤ |a - b| :=
  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.

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

example : x ∣ y * x * z := by
  apply dvd_mul_of_dvd_left
  apply dvd_mul_left

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

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

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

```
variable (m n : ℕ)

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

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

```
example : Nat.gcd m n = Nat.gcd n m := by
  sorry
```

Hint: you can use `dvd_antisymm`, but if you do, Lean will complain that the expression is ambiguous between the generic theorem and the version `Nat.dvd_antisymm`, the one specifically for the natural numbers. You can use `_root_.dvd_antisymm` to specify the generic one; either one will work.

2.5 Proving Facts about Algebraic Structures

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

```
variable {α : Type*} [PartialOrder α]
variable (x y z : α)

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

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 carriers of algebraic structures like rings and groups, respectively, but in general Greek letters are used for types, especially when there is little or no structure associated with them.

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

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

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

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

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

```
variable {α : Type*} [Lattice α]
variable (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`.

Using `apply le_trans` when seeing a goal `x ≤ z` is not a great idea. Indeed Lean has no way to guess which intermediate element `y` we want to use. So `apply le_trans` produces three goals that look like `x ≤ ?a`, `?a ≤ z` and α where `?a` (probably with a more complicated auto-generated name) stands for the mysterious `y`. The last goal, with type α, is to provide the value of `y`. It comes lasts because Lean hopes to automatically infer it from the proof of the first goal `x ≤ ?a`. In order to avoid this unappealing situation, you can use the `calc` tactic to explicitly provide `y`. Alternatively, you can use the `trans` tactic which takes `y` as an argument and produces the expected goals `x ≤ y` and `y ≤ z`. Of course you can also avoid this issue by providing directly a full proof such as `exact le_trans inf_le_left inf_le_right`, but this requires a lot more planning.

```
example : x ⊓ y = y ⊓ x := by
  sorry

example : x ⊓ y ⊓ z = x ⊓ (y ⊓ z) := by
  sorry

example : x ⊔ y = y ⊔ x := by
  sorry

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

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

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

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

theorem absorb2 : x ⊔ x ⊓ y = x := by
  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:

```
variable {α : Type*} [DistribLattice α]
variable (x y z : α)

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

```
variable {α : Type*} [Lattice α]
variable (a b c : α)

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

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

It is possible to combine axiomatic structures into larger ones. For example, a *strict 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:

```
variable {R : Type*} [Ring R] [PartialOrder R] [IsStrictOrderedRing R]
variable (a b c : R)

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

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

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

It is then an extended exercise to show that many common facts used to reason about arithmetic and the ordering on the real numbers hold generically for any ordered ring. Here are a couple of examples you can try, using only properties of rings, partial orders, and the facts enumerated in the last two examples (beware that those rings are not assumed to be commutative, so the *ring* tactic is not available):

```
example (h : a ≤ b) : 0 ≤ b - a := by
  sorry

example (h: 0 ≤ b - a) : a ≤ b := by
  sorry

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

```
variable {X : Type*} [MetricSpace X]
variable (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 := by
  sorry
```

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

CHAPTER

THREE

LOGIC

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

3.1 Implication and the Universal Quantifier

Consider the statement after the `#check`:

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

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

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

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

You have already seen that even though the universal quantifier in this statement ranges over objects and the implication arrows introduce hypotheses, Lean treats the two in very similar ways. In particular, if you have proved a theorem of that form, you can apply it to objects and hypotheses in the same way. We will use as an example the following statement that we will help you to prove a bit later:

```
theorem my_lemma : ∀ x y ε : ℝ, 0 < ε → ε ≤ 1 → |x| < ε → |y| < ε → |x * y| < ε :=
  sorry

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

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

end
```

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

```
theorem my_lemma2 : ∀ {x y ε : ℝ}, 0 < ε → ε ≤ 1 → |x| < ε → |y| < ε → |x * y| < ε␣
↪:=
  sorry

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

#check my_lemma2 h₀ h₁ ha hb

end
```

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

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

```
theorem my_lemma3 :
    ∀ {x y ε : ℝ}, 0 < ε → ε ≤ 1 → |x| < ε → |y| < ε → |x * y| < ε := by
  intro x y ε epos ele1 xlt ylt
  sorry
```

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

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

```
theorem my_lemma4 :
    ∀ {x y ε : ℝ}, 0 < ε → ε ≤ 1 → |x| < ε → |y| < ε → |x * y| < ε := by
  intro x y ε epos ele1 xlt ylt
  calc
    |x * y| = |x| * |y| := sorry
    _ ≤ |x| * ε := sorry
    _ < 1 * ε := sorry
    _ = ε := sorry
```

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

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

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

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

In the next example, `fun x ↦ f x + g x` is the function that maps `x` to `f x + g x`. Going from the expression `f x + g x` to this function is called a lambda abstraction in type theory.

```
example (hfa : FnUb f a) (hgb : FnUb g b) : FnUb (fun x ↦ f x + g x) (a + b) := by
  intro x
  dsimp
  apply add_le_add
  apply hfa
  apply hgb
```

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

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

```
example (hfa : FnLb f a) (hgb : FnLb g b) : FnLb (fun x ↦ f x + g x) (a + b) :=
  sorry

example (nnf : FnLb f 0) (nng : FnLb g 0) : FnLb (fun x ↦ f x * g x) 0 :=
  sorry

example (hfa : FnUb f a) (hgb : FnUb g b) (nng : FnLb g 0) (nna : 0 ≤ a) :
    FnUb (fun x ↦ f x * g x) (a * b) :=
  sorry
```

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

```
variable {α : Type*} {R : Type*} [AddCommMonoid R] [PartialOrder R]␣
↪[IsOrderedCancelAddMonoid R]

#check add_le_add

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

theorem fnUb_add {f g : α → R} {a b : R} (hfa : FnUb' f a) (hgb : FnUb' g b) :
    FnUb' (fun x ↦ f x + g x) (a + b) := fun x ↦ add_le_add (hfa x) (hgb x)
```

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

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

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

The property `Monotone f` is defined to be exactly the expression after the colon. We need to put the `@` symbol before `h` because if we don't, Lean expands the implicit arguments to `h` and inserts placeholders.

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

```
example (mf : Monotone f) (mg : Monotone g) : Monotone fun x ↦ f x + g x := by
  intro a b aleb
  apply add_le_add
  apply mf aleb
  apply mg aleb
```

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

```
example (mf : Monotone f) (mg : Monotone g) : Monotone fun x ↦ f x + g x :=
  fun a b aleb ↦ add_le_add (mf aleb) (mg aleb)
```

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

Try proving these, with either tactics or proof terms:

```
example {c : ℝ} (mf : Monotone f) (nnc : 0 ≤ c) : Monotone fun x ↦ c * f x :=
  sorry

example (mf : Monotone f) (mg : Monotone g) : Monotone fun x ↦ f (g x) :=
  sorry
```

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

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

def FnOdd (f : ℝ → ℝ) : Prop :=
  ∀ x, f x = -f (-x)

example (ef : FnEven f) (eg : FnEven g) : FnEven fun x ↦ f x + g x := by
  intro x
  calc
    (fun x ↦ f x + g x) x = f x + g x := rfl
    _ = f (-x) + g (-x) := by rw [ef, eg]


example (of : FnOdd f) (og : FnOdd g) : FnEven fun x ↦ f x * g x := by
```

(continues on next page)

(continued from previous page)

```
  sorry

example (ef : FnEven f) (og : FnOdd g) : FnOdd fun x ↦ f x * g x := by
  sorry

example (ef : FnEven f) (og : FnOdd g) : FnEven fun x ↦ f (g x) := by
  sorry
```

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

You can find implicit universal quantifiers all over the place, once you know how to spot them.

Mathlib includes a good library for manipulating sets. Recall that Lean does not use foundations based on set theory, so here the word set has its mundane meaning of a collection of mathematical objects of some given type α. If `x` has type α and `s` has type `Set α`, then `x ∈ s` is a proposition that asserts that `x` is an element of `s`. If `y` has some different type β then the expression `y ∈ s` makes no sense. Here "makes no sense" means "has no type hence Lean does not accept it as a well-formed statement". This contrasts with Zermelo-Fraenkel set theory for instance where `a ∈ b` is a well-formed statement for every mathematical objects `a` and `b`. For instance `sin ∈ cos` is a well-formed statement in ZF. This defect of set theoretic foundations is an important motivation for not using it in a proof assistant which is meant to assist us by detecting meaningless expressions. In Lean `sin` has type $\mathbb{R} \to \mathbb{R}$ and `cos` has type $\mathbb{R} \to \mathbb{R}$ which is not equal to `Set (ℝ → ℝ)`, even after unfolding definitions, so the statement `sin ∈ cos` makes no sense. One can also use Lean to work on set theory itself. For instance the independence of the continuum hypothesis from the axioms of Zermelo-Fraenkel has been formalized in Lean. But such a meta-theory of set theory is completely beyond the scope of this book.

If `s` and `t` are of type `Set α`, then the subset relation `s ⊆ t` is defined to mean `∀ {x : α}, x ∈ s → x ∈ t`. The variable in the quantifier is marked implicit so that given `h : s ⊆ t` and `h' : x ∈ s`, we can write `h h'` as justification for `x ∈ t`. The following example provides a tactic proof and a proof term justifying the reflexivity of the subset relation, and asks you to do the same for transitivity.

```
variable {α : Type*} (r s t : Set α)

example : s ⊆ s := by
  intro x xs
  exact xs

theorem Subset.refl : s ⊆ s := fun x xs ↦ xs

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

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

```
variable {α : Type*} [PartialOrder α]
variable (s : Set α) (a b : α)

def SetUb (s : Set α) (a : α) :=
  ∀ x, x ∈ s → x ≤ a
```

(continues on next page)

(continued from previous page)

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

We close this section with one last important example. A function f is said to be *injective* if for every x_1 and x_2, if $f(x_1) = f(x_2)$ then $x_1 = x_2$. Mathlib defines `Function.Injective f` with x_1 and x_2 implicit. The next example shows that, on the real numbers, any function that adds a constant is injective. We then ask you to show that multiplication by a nonzero constant is also injective, using the lemma name in the example as a source of inspiration. Recall you should use Ctrl-space completion after guessing the beginning of a lemma name.

```
open Function

example (c : ℝ) : Injective fun x ↦ x + c := by
  intro x₁ x₂ h'
  exact (add_left_inj c).mp h'

example {c : ℝ} (h : c ≠ 0) : Injective fun x ↦ c * x := by
  sorry
```

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

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

example (injg : Injective g) (injf : Injective f) : Injective fun x ↦ g (f x) := by
  sorry
```

3.2 The Existential Quantifier

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

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

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

You can give the `use` tactic proofs as well as data:

```
example : ∃ x : ℝ, 2 < x ∧ x < 3 := by
  have h1 : 2 < (5 : ℝ) / 2 := by norm_num
  have h2 : (5 : ℝ) / 2 < 3 := by norm_num
  use 5 / 2, h1, h2
```

In fact, the `use` tactic automatically tries to use available assumptions as well.

```
example : ∃ x : ℝ, 2 < x ∧ x < 3 := by
  have h : 2 < (5 : ℝ) / 2 ∧ (5 : ℝ) / 2 < 3 := by norm_num
  use 5 / 2
```

Alternatively, we can use Lean's *anonymous constructor* notation to construct a proof of an existential quantifier.

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

Notice that there is no `by`; here we are giving an explicit proof term. The left and right angle brackets, which can be entered as `\<` and `\>` respectively, tell Lean to put together the given data using whatever construction is appropriate for the current goal. We can use the notation without going first into tactic mode:

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

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

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

def FnLb (f : ℝ → ℝ) (a : ℝ) : Prop :=
  ∀ x, a ≤ f x

def FnHasUb (f : ℝ → ℝ) :=
  ∃ a, FnUb f a

def FnHasLb (f : ℝ → ℝ) :=
  ∃ a, FnLb f a
```

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

```
variable {f g : ℝ → ℝ}

example (ubf : FnHasUb f) (ubg : FnHasUb g) : FnHasUb fun x ↦ f x + g x := by
  rcases ubf with ⟨a, ubfa⟩
  rcases ubg with ⟨b, ubgb⟩
  use a + b
  apply fnUb_add ubfa ubgb
```

The `rcases` tactic unpacks the information in the existential quantifier. The annotations like `⟨a, ubfa⟩`, written with the same angle brackets as the anonymous constructors, are known as *patterns*, and they describe the information that we expect to find when we unpack the main argument. Given the hypothesis `ubf` that there is an upper bound for `f`, `rcases ubf with ⟨a, ubfa⟩` adds a new variable `a` for an upper bound to the context, together with the hypothesis `ubfa` that it has the given property. The goal is left unchanged; what *has* changed is that we can now use the new object and the new hypothesis to prove the goal. This is a common method of reasoning in mathematics: we unpack objects whose existence is asserted or implied by some hypothesis, and then use it to establish the existence of something else.

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

```
example (lbf : FnHasLb f) (lbg : FnHasLb g) : FnHasLb fun x ↦ f x + g x := by
  sorry

example {c : ℝ} (ubf : FnHasUb f) (h : c ≥ 0) : FnHasUb fun x ↦ c * f x := by
  sorry
```

The "r" in `rcases` stands for "recursive," because it allows us to use arbitrarily complex patterns to unpack nested data. The `rintro` tactic is a combination of `intro` and `rcases`:

```
example : FnHasUb f → FnHasUb g → FnHasUb fun x ↦ f x + g x := by
  rintro ⟨a, ubfa⟩ ⟨b, ubgb⟩
  exact ⟨a + b, fnUb_add ubfa ubgb⟩
```

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

```
example : FnHasUb f → FnHasUb g → FnHasUb fun x ↦ f x + g x :=
  fun ⟨a, ubfa⟩ ⟨b, ubgb⟩ ↦ ⟨a + b, fnUb_add ubfa ubgb⟩
```

The task of unpacking information in a hypothesis is so important that Lean and Mathlib provide a number of ways to do it. For example, the `obtain` tactic provides suggestive syntax:

```
example (ubf : FnHasUb f) (ubg : FnHasUb g) : FnHasUb fun x ↦ f x + g x := by
  obtain ⟨a, ubfa⟩ := ubf
  obtain ⟨b, ubgb⟩ := ubg
  exact ⟨a + b, fnUb_add ubfa ubgb⟩
```

Think of the first `obtain` instruction as matching the "contents" of `ubf` with the given pattern and assigning the components to the named variables. `rcases` and `obtain` are said to `destruct` their arguments.

Lean also supports syntax that is similar to that used in other functional programming languages:

```
example (ubf : FnHasUb f) (ubg : FnHasUb g) : FnHasUb fun x ↦ f x + g x := by
  cases ubf
  case intro a ubfa =>
    cases ubg
    case intro b ubgb =>
      exact ⟨a + b, fnUb_add ubfa ubgb⟩

example (ubf : FnHasUb f) (ubg : FnHasUb g) : FnHasUb fun x ↦ f x + g x := by
  cases ubf
  next a ubfa =>
    cases ubg
    next b ubgb =>
      exact ⟨a + b, fnUb_add ubfa ubgb⟩

example (ubf : FnHasUb f) (ubg : FnHasUb g) : FnHasUb fun x ↦ f x + g x := by
  match ubf, ubg with
    | ⟨a, ubfa⟩, ⟨b, ubgb⟩ =>
      exact ⟨a + b, fnUb_add ubfa ubgb⟩

example (ubf : FnHasUb f) (ubg : FnHasUb g) : FnHasUb fun x ↦ f x + g x :=
  match ubf, ubg with
    | ⟨a, ubfa⟩, ⟨b, ubgb⟩ =>
      ⟨a + b, fnUb_add ubfa ubgb⟩
```

In the first example, if you put your cursor after `cases ubf`, you will see that the tactic produces a single goal, which Lean has tagged `intro`. (The particular name chosen comes from the internal name for the axiomatic primitive that builds a proof of an existential statement.) The `case` tactic then names the components. The second example is similar, except using `next` instead of `case` means that you can avoid mentioning `intro`. The word `match` in the last two examples highlights that what we are doing here is what computer scientists call "pattern matching." Notice that the third proof begins by `by`, after which the tactic version of `match` expects a tactic proof on the right side of the arrow. The last example is a proof term: there are no tactics in sight.

For the rest of this book, we will stick to `rcases`, `rintro`, and `obtain`, as the preferred ways of using an existential

quantifier. But it can't hurt to see the alternative syntax, especially if there is a chance you will find yourself in the company of computer scientists.

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

```
variable {α : Type*} [CommRing α]

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

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

This proof doesn't provide much insight, but here is one way to motivate it. A *Gaussian integer* is a number of the form $a + bi$ where a and b are integers and $i = \sqrt{-1}$. The *norm* of the Gaussian integer $a + bi$ is, by definition, $a^2 + b^2$. So the norm of a Gaussian integer is a sum of squares, and any sum of squares can be expressed in this way. The theorem above reflects the fact that norm of a product of Gaussian integers is the product of their norms: if x is the norm of $a + bi$ and y in the norm of $c + di$, then xy is the norm of $(a + bi)(c + di)$. Our cryptic proof illustrates the fact that the proof that is easiest to formalize isn't always the most perspicuous one. In Section 7.3, we will provide you with the means to define the Gaussian integers and use them to provide an alternative proof.

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

```
theorem sumOfSquares_mul' {x y : α} (sosx : SumOfSquares x) (sosy : SumOfSquares y) :
    SumOfSquares (x * y) := by
  rcases sosx with ⟨a, b, rfl⟩
  rcases sosy with ⟨c, d, rfl⟩
  use a * c - b * d, a * d + b * c
  ring
```

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

```
example (divab : a ∣ b) (divbc : b ∣ c) : a ∣ c := by
  rcases divab with ⟨d, beq⟩
  rcases divbc with ⟨e, ceq⟩
  rw [ceq, beq]
  use d * e; ring
```

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

Then try proving the following:

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

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

```
example {c : ℝ} : Surjective fun x ↦ x + c := by
  intro x
  use x - c
  dsimp; ring
```

Try this example yourself using the theorem `mul_div_cancel₀`.:

```
example {c : ℝ} (h : c ≠ 0) : Surjective fun x ↦ c * x := by
  sorry
```

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

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

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

```
example {f : ℝ → ℝ} (h : Surjective f) : ∃ x, f x ^ 2 = 4 := by
  rcases h 2 with ⟨x, hx⟩
  use x
  rw [hx]
  norm_num
```

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

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

example (surjg : Surjective g) (surjf : Surjective f) : Surjective fun x ↦ g (f x) :=␣
↪by
  sorry
```

3.3 Negation

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

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

```
example (h : a < b) : ¬b < a := by
  intro h'
```

(continues on next page)

(continued from previous page)

```
  have : a < a := lt_trans h h'
  apply lt_irrefl a this
```

This example introduces a couple of new tricks. First, when you use `have` without providing a label, Lean uses the name `this`, providing a convenient way to refer back to it. Because the proof is so short, we provide an explicit proof term. But what you should really be paying attention to in this proof is the result of the `intro` tactic, which leaves a goal of `False`, and the fact that we eventually prove `False` by applying `lt_irrefl` to a proof of `a < a`.

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

```
example (h : ∀ a, ∃ x, f x > a) : ¬FnHasUb f := by
  intro fnub
  rcases fnub with ⟨a, fnuba⟩
  rcases h a with ⟨x, hx⟩
  have : f x ≤ a := fnuba x
  linarith
```

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

See if you can prove these in a similar way:

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

example : ¬FnHasUb fun x ↦ x :=
  sorry
```

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

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

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

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

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

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

```
example : ¬∀ {f : ℝ → ℝ}, Monotone f → ∀ {a b}, f a ≤ f b → a ≤ b := by
  intro h
  let f := fun x : ℝ ↦ (0 : ℝ)
  have monof : Monotone f := by sorry
  have h' : f 1 ≤ f 0 := le_refl _
  sorry
```

This example introduces the `let` tactic, which adds a *local definition* to the context. If you put the cursor after the `let` command, in the goal window you will see that the definition `f : ℝ → ℝ := fun x ↦ 0` has been added to the

context. Lean will unfold the definition of `f` when it has to. In particular, when we prove `f 1 ≤ f 0` with `le_refl`, Lean reduces `f 1` and `f 0` to `0`.

Use `le_of_not_gt` to prove the following:

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

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

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

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

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

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

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

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

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

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

```
example (h : ¬¬Q) : Q := by
  sorry

example (h : Q) : ¬¬Q := by
  sorry
```

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

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

It is often tedious to work with compound statements with a negation in front, and it is a common mathematical pattern to

replace such statements with equivalent forms in which the negation has been pushed inward. To facilitate this, Mathlib offers a `push_neg` tactic, which restates the goal in this way. The command `push_neg at h` restates the hypothesis `h`.

```
example (h : ¬∀ a, ∃ x, f x > a) : FnHasUb f := by
  push_neg at h
  exact h

example (h : ¬FnHasUb f) : ∀ a, ∃ x, f x > a := by
  dsimp only [FnHasUb, FnUb] at h
  push_neg at h
  exact h
```

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

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

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

```
example (h : ¬FnHasUb f) : ∀ a, ∃ x, f x > a := by
  contrapose! h
  exact h

example (x : ℝ) (h : ∀ ε > 0, x ≤ ε) : x ≤ 0 := by
  contrapose! h
  use x / 2
  constructor <;> linarith
```

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

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

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

```
example (h : 0 < 0) : a > 37 := by
  exfalso
  apply lt_irrefl 0 h

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

example (h : 0 < 0) : a > 37 := by
  have h' : ¬0 < 0 := lt_irrefl 0
  contradiction
```

The `exfalso` tactic replaces the current goal with the goal of proving `False`. Given `h : P` and `h' : ¬ P`, the term `absurd h h'` establishes any proposition. Finally, the `contradiction` tactic tries to close a goal by finding

a contradiction in the hypotheses, such as a pair of the form `h : P` and `h' : ¬ P`. Of course, in this example, `linarith` also works.

3.4 Conjunction and Iff

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

```
example {x y : ℝ} (h₀ : x ≤ y) (h₁ : ¬y ≤ x) : x ≤ y ∧ x ≠ y := by
  constructor
  · assumption
  intro h
  apply h₁
  rw [h]
```

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

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

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

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

```
example {x y : ℝ} (h : x ≤ y ∧ x ≠ y) : ¬y ≤ x := by
  rcases h with ⟨h₀, h₁⟩
  contrapose! h₁
  exact le_antisymm h₀ h₁

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

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

In analogy to the `obtain` tactic, there is also a pattern-matching `have`:

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

In contrast to `rcases`, here the `have` tactic leaves `h` in the context. And even though we won't use them, once again we have the computer scientists' pattern-matching syntax:

```
example {x y : ℝ} (h : x ≤ y ∧ x ≠ y) : ¬y ≤ x := by
  cases h
  case intro h₀ h₁ =>
    contrapose! h₁
    exact le_antisymm h₀ h₁

example {x y : ℝ} (h : x ≤ y ∧ x ≠ y) : ¬y ≤ x := by
  cases h
  next h₀ h₁ =>
    contrapose! h₁
    exact le_antisymm h₀ h₁

example {x y : ℝ} (h : x ≤ y ∧ x ≠ y) : ¬y ≤ x := by
  match h with
    | ⟨h₀, h₁⟩ =>
        contrapose! h₁
        exact le_antisymm h₀ h₁
```

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

```
example {x y : ℝ} (h : x ≤ y ∧ x ≠ y) : ¬y ≤ x := by
  intro h'
  apply h.right
  exact le_antisymm h.left h'

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

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

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

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

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

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

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

You can also use the `use` tactic:

```
example : ∃ x : ℝ, 2 < x ∧ x < 4 := by
  use 5 / 2
  constructor <;> norm_num

example : ∃ m n : ℕ, 4 < m ∧ m < n ∧ n < 10 ∧ Nat.Prime m ∧ Nat.Prime n := by
  use 5
  use 7
  norm_num
```

(continues on next page)

(continued from previous page)

```
example {x y : ℝ} : x ≤ y ∧ x ≠ y → x ≤ y ∧ ¬y ≤ x := by
  rintro ⟨h₀, h₁⟩
  use h₀
  exact fun h' ↦ h₁ (le_antisymm h₀ h')
```

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

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

```
example {x y : ℝ} (h : x ≤ y) : ¬y ≤ x ↔ x ≠ y := by
  constructor
  · contrapose!
    rintro rfl
    rfl
  contrapose!
  exact le_antisymm h

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

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

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

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

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

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

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

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

```
example (x : ℝ) : |x + 3| < 5 → -8 < x ∧ x < 2 := by
  rw [abs_lt]
  intro h
  constructor <;> linarith

example : 3 ∣ Nat.gcd 6 15 := by
  rw [Nat.dvd_gcd_iff]
  constructor <;> norm_num
```

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

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

example : ¬Monotone fun x : ℝ ↦ -x := by
  sorry
```

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

```
variable {α : Type*} [PartialOrder α]
variable (a b : α)

example : a < b ↔ a ≤ b ∧ a ≠ b := by
  rw [lt_iff_le_not_le]
  sorry
```

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

```
variable {α : Type*} [Preorder α]
variable (a b c : α)

example : ¬a < a := by
  rw [lt_iff_le_not_le]
  sorry

example : a < b → b < c → a < c := by
  simp only [lt_iff_le_not_le]
  sorry
```

3.5 Disjunction

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

```
variable {x y : ℝ}

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

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

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

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

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

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

```
example : x < |y| → x < y ∨ x < -y := by
  rcases le_or_gt 0 y with h | h
  · rw [abs_of_nonneg h]
    intro h; left; exact h
  · rw [abs_of_neg h]
    intro h; right; exact h
```

Notice that the pattern changes from $\langle h_0, h_1 \rangle$ in the case of a conjunction to $h_0 \mid h_1$ in the case of a disjunction. Think of the first pattern as matching against data the contains *both* an h_0 and a h_1, whereas second pattern, with the bar, matches against data that contains *either* an h_0 or h_1. In this case, because the two goals are separate, we have chosen to use the same name, `h`, in each case.

The absolute value function is defined in such a way that we can immediately prove that $x \geq 0$ implies $|x| = x$ (this is the theorem `abs_of_nonneg`) and $x < 0$ implies $|x| = -x$ (this is `abs_of_neg`). The expression `le_or_gt 0 x` establishes $0 \leq x \vee x < 0$, allowing us to split on those two cases.

Lean also supports the computer scientists' pattern-matching syntax for disjunction. Now the `cases` tactic is more attractive, because it allows us to name each `case`, and name the hypothesis that is introduced closer to where it is used.

```
example : x < |y| → x < y ∨ x < -y := by
  cases le_or_gt 0 y
  case inl h =>
    rw [abs_of_nonneg h]
    intro h; left; exact h
  case inr h =>
    rw [abs_of_neg h]
    intro h; right; exact h
```

The names `inl` and `inr` are short for "intro left" and "intro right," respectively. Using `case` has the advantage that you can prove the cases in either order; Lean uses the tag to find the relevant goal. If you don't care about that, you can use `next`, or `match`, or even a pattern-matching `have`.

```
example : x < |y| → x < y ∨ x < -y := by
  cases le_or_gt 0 y
  next h =>
    rw [abs_of_nonneg h]
    intro h; left; exact h
  next h =>
    rw [abs_of_neg h]
```

(continues on next page)

(continued from previous page)

```
    intro h; right; exact h

example : x < |y| → x < y ∨ x < -y := by
  match le_or_gt 0 y with
    | Or.inl h =>
      rw [abs_of_nonneg h]
      intro h; left; exact h
    | Or.inr h =>
      rw [abs_of_neg h]
      intro h; right; exact h
```

In the case of `match`, we need to use the full names `Or.inl` and `Or.inr` of the canonical ways to prove a disjunction. In this textbook, we will generally use `rcases` to split on the cases of a disjunction.

Try proving the triangle inequality using the first two theorems in the next snippet. They are given the same names they have in Mathlib.

```
namespace MyAbs

theorem le_abs_self (x : ℝ) : x ≤ |x| := by
  sorry

theorem neg_le_abs_self (x : ℝ) : -x ≤ |x| := by
  sorry

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
variable {R : Type*} [CommRing R] [IsDomain R]
variable (x y : R)

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

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

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

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

```
example (P : Prop) : ¬¬P → P := by
  intro h
  cases em P
  · assumption
  · contradiction
```

Alternatively, you can use the `by_cases` tactic.

```
example (P : Prop) : ¬¬P → P := by
  intro h
  by_cases h' : P
  · assumption
  contradiction
```

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

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

3.6 Sequences and Convergence

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

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

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

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

```
example : (fun x y : ℝ ↦ (x + y) ^ 2) = fun x y : ℝ ↦ x ^ 2 + 2 * x * y + y ^ 2 := by
  ext
  ring
```

We'll see later that `ext` is actually more general, and also one can specify the name of the variables that appear. For instance you can try to replace `ext` with `ext u v` in the above proof. The second tactic, the `congr` tactic, allows us to prove an equation between two expressions by reconciling the parts that are different:

```
example (a b : ℝ) : |a| = |a - b + b| := by
  congr
  ring
```

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

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

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

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

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

```
theorem convergesTo_const (a : ℝ) : ConvergesTo (fun x : ℕ ↦ a) a := by
  intro ε εpos
  use 0
  intro n nge
  rw [sub_self, abs_zero]
  apply εpos
```

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

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

```
theorem convergesTo_add {s t : ℕ → ℝ} {a b : ℝ}
      (cs : ConvergesTo s a) (ct : ConvergesTo t b) :
    ConvergesTo (fun n ↦ s n + t n) (a + b) := by
  intro ε εpos
  dsimp -- this line is not needed but cleans up the goal a bit.
  have ε2pos : 0 < ε / 2 := by linarith
  rcases cs (ε / 2) ε2pos with ⟨Ns, hs⟩
  rcases ct (ε / 2) ε2pos with ⟨Nt, ht⟩
  use max Ns Nt
  sorry
```

As hints, you can use `le_of_max_le_left` and `le_of_max_le_right`, and `norm_num` can prove `ε / 2 + ε / 2 = ε`. Also, it is helpful to use the `congr` tactic to show that `|s n + t n - (a + b)|` is equal to `|(s n - a) + (t n - b)|`, since then you can use the triangle inequality. Notice that we marked all the variables `s`, `t`, `a`, and `b` implicit because they can be inferred from the hypotheses.

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

```
theorem convergesTo_mul_const {s : ℕ → ℝ} {a : ℝ} (c : ℝ) (cs : ConvergesTo s a) :
    ConvergesTo (fun n ↦ c * s n) (c * a) := by
  by_cases h : c = 0
  · convert convergesTo_const 0
    · rw [h]
      ring
    rw [h]
    ring
  have acpos : 0 < |c| := abs_pos.mpr h
  sorry
```

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

```
theorem exists_abs_le_of_convergesTo {s : ℕ → ℝ} {a : ℝ} (cs : ConvergesTo s a) :
    ∃ N b, ∀ n, N ≤ n → |s n| < b := by
  rcases cs 1 zero_lt_one with ⟨N, h⟩
  use N, |a| + 1
  sorry
```

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

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

```
theorem aux {s t : ℕ → ℝ} {a : ℝ} (cs : ConvergesTo s a) (ct : ConvergesTo t 0) :
    ConvergesTo (fun n ↦ s n * t n) 0 := by
  intro ε εpos
  dsimp
  rcases exists_abs_le_of_convergesTo cs with ⟨N₀, B, h₀⟩
  have Bpos : 0 < B := lt_of_le_of_lt (abs_nonneg _) (h₀ N₀ (le_refl _))
  have pos₀ : ε / B > 0 := div_pos εpos Bpos
  rcases ct _ pos₀ with ⟨N₁, h₁⟩
  sorry
```

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

```
theorem convergesTo_mul {s t : ℕ → ℝ} {a b : ℝ}
      (cs : ConvergesTo s a) (ct : ConvergesTo t b) :
    ConvergesTo (fun n ↦ s n * t n) (a * b) := by
  have h₁ : ConvergesTo (fun n ↦ s n * (t n + -b)) 0 := by
    apply aux cs
    convert convergesTo_add ct (convergesTo_const (-b))
    ring
  have := convergesTo_add h₁ (convergesTo_mul_const b cs)
  convert convergesTo_add h₁ (convergesTo_mul_const b cs) using 1
  · ext; ring
  ring
```

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

```
theorem convergesTo_unique {s : ℕ → ℝ} {a b : ℝ}
      (sa : ConvergesTo s a) (sb : ConvergesTo s b) :
    a = b := by
  by_contra abne
  have : |a - b| > 0 := by sorry
  let ε := |a - b| / 2
  have εpos : ε > 0 := by
    change |a - b| / 2 > 0
    linarith
  rcases sa ε εpos with ⟨Na, hNa⟩
  rcases sb ε εpos with ⟨Nb, hNb⟩
  let N := max Na Nb
  have absa : |s N - a| < ε := by sorry
  have absb : |s N - b| < ε := by sorry
  have : |a - b| < |a - b| := by sorry
  exact lt_irrefl _ this
```

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

```
variable {α : Type*} [LinearOrder α]

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

In Section 11.1, we will see that Mathlib has mechanisms for dealing with convergence in vastly more general terms, not only abstracting away particular features of the domain and codomain, but also abstracting over different types of convergence.

CHAPTER

FOUR

SETS AND FUNCTIONS

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

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

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

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

4.1 Sets

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

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

```
variable {α : Type*}
variable (s t u : Set α)
open Set

example (h : s ⊆ t) : s ∩ u ⊆ t ∩ u := by
  rw [subset_def, inter_def, inter_def]
  rw [subset_def] at h
  simp only [mem_setOf]
```

(continues on next page)

(continued from previous page)

```
  rintro x ⟨xs, xu⟩
  exact ⟨h _ xs, xu⟩

example (h : s ⊆ t) : s ∩ u ⊆ t ∩ u := by
  simp only [subset_def, mem_inter_iff] at *
  rintro x ⟨xs, xu⟩
  exact ⟨h _ xs, xu⟩
```

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

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

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

```
example (h : s ⊆ t) : s ∩ u ⊆ t ∩ u :=
  fun x ⟨xs, xu⟩ ↦ ⟨h xs, xu⟩
```

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

```
example : s ∩ (t ∪ u) ⊆ s ∩ t ∪ s ∩ u := by
  intro x hx
  have xs : x ∈ s := hx.1
  have xtu : x ∈ t ∪ u := hx.2
  rcases xtu with xt | xu
  · left
    show x ∈ s ∩ t
    exact ⟨xs, xt⟩
  · right
    show x ∈ s ∩ u
    exact ⟨xs, xu⟩
```

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

```
example : s ∩ (t ∪ u) ⊆ s ∩ t ∪ s ∩ u := by
  rintro x ⟨xs, xt | xu⟩
  · left; exact ⟨xs, xt⟩
  · right; exact ⟨xs, xu⟩
```

As an exercise, try proving the other inclusion:

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

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

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

```
example : (s \ t) \ u ⊆ s \ (t ∪ u) := by
  intro x xstu
  have xs : x ∈ s := xstu.1.1
  have xnt : x ∉ t := xstu.1.2
  have xnu : x ∉ u := xstu.2
  constructor
  · exact xs
  intro xtu
  -- x ∈ t ∨ x ∈ u
  rcases xtu with xt | xu
  · show False; exact xnt xt
  · show False; exact xnu xu

example : (s \ t) \ u ⊆ s \ (t ∪ u) := by
  rintro x ⟨⟨xs, xnt⟩, xnu⟩
  use xs
  rintro (xt | xu) <;> contradiction
```

As an exercise, prove the reverse inclusion:

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

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

```
example : s ∩ t = t ∩ s := by
  ext x
  simp only [mem_inter_iff]
  constructor
  · rintro ⟨xs, xt⟩; exact ⟨xt, xs⟩
  · rintro ⟨xt, xs⟩; exact ⟨xs, xt⟩
```

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

```
example : s ∩ t = t ∩ s :=
  Set.ext fun x ↦ ⟨fun ⟨xs, xt⟩ ↦ ⟨xt, xs⟩, fun ⟨xt, xs⟩ ↦ ⟨xs, xt⟩⟩
```

Here is an even shorter proof, using the simplifier:

```
example : s ∩ t = t ∩ s := by ext x; simp [and_comm]
```

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

```
example : s ∩ t = t ∩ s := by
  apply Subset.antisymm
  · rintro x ⟨xs, xt⟩; exact ⟨xt, xs⟩
  · rintro x ⟨xt, xs⟩; exact ⟨xs, xt⟩
```

Try finishing this proof term:

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

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

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

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

example : s ∪ s ∩ t = s := by
  sorry

example : s \ t ∪ t = s ∪ t := by
  sorry

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

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

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

```
def evens : Set ℕ :=
  { n | Even n }

def odds : Set ℕ :=
  { n | ¬Even n }

example : evens ∪ odds = univ := by
  rw [evens, odds]
  ext n
  simp [-Nat.not_even_iff_odd]
  apply Classical.em
```

You should step through this proof and make sure you understand what is going on. Note we tell the simplifier to *not* use the lemma `Nat.not_even_iff` because we want to keep `¬ Even n` in our goal. Try deleting the line `rw [evens, odds]` and confirm that the proof still works.

In fact, set-builder notation is used to define

- `s ∩ t` as `{x | x ∈ s ∧ x ∈ t}`,
- `s ∪ t` as `{x | x ∈ s ∨ x ∈ t}`,
- $\emptyset$ as `{x | False}`, and
- `univ` as `{x | True}`.

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

```
example (x : ℕ) (h : x ∈ (∅ : Set ℕ)) : False :=
  h

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

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

and Nat.odd_iff.

```
example : { n | Nat.Prime n } ∩ { n | n > 2 } ⊆ { n | ¬Even n } := by
  sorry
```

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

```
#print Prime

#print Nat.Prime

example (n : ℕ) : Prime n ↔ Nat.Prime n :=
  Nat.prime_iff.symm

example (n : ℕ) (h : Prime n) : Nat.Prime n := by
  rw [Nat.prime_iff]
  exact h
```

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

```
example (n : ℕ) (h : Prime n) : Nat.Prime n := by
  rwa [Nat.prime_iff]
```

Lean introduces the notation ∀ x ∈ s, ..., "for every x in s .," as an abbreviation for ∀ x, x ∈ s → .... It also introduces the notation ∃ x ∈ s, ..., "there exists an x in s such that .." These are sometimes known as *bounded quantifiers*, because the construction serves to restrict their significance to the set s. As a result, theorems in the library that make use of them often contain ball or bex in the name. The theorem bex_def asserts that ∃ x ∈ s, ... is equivalent to ∃ x, x ∈ s ∧ ..., but when they are used with rintro, use, and anonymous constructors, these two expressions behave roughly the same. As a result, we usually don't need to use bex_def to transform them explicitly. Here are some examples of how they are used:

```
variable (s t : Set ℕ)

example (h₀ : ∀ x ∈ s, ¬Even x) (h₁ : ∀ x ∈ s, Prime x) : ∀ x ∈ s, ¬Even x ∧ Prime x␣
↪:= by
  intro x xs
  constructor
  · apply h₀ x xs
  apply h₁ x xs

example (h : ∃ x ∈ s, ¬Even x ∧ Prime x) : ∃ x ∈ s, Prime x := by
  rcases h with ⟨x, xs, _, prime_x⟩
  use x, xs
```

See if you can prove these slight variations:

```
section
variable (ssubt : s ⊆ t)

example (h₀ : ∀ x ∈ t, ¬Even x) (h₁ : ∀ x ∈ t, Prime x) : ∀ x ∈ s, ¬Even x ∧ Prime x␣
↪:= by
  sorry

example (h : ∃ x ∈ s, ¬Even x ∧ Prime x) : ∃ x ∈ t, Prime x := by
```

(continues on next page)

(continued from previous page)

```
  sorry

end
```

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

```
variable {α I : Type*}
variable (A B : I → Set α)
variable (s : Set α)

open Set

example : (s ∩ ⋃ i, A i) = ⋃ i, A i ∩ s := by
  ext x
  simp only [mem_inter_iff, mem_iUnion]
  constructor
  · rintro ⟨xs, ⟨i, xAi⟩⟩
    exact ⟨i, xAi, xs⟩
  rintro ⟨i, xAi, xs⟩
  exact ⟨xs, ⟨i, xAi⟩⟩

example : (⋂ i, A i ∩ B i) = (⋂ i, A i) ∩ ⋂ i, B i := by
  ext x
  simp only [mem_inter_iff, mem_iInter]
  constructor
  · intro h
    constructor
    · intro i
      exact (h i).1
    intro i
    exact (h i).2
  rintro ⟨h1, h2⟩ i
  constructor
  · exact h1 i
  exact h2 i
```

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

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

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

Mathlib also has bounded unions and intersections, which are analogous to the bounded quantifiers. You can unpack their meaning with `mem_iUnion₂` and `mem_iInter₂`. As the following examples show, Lean's simplifier carries out these replacements as well.

```
def primes : Set ℕ :=
  { x | Nat.Prime x }

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

(continues on next page)

(continued from previous page)

```
  ext
  rw [mem_iUnion₂]
  simp

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

example : (⋂ p ∈ primes, { x | ¬p ∣ x }) ⊆ { x | x = 1 } := by
  intro x
  contrapose!
  simp
  apply Nat.exists_prime_and_dvd
```

Try solving the following example, which is similar. If you start typing `eq_univ`, tab completion will tell you that `apply eq_univ_of_forall` is a good way to start the proof. We also recommend using the theorem `Nat.exists_infinite_primes`.

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

Give a collection of sets, `s : Set (Set α)`, their union, `⋃₀ s`, has type `Set α` and is defined as `{x | ∃ t ∈ s, x ∈ t}`. Similarly, their intersection, `⋂₀ s`, is defined as `{x | ∀ t ∈ s, x ∈ t}`. These operations are called `sUnion` and `sInter`, respectively. The following examples show their relationship to bounded union and intersection.

```
variable {α : Type*} (s : Set (Set α))

example : ⋃₀ s = ⋃ t ∈ s, t := by
  ext x
  rw [mem_iUnion₂]
  simp

example : ⋂₀ s = ⋂ t ∈ s, t := by
  ext x
  rw [mem_iInter₂]
  rfl
```

In the library, these identities are called `sUnion_eq_biUnion` and `sInter_eq_biInter`.

4.2 Functions

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

```
variable {α β : Type*}
variable (f : α → β)
variable (s t : Set α)
variable (u v : Set β)

open Function
open Set

example : f ⁻¹' (u ∩ v) = f ⁻¹' u ∩ f ⁻¹' v := by
```

(continues on next page)

(continued from previous page)

```
  ext
  rfl
```

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

```
example : f '' (s ∪ t) = f '' s ∪ f '' t := by
  ext y; constructor
  · rintro ⟨x, xs | xt, rfl⟩
    · left
      use x, xs
    right
    use x, xt
  rintro (⟨x, xs, rfl⟩ | ⟨x, xt, rfl⟩)
  · use x, Or.inl xs
  use x, Or.inr xt
```

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

Here is another example:

```
example : s ⊆ f ⁻¹' (f '' s) := by
  intro x xs
  show f x ∈ f '' s
  use x, xs
```

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

The following equivalence is a good exercise:

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

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

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

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

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

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

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

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

(continues on next page)

(continued from previous page)

```
  sorry

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

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

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

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

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

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

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

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

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

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

```
variable {I : Type*} (A : I → Set α) (B : I → Set β)

example : (f '' ⋃ i, A i) = ⋃ i, f '' A i := by
  sorry

example : (f '' ⋂ i, A i) ⊆ ⋂ i, f '' A i := by
  sorry

example (i : I) (injf : Injective f) : (⋂ i, f '' A i) ⊆ f '' ⋂ i, A i := by
  sorry

example : (f ⁻¹' ⋃ i, B i) = ⋃ i, f ⁻¹' B i := by
  sorry

example : (f ⁻¹' ⋂ i, B i) = ⋂ i, f ⁻¹' B i := by
  sorry
```

The library defines a predicate `InjOn f s` to say that `f` is injective on `s`. It is defined as follows:

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

The statement `Injective f` is provably equivalent to `InjOn f univ`. Similarly, the library defines `range f` to

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

Here are some examples of `InjOn` and `range` in use:

```
open Set Real

example : InjOn log { x | x > 0 } := by
  intro x xpos y ypos
  intro e
  -- log x = log y
  calc
    x = exp (log x) := by rw [exp_log xpos]
    _ = exp (log y) := by rw [e]
    _ = y := by rw [exp_log ypos]


example : range exp = { y | y > 0 } := by
  ext y; constructor
  · rintro ⟨x, rfl⟩
    apply exp_pos
  intro ypos
  use log y
  rw [exp_log ypos]
```

Try proving these:

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

example : InjOn (fun x ↦ x ^ 2) { x : ℝ | x ≥ 0 } := by
  sorry

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

example : (range fun x ↦ x ^ 2) = { y : ℝ | y ≥ 0 } := by
  sorry
```

To define the inverse of a function `f : α → β`, we will use two new ingredients. First, we need to deal with the fact that an arbitrary type in Lean may be empty. To define the inverse to `f` at `y` when there is no `x` satisfying `f x = y`, we want to assign a default value in α. Adding the annotation `[Inhabited α]` as a variable is tantamount to assuming that α has a preferred element, which is denoted `default`. Second, in the case where there is more than one `x` such that `f x = y`, the inverse function needs to *choose* one of them. This requires an appeal to the *axiom of choice*. Lean allows various ways of accessing it; one convenient method is to use the classical `choose` operator, illustrated below.

```
variable {α β : Type*} [Inhabited α]

#check (default : α)

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

#check Classical.choose h

example : P (Classical.choose h) :=
  Classical.choose_spec h
```

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

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

```
noncomputable section

open Classical

def inverse (f : α → β) : β → α := fun y : β ↦
  if h : ∃ x, f x = y then Classical.choose h else default

theorem inverse_spec {f : α → β} (y : β) (h : ∃ x, f x = y) : f (inverse f y) = y :=␣
↪by
  rw [inverse, dif_pos h]
  exact Classical.choose_spec h
```

The lines `noncomputable section` and `open Classical` are needed because we are using classical logic in an essential way. On input `y`, the function `inverse f` returns some value of `x` satisfying `f x = y` if there is one, and a default element of α otherwise. This is an instance of a *dependent if* construction, since in the positive case, the value returned, `Classical.choose h`, depends on the assumption `h`. The identity `dif_pos h` rewrites `if h : e then a else b` to `a` given `h : e`, and, similarly, `dif_neg h` rewrites it to `b` given `h : ¬ e`. There are also versions `if_pos` and `if_neg` that works for non-dependent if constructions and will be used in the next section. The theorem `inverse_spec` says that `inverse f` meets the first part of this specification.

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

```
variable (f : α → β)

open Function

example : Injective f ↔ LeftInverse (inverse f) f :=
  sorry

example : Surjective f ↔ RightInverse (inverse f) f :=
  sorry
```

We close this section with a type-theoretic statement of Cantor's famous theorem that there is no surjective function from a set to its power set. See if you can understand the proof, and then fill in the two lines that are missing.

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

(continues on next page)

(continued from previous page)

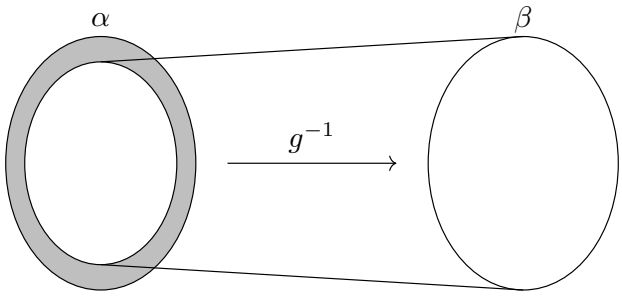
```
  sorry
  contradiction
```

4.3 The Schröder-Bernstein Theorem

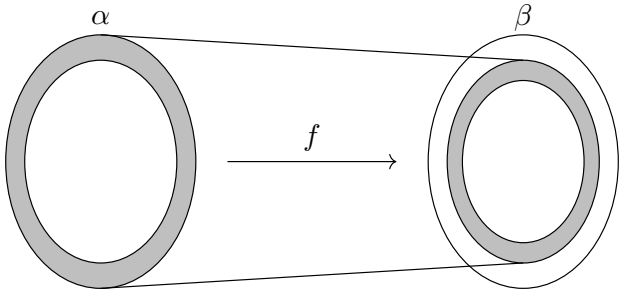
We close this chapter with an elementary but nontrivial theorem of set theory. Let α and β be sets. (In our formalization, they will actually be types.) Suppose $f : \alpha \to \beta$ and $g : \beta \to \alpha$ are both injective. Intuitively, this means that α is no bigger than β and vice-versa. If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

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

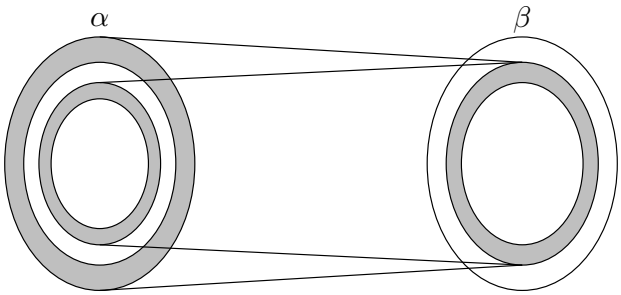
To understand the idea behind the proof, consider the image of the map g in α. On that image, the inverse of g is defined and is a bijection with β.



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



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



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

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

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

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

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

```
noncomputable section
open Classical
variable {α β : Type*} [Nonempty β]
```

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

```
#check (invFun g : α → β)
#check (leftInverse_invFun : Injective g → LeftInverse (invFun g) g)
#check (leftInverse_invFun : Injective g → ∀ y, invFun g (g y) = y)
#check (invFun_eq : (∃ y, g y = x) → g (invFun g x) = x)
```

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

```
variable (f : α → β) (g : β → α)

def sbAux : ℕ → Set α
  | 0 => univ \ g '' univ
  | n + 1 => g '' (f '' sbAux n)

def sbSet :=
  ⋃ n, sbAux f g n
```

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

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

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

```
def sbFun (x : α) : β :=
  if x ∈ sbSet f g then f x else invFun g x
```

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

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

```
theorem sb_right_inv {x : α} (hx : x ∉ sbSet f g) : g (invFun g x) = x := by
  have : x ∈ g '' univ := by
    contrapose! hx
    rw [sbSet, mem_iUnion]
    use 0
    rw [sbAux, mem_diff]
    sorry
  have : ∃ y, g y = x := by
    sorry
  sorry
```

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

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

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

```
theorem sb_injective (hf : Injective f) : Injective (sbFun f g) := by
  set A := sbSet f g with A_def
  set h := sbFun f g with h_def
  intro x₁ x₂
  intro (hxeq : h x₁ = h x₂)
  show x₁ = x₂
  simp only [h_def, sbFun, ← A_def] at hxeq
  by_cases xA : x₁ ∈ A ∨ x₂ ∈ A
  · wlog x₁A : x₁ ∈ A generalizing x₁ x₂ hxeq xA
    · symm
      apply this hxeq.symm xA.symm (xA.resolve_left x₁A)
    have x₂A : x₂ ∈ A := by
      apply _root_.not_imp_self.mp
      intro (x₂nA : x₂ ∉ A)
```

(continues on next page)

(continued from previous page)

```
      rw [if_pos x₁A, if_neg x₂nA] at hxeq
      rw [A_def, sbSet, mem_iUnion] at x₁A
      have x₂eq : x₂ = g (f x₁) := by
        sorry
      rcases x₁A with ⟨n, hn⟩
      rw [A_def, sbSet, mem_iUnion]
      use n + 1
      simp [sbAux]
      exact ⟨x₁, hn, x₂eq.symm⟩
    sorry
  push_neg at xA
  sorry
```

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

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

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

Once again, we encourage you to step through the proof and fill in the missing parts. The tactic `rcases n with _ | n` splits on the cases `g y ∈ sbAux f g 0` and `g y ∈ sbAux f g (n + 1)`. In both cases, calling the simplifier with `simp [sbAux]` applies the corresponding defining equation of `sbAux`.

```
theorem sb_surjective (hg : Injective g) : Surjective (sbFun f g) := by
  set A := sbSet f g with A_def
  set h := sbFun f g with h_def
  intro y
  by_cases gyA : g y ∈ A
  · rw [A_def, sbSet, mem_iUnion] at gyA
    rcases gyA with ⟨n, hn⟩
    rcases n with _ | n
    · simp [sbAux] at hn
    simp [sbAux] at hn
    rcases hn with ⟨x, xmem, hx⟩
    use x
    have : x ∈ A := by
      rw [A_def, sbSet, mem_iUnion]
      exact ⟨n, xmem⟩
    rw [h_def, sbFun, if_pos this]
    apply hg hx

  sorry
```

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

```
theorem schroeder_bernstein {f : α → β} {g : β → α} (hf : Injective f) (hg :␣
↪Injective g) :
```

(continues on next page)

(continued from previous page)

```
  ∃ h : α → β, Bijective h :=
⟨sbFun f g, sb_injective f g hf, sb_surjective f g hg⟩
```

CHAPTER

FIVE

ELEMENTARY NUMBER THEORY

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

5.1 Irrational Roots

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

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

```
#print Nat.Coprime

example (m n : Nat) (h : m.Coprime n) : m.gcd n = 1 :=
  h

example (m n : Nat) (h : m.Coprime n) : m.gcd n = 1 := by
  rw [Nat.Coprime] at h
  exact h

example : Nat.Coprime 12 7 := by norm_num

example : Nat.gcd 12 8 = 4 := by norm_num
```

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

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

```
#check Nat.prime_def_lt
```

(continues on next page)

(continued from previous page)

```
example (p : ℕ) (prime_p : Nat.Prime p) : 2 ≤ p ∧ ∀ m : ℕ, m < p → m ∣ p → m = 1 :=␣
↪by
  rwa [Nat.prime_def_lt] at prime_p

#check Nat.Prime.eq_one_or_self_of_dvd

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

example : Nat.Prime 17 := by norm_num

-- commonly used
example : Nat.Prime 2 :=
  Nat.prime_two

example : Nat.Prime 3 :=
  Nat.prime_three
```

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

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

```
#check Nat.Prime.dvd_mul
#check Nat.Prime.dvd_mul Nat.prime_two
#check Nat.prime_two.dvd_mul

theorem even_of_even_sqr {m : ℕ} (h : 2 ∣ m ^ 2) : 2 ∣ m := by
  rw [pow_two, Nat.prime_two.dvd_mul] at h
  cases h <;> assumption

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

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

```
example (a b c : Nat) (h : a * b = a * c) (h' : a ≠ 0) : b = c :=
  -- apply? suggests the following:
  (mul_right_inj' h').mp h
```

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

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

(continues on next page)

(continued from previous page)

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

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

```
example {m n p : ℕ} (coprime_mn : m.Coprime n) (prime_p : p.Prime) : m ^ 2 ≠ p * n ^␣
↪2 := by
  sorry
```

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

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

```
#check Nat.primeFactorsList
#check Nat.prime_of_mem_primeFactorsList
#check Nat.prod_primeFactorsList
#check Nat.primeFactorsList_unique
```

You can browse these theorems and others nearby, even though we have not talked about list membership, products, or permutations yet. We won't need any of that for the task at hand. We will instead use the fact that Mathlib has a function `Nat.factorization`, that represents the same data as a function. Specifically, `Nat.factorization n p`, which we can also write `n.factorization p`, returns the multiplicity of `p` in the prime factorization of `n`. We will use the following three facts.

```
theorem factorization_mul' {m n : ℕ} (mnez : m ≠ 0) (nnez : n ≠ 0) (p : ℕ) :
    (m * n).factorization p = m.factorization p + n.factorization p := by
  rw [Nat.factorization_mul mnez nnez]
  rfl

theorem factorization_pow' (n k p : ℕ) :
    (n ^ k).factorization p = k * n.factorization p := by
  rw [Nat.factorization_pow]
  rfl

theorem Nat.Prime.factorization' {p : ℕ} (prime_p : p.Prime) :
    p.factorization p = 1 := by
```

(continues on next page)

(continued from previous page)

```
  rw [prime_p.factorization]
  simp
```

In fact, `n.factorization` is defined in Lean as a function of finite support, which explains the strange notation you will see as you step through the proofs above. Don't worry about this now. For our purposes here, we can use the three theorems above as a black box.

The next example shows that the simplifier is smart enough to replace `n^2 ≠ 0` by `n ≠ 0`. The tactic `simpa` just calls `simp` followed by `assumption`.

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

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

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

To use `Nat.count_factors_mul_of_pos` with `r * n^k`, we need to know that `r` is positive. But when `r` is zero, the theorem below is trivial, and easily proved by the simplifier. So the proof is carried out in cases. The line `rcases r with _ | r` replaces the goal with two versions: one in which `r` is replaced by `0`, and the other in which `r` is replaces by `r + 1`. In the second case, we can use the theorem `r.succ_ne_zero`, which establishes `r + 1 ≠ 0` (`succ` stands for successor).

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

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

Note that this example does not assume that `p` is prime, but the conclusion is trivial when `p` is not prime since `r.factorization p` is then zero by definition, and the proof works in all cases anyway.

```
example {m n k r : ℕ} (nnz : n ≠ 0) (pow_eq : m ^ k = r * n ^ k) {p : ℕ} :
    k ∣ r.factorization p := by
  rcases r with _ | r
  · simp
  have npow_nz : n ^ k ≠ 0 := fun npowz ↦ nnz (pow_eq_zero npowz)
  have eq1 : (m ^ k).factorization p = k * m.factorization p := by
    sorry
  have eq2 : ((r + 1) * n ^ k).factorization p =
      k * n.factorization p + (r + 1).factorization p := by
    sorry
  have : r.succ.factorization p = k * m.factorization p - k * n.factorization p := by
    rw [← eq1, pow_eq, eq2, add_comm, Nat.add_sub_cancel]
  rw [this]
  sorry
```

There are a number of ways in which we might want to improve on these results. To start with, a proof that the square

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

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

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

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

5.2 Induction and Recursion

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

```
inductive Nat where
  | zero : Nat
  | succ (n : Nat) : Nat
```

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

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

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

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

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

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

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

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

```
example : fac 0 = 1 :=
  rfl

example : fac 0 = 1 := by
  rw [fac]

example : fac 0 = 1 := by
  simp [fac]

example (n : ℕ) : fac (n + 1) = (n + 1) * fac n :=
  rfl

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

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

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

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

```
theorem fac_pos (n : ℕ) : 0 < fac n := by
  induction' n with n ih
  · rw [fac]
    exact zero_lt_one
  rw [fac]
  exact mul_pos n.succ_pos ih
```

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

```
theorem dvd_fac {i n : ℕ} (ipos : 0 < i) (ile : i ≤ n) : i ∣ fac n := by
  induction' n with n ih
  · exact absurd ipos (not_lt_of_ge ile)
  rw [fac]
  rcases Nat.of_le_succ ile with h | h
  · apply dvd_mul_of_dvd_right (ih h)
  rw [h]
  apply dvd_mul_right
```

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

```
theorem pow_two_le_fac (n : ℕ) : 2 ^ (n - 1) ≤ fac n := by
  rcases n with _ | n
  · simp [fac]
  sorry
```

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

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

```
variable {α : Type*} (s : Finset ℕ) (f : ℕ → ℕ) (n : ℕ)

#check Finset.sum s f
#check Finset.prod s f

open BigOperators
open Finset

example : s.sum f = Σ x ∈ s, f x :=
  rfl

example : s.prod f = Π x ∈ s, f x :=
  rfl

example : (range n).sum f = Σ x ∈ range n, f x :=
  rfl

example : (range n).prod f = Π x ∈ range n, f x :=
  rfl
```

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

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

example (f : ℕ → ℕ) (n : ℕ) : Σ x ∈ range n.succ, f x = Σ x ∈ range n, f x + f n :=
  Finset.sum_range_succ f n

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

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

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

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

```
example (n : ℕ) : fac n = Π i ∈ range n, (i + 1) := by
  induction' n with n ih
  · simp [fac, prod_range_zero]
  simp [fac, ih, prod_range_succ, mul_comm]
```

The fact that we include `mul_comm` as a simplification rule deserves comment. It should seem dangerous to simplify with the identity `x * y = y * x`, which would ordinarily loop indefinitely. Lean's simplifier is smart enough to recognize that, and applies the rule only in the case where the resulting term has a smaller value in some fixed but arbitrary ordering of the terms. The following example shows that simplifying using the three rules `mul_assoc`, `mul_comm`, and `mul_left_comm` manages to identify products that are the same up to the placement of parentheses and ordering of variables.

```
example (a b c d e f : ℕ) : a * (b * c * f * (d * e)) = d * (a * f * e) * (c * b) :=␣
↪by
  simp [mul_assoc, mul_comm, mul_left_comm]
```

Roughly, the rules work by pushing parentheses to the right and then re-ordering the expressions on both sides until they both follow the same canonical order. Simplifying with these rules, and the corresponding rules for addition, is a handy trick.

Returning to summation identities, we suggest stepping through the following proof that the sum of the natural numbers up to and including n is $n(n+1)/2$. The first step of the proof clears the denominator. This is generally useful when formalizing identities, because calculations with division generally have side conditions. (It is similarly useful to avoid using subtraction on the natural numbers when possible.)

```
theorem sum_id (n : ℕ) : Σ i ∈ range (n + 1), i = n * (n + 1) / 2 := by
  symm; apply Nat.div_eq_of_eq_mul_right (by norm_num : 0 < 2)
  induction' n with n ih
  · simp
  rw [Finset.sum_range_succ, mul_add 2, ← ih]
  ring
```

We encourage you to prove the analogous identity for sums of squares, and other identities you can find on the web.

```
theorem sum_sqr (n : ℕ) : Σ i ∈ range (n + 1), i ^ 2 = n * (n + 1) * (2 * n + 1) / 6␣
↪:= by
  sorry
```

In Lean's core library, addition and multiplication are themselves defined using recursive definitions, and their fundamental properties are established using induction. If you like thinking about foundational topics like that, you might enjoy working through proofs of the commutativity and associativity of multiplication and addition and the distributivity of multiplication over addition. You can do this on a copy of the natural numbers following the outline below. Notice that we can use the `induction` tactic with `MyNat`; Lean is smart enough to know to use the relevant induction principle (which is, of course, the same as that for `Nat`).

We start you off with the commutativity of addition. A good rule of thumb is that because addition and multiplication are defined by recursion on the second argument, it is generally advantageous to do proofs by induction on a variable that occurs in that position. It is a bit tricky to decide which variable to use in the proof of associativity.

It can be confusing to write things without the usual notation for zero, one, addition, and multiplication. We will learn how to define such notation later. Working in the namespace `MyNat` means that we can write `zero` and `succ` rather than `MyNat.zero` and `MyNat.succ`, and that these interpretations of the names take precedence over others. Outside the namespace, the full name of the `add` defined below, for example, is `MyNat.add`.

If you find that you *really* enjoy this sort of thing, try defining truncated subtraction and exponentiation and proving some of their properties as well. Remember that truncated subtraction cuts off at zero. To define that, it is useful to define a predecessor function, `pred`, that subtracts one from any nonzero number and fixes zero. The function `pred` can be defined by a simple instance of recursion.

```
inductive MyNat where
  | zero : MyNat
  | succ : MyNat → MyNat
```

(continues on next page)

(continued from previous page)

```
namespace MyNat

def add : MyNat → MyNat → MyNat
  | x, zero => x
  | x, succ y => succ (add x y)

def mul : MyNat → MyNat → MyNat
  | x, zero => zero
  | x, succ y => add (mul x y) x

theorem zero_add (n : MyNat) : add zero n = n := by
  induction' n with n ih
  · rfl
  rw [add, ih]

theorem succ_add (m n : MyNat) : add (succ m) n = succ (add m n) := by
  induction' n with n ih
  · rfl
  rw [add, ih]
  rfl

theorem add_comm (m n : MyNat) : add m n = add n m := by
  induction' n with n ih
  · rw [zero_add]
    rfl
  rw [add, succ_add, ih]

theorem add_assoc (m n k : MyNat) : add (add m n) k = add m (add n k) := by
  sorry
theorem mul_add (m n k : MyNat) : mul m (add n k) = add (mul m n) (mul m k) := by
  sorry
theorem zero_mul (n : MyNat) : mul zero n = zero := by
  sorry
theorem succ_mul (m n : MyNat) : mul (succ m) n = add (mul m n) n := by
  sorry
theorem mul_comm (m n : MyNat) : mul m n = mul n m := by
  sorry
end MyNat
```

5.3 Infinitely Many Primes

Let us continue our exploration of induction and recursion with another mathematical standard: a proof that there are infinitely many primes. One way to formulate this is as the statement that for every natural number n, there is a prime number greater than n. To prove this, let p be any prime factor of $n! + 1$. If p is less than or equal to n, it divides $n!$. Since it also divides $n! + 1$, it divides 1, a contradiction. Hence p is greater than n.

To formalize that proof, we need to show that any number greater than or equal to 2 has a prime factor. To do that, we will need to show that any natural number that is not equal to 0 or 1 is greater-than or equal to 2. And this brings us to a quirky feature of formalization: it is often trivial statements like this that are among the most annoying to formalize. Here we consider a few ways to do it.

To start with, we can use the `cases` tactic and the fact that the successor function respects the ordering on the natural numbers.

```
theorem two_le {m : ℕ} (h0 : m ≠ 0) (h1 : m ≠ 1) : 2 ≤ m := by
  cases m; contradiction
  case succ m =>
    cases m; contradiction
    repeat apply Nat.succ_le_succ
    apply zero_le
```

Another strategy is to use the tactic `interval_cases`, which automatically splits the goal into cases when the variable in question is contained in an interval of natural numbers or integers. Remember that you can hover over it to see its documentation.

```
example {m : ℕ} (h0 : m ≠ 0) (h1 : m ≠ 1) : 2 ≤ m := by
  by_contra h
  push_neg at h
  interval_cases m <;> contradiction
```

Recall that the semicolon after `interval_cases m` means that the next tactic is applied to each of the cases that it generates. Yet another option is to use the tactic `decide`, which tries to find a decision procedure to solve the problem. Lean knows that you can decide the truth value of a statement that begins with a bounded quantifier `∀ x, x < n →` `...` or `∃ x, x < n ∧ ...` by deciding each of the finitely many instances.

```
example {m : ℕ} (h0 : m ≠ 0) (h1 : m ≠ 1) : 2 ≤ m := by
  by_contra h
  push_neg at h
  revert h0 h1
  revert h m
  decide
```

With the theorem `two_le` in hand, let's start by showing that every natural number greater than two has a prime divisor. Mathlib contains a function `Nat.minFac` that returns the smallest prime divisor, but for the sake of learning new parts of the library, we'll avoid using it and prove the theorem directly.

Here, ordinary induction isn't enough. We want to use *strong induction*, which allows us to prove that every natural number n has a property P by showing that for every number n, if P holds of all values less than n, it holds at n as well. In Lean, this principle is called `Nat.strong_induction_on`, and we can use the `using` keyword to tell the induction tactic to use it. Notice that when we do that, there is no base case; it is subsumed by the general induction step.

The argument is simply as follows. Assuming $n \geq 2$, if n is prime, we're done. If it isn't, then by one of the characterizations of what it means to be a prime number, it has a nontrivial factor, m, and we can apply the inductive hypothesis to that. Step through the next proof to see how that plays out.

```
theorem exists_prime_factor {n : Nat} (h : 2 ≤ n) : ∃ p : Nat, p.Prime ∧ p ∣ n := by
  by_cases np : n.Prime
  · use n, np
  induction' n using Nat.strong_induction_on with n ih
  rw [Nat.prime_def_lt] at np
  push_neg at np
  rcases np h with ⟨m, mltn, mdvdn, mne1⟩
  have : m ≠ 0 := by
    intro mz
    rw [mz, zero_dvd_iff] at mdvdn
    linarith
  have mgt2 : 2 ≤ m := two_le this mne1
  by_cases mp : m.Prime
  · use m, mp
  · rcases ih m mltn mgt2 mp with ⟨p, pp, pdvd⟩
```

(continues on next page)

(continued from previous page)

```
    use p, pp
    apply pdvd.trans mdvdn
```

We can now prove the following formulation of our theorem. See if you can fill out the sketch. You can use `Nat.factorial_pos`, `Nat.dvd_factorial`, and `Nat.dvd_sub'`.

```
theorem primes_infinite : ∀ n, ∃ p > n, Nat.Prime p := by
  intro n
  have : 2 ≤ Nat.factorial n + 1 := by
    sorry
  rcases exists_prime_factor this with ⟨p, pp, pdvd⟩
  refine ⟨p, ?_, pp⟩
  show p > n
  by_contra ple
  push_neg at ple
  have : p ∣ Nat.factorial n := by
    sorry
  have : p ∣ 1 := by
    sorry
  show False
  sorry
```

Let's consider a variation of the proof above, where instead of using the factorial function, we suppose that we are given by a finite set $\{p_1, \ldots, p_n\}$ and we consider a prime factor of $\prod_{i=1}^{n} p_i + 1$. That prime factor has to be distinct from each p_i, showing that there is no finite set that contains all the prime numbers.

Formalizing this argument requires us to reason about finite sets. In Lean, for any type α, the type `Finset α` represents finite sets of elements of type α. Reasoning about finite sets computationally requires having a procedure to test equality on α, which is why the snippet below includes the assumption `[DecidableEq α]`. For concrete data types like $\mathbb{N}$, $\mathbb{Z}$, and $\mathbb{Q}$, the assumption is satisfied automatically. When reasoning about the real numbers, it can be satisfied using classical logic and abandoning the computational interpretation.

We use the command `open Finset` to avail ourselves of shorter names for the relevant theorems. Unlike the case with sets, most equivalences involving finsets do not hold definitionally, so they need to be expanded manually using equivalences like `Finset.subset_iff`, `Finset.mem_union`, `Finset.mem_inter`, and `Finset.mem_sdiff`. The `ext` tactic can still be used to show that two finite sets are equal by showing that every element of one is an element of the other.

```
open Finset

section
variable {α : Type*} [DecidableEq α] (r s t : Finset α)

example : r ∩ (s ∪ t) ⊆ r ∩ s ∪ r ∩ t := by
  rw [subset_iff]
  intro x
  rw [mem_inter, mem_union, mem_union, mem_inter, mem_inter]
  tauto

example : r ∩ (s ∪ t) ⊆ r ∩ s ∪ r ∩ t := by
  simp [subset_iff]
  intro x
  tauto

example : r ∩ s ∪ r ∩ t ⊆ r ∩ (s ∪ t) := by
  simp [subset_iff]
```

(continues on next page)

(continued from previous page)

```
  intro x
  tauto

example : r ∩ s ∪ r ∩ t = r ∩ (s ∪ t) := by
  ext x
  simp
  tauto

end
```

We have used a new trick: the `tauto` tactic (and a strengthened version, `tauto!`, which uses classical logic) can be used to dispense with propositional tautologies. See if you can use these methods to prove the two examples below.

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

The theorem `Finset.dvd_prod_of_mem` tells us that if an `n` is an element of a finite set `s`, then `n` divides `∏ i ∈ s, i`.

```
example (s : Finset ℕ) (n : ℕ) (h : n ∈ s) : n ∣ ∏ i ∈ s, i :=
  Finset.dvd_prod_of_mem _ h
```

We also need to know that the converse holds in the case where `n` is prime and `s` is a set of primes. To show that, we need the following lemma, which you should be able to prove using the theorem `Nat.Prime.eq_one_or_self_of_dvd`.

```
theorem _root_.Nat.Prime.eq_of_dvd_of_prime {p q : ℕ}
      (prime_p : Nat.Prime p) (prime_q : Nat.Prime q) (h : p ∣ q) :
    p = q := by
  sorry
```

We can use this lemma to show that if a prime `p` divides a product of a finite set of primes, then it is equal to one of them. Mathlib provides a useful principle of induction on finite sets: to show that a property holds of an arbitrary finite set `s`, show that it holds of the empty set, and show that it is preserved when we add a single new element `a ∉ s`. The principle is known as `Finset.induction_on`. When we tell the induction tactic to use it, we can also specify the names `a` and `s`, the name for the assumption `a ∉ s` in the inductive step, and the name of the inductive hypothesis. The expression `Finset.insert a s` denotes the union of `s` with the singleton `a`. The identities `Finset.prod_empty` and `Finset.prod_insert` then provide the relevant rewrite rules for the product. In the proof below, the first `simp` applies `Finset.prod_empty`. Step through the beginning of the proof to see the induction unfold, and then finish it off.

```
theorem mem_of_dvd_prod_primes {s : Finset ℕ} {p : ℕ} (prime_p : p.Prime) :
    (∀ n ∈ s, Nat.Prime n) → (p ∣ ∏ n ∈ s, n) → p ∈ s := by
  intro h₀ h₁
  induction' s using Finset.induction_on with a s ans ih
  · simp at h₁
    linarith [prime_p.two_le]
  simp [Finset.prod_insert ans, prime_p.dvd_mul] at h₀ h₁
  rw [mem_insert]
  sorry
```

We need one last property of finite sets. Given an element `s : Set α` and a predicate `P` on α, in Chapter 4 we wrote `{ x ∈ s | P x }` for the set of elements of `s` that satisfy `P`. Given `s : Finset α`, the analogous notion is written `s.filter P`.

```
example (s : Finset ℕ) (x : ℕ) : x ∈ s.filter Nat.Prime ↔ x ∈ s ∧ x.Prime :=
  mem_filter
```

We now prove an alternative formulation of the statement that there are infinitely many primes, namely, that given any `s : Finset ℕ`, there is a prime `p` that is not an element of `s`. Aiming for a contradiction, we assume that all the primes are in `s`, and then cut down to a set `s'` that contains all and only the primes. Taking the product of that set, adding one, and finding a prime factor of the result leads to the contradiction we are looking for. See if you can complete the sketch below. You can use `Finset.prod_pos` in the proof of the first `have`.

```
theorem primes_infinite' : ∀ s : Finset Nat, ∃ p, Nat.Prime p ∧ p ∉ s := by
  intro s
  by_contra h
  push_neg at h
  set s' := s.filter Nat.Prime with s'_def
  have mem_s' : ∀ {n : ℕ}, n ∈ s' ↔ n.Prime := by
    intro n
    simp [s'_def]
    apply h
  have : 2 ≤ (∏ i ∈ s', i) + 1 := by
    sorry
  rcases exists_prime_factor this with ⟨p, pp, pdvd⟩
  have : p ∣ ∏ i ∈ s', i := by
    sorry
  have : p ∣ 1 := by
    convert Nat.dvd_sub pdvd this
    simp
  show False
  sorry
```

We have thus seen two ways of saying that there are infinitely many primes: saying that they are not bounded by any `n`, and saying that they are not contained in any finite set `s`. The two proofs below show that these formulations are equivalent. In the second, in order to form `s.filter Q`, we have to assume that there is a procedure for deciding whether or not `Q` holds. Lean knows that there is a procedure for `Nat.Prime`. In general, if we use classical logic by writing `open Classical`, we can dispense with the assumption.

In Mathlib, `Finset.sup s f` denotes the supremum of the values of `f x` as `x` ranges over `s`, returning `0` in the case where `s` is empty and the codomain of `f` is ℕ. In the first proof, we use `s.sup id`, where `id` is the identity function, to refer to the maximum value in `s`.

```
theorem bounded_of_ex_finset (Q : ℕ → Prop) :
    (∃ s : Finset ℕ, ∀ k, Q k → k ∈ s) → ∃ n, ∀ k, Q k → k < n := by
  rintro ⟨s, hs⟩
  use s.sup id + 1
  intro k Qk
  apply Nat.lt_succ_of_le
  show id k ≤ s.sup id
  apply le_sup (hs k Qk)

theorem ex_finset_of_bounded (Q : ℕ → Prop) [DecidablePred Q] :
    (∃ n, ∀ k, Q k → k ≤ n) → ∃ s : Finset ℕ, ∀ k, Q k ↔ k ∈ s := by
  rintro ⟨n, hn⟩
  use (range (n + 1)).filter Q
  intro k
  simp [Nat.lt_succ_iff]
  exact hn k
```

A small variation on our second proof that there are infinitely many primes shows that there are infinitely many primes

congruent to 3 modulo 4. The argument goes as follows. First, notice that if the product of two numbers m and n is equal to 3 modulo 4, then one of the two numbers is congruent to 3 modulo 4. After all, both have to be odd, and if they are both congruent to 1 modulo 4, so is their product. We can use this observation to show that if some number greater than 2 is congruent to 3 modulo 4, then that number has a prime divisor that is also congruent to 3 modulo 4.

Now suppose there are only finitely many prime numbers congruent to 3 modulo 4, say, $p_1, \ldots, p_k$. Without loss of generality, we can assume that $p_1 = 3$. Consider the product $4 \prod_{i=2}^{k} p_i + 3$. It is easy to see that this is congruent to 3 modulo 4, so it has a prime factor p congruent to 3 modulo 4. It can't be the case that $p = 3$; since p divides $4 \prod_{i=2}^{k} p_i + 3$, if p were equal to 3 then it would also divide $\prod_{i=2}^{k} p_i$, which implies that p is equal to one of the p_i for $i = 2, \ldots, k$; and we have excluded 3 from this list. So p has to be one of the other elements p_i. But in that case, p divides $4 \prod_{i=2}^{k} p_i$ and hence 3, which contradicts the fact that it is not 3.

In Lean, the notation `n % m`, read "`n` modulo `m`," denotes the remainder of the division of `n` by `m`.

```
example : 27 % 4 = 3 := by norm_num
```

We can then render the statement "`n` is congruent to 3 modulo 4" as `n % 4 = 3`. The following example and theorems sum up the facts about this function that we will need to use below. The first named theorem is another illustration of reasoning by a small number of cases. In the second named theorem, remember that the semicolon means that the subsequent tactic block is applied to all the goals created by the preceding tactic.

```
example (n : ℕ) : (4 * n + 3) % 4 = 3 := by
  rw [add_comm, Nat.add_mul_mod_self_left]

theorem mod_4_eq_3_or_mod_4_eq_3 {m n : ℕ} (h : m * n % 4 = 3) : m % 4 = 3 ∨ n % 4 =␣
↪3 := by
  revert h
  rw [Nat.mul_mod]
  have : m % 4 < 4 := Nat.mod_lt m (by norm_num)
  interval_cases m % 4 <;> simp [-Nat.mul_mod_mod]
  have : n % 4 < 4 := Nat.mod_lt n (by norm_num)
  interval_cases n % 4 <;> simp

theorem two_le_of_mod_4_eq_3 {n : ℕ} (h : n % 4 = 3) : 2 ≤ n := by
  apply two_le <;>
    · intro neq
      rw [neq] at h
      norm_num at h
```

We will also need the following fact, which says that if `m` is a nontrivial divisor of `n`, then so is `n / m`. See if you can complete the proof using `Nat.div_dvd_of_dvd` and `Nat.div_lt_self`.

```
theorem aux {m n : ℕ} (h₀ : m ∣ n) (h₁ : 2 ≤ m) (h₂ : m < n) : n / m ∣ n ∧ n / m < n :=␣
↪by
  sorry
```

Now put all the pieces together to prove that any number congruent to 3 modulo 4 has a prime divisor with that same property.

```
theorem exists_prime_factor_mod_4_eq_3 {n : Nat} (h : n % 4 = 3) :
    ∃ p : Nat, p.Prime ∧ p ∣ n ∧ p % 4 = 3 := by
  by_cases np : n.Prime
  · use n
  induction' n using Nat.strong_induction_on with n ih
  rw [Nat.prime_def_lt] at np
  push_neg at np
  rcases np (two_le_of_mod_4_eq_3 h) with ⟨m, mltn, mdvdn, mne1⟩
```

(continues on next page)

(continued from previous page)

```
  have mge2 : 2 ≤ m := by
    apply two_le _ mne1
    intro mz
    rw [mz, zero_dvd_iff] at mdvdn
    linarith
  have neq : m * (n / m) = n := Nat.mul_div_cancel' mdvdn
  have : m % 4 = 3 ∨ n / m % 4 = 3 := by
    apply mod_4_eq_3_or_mod_4_eq_3
    rw [neq, h]
  rcases this with h1 | h1
  . sorry
  . sorry
```

We are in the home stretch. Given a set `s` of prime numbers, we need to talk about the result of removing 3 from that set, if it is present. The function `Finset.erase` handles that.

```
example (m n : ℕ) (s : Finset ℕ) (h : m ∈ erase s n) : m ≠ n ∧ m ∈ s := by
  rwa [mem_erase] at h

example (m n : ℕ) (s : Finset ℕ) (h : m ∈ erase s n) : m ≠ n ∧ m ∈ s := by
  simp at h
  assumption
```

We are now ready to prove that there are infinitely many primes congruent to 3 modulo 4. Fill in the missing parts below. Our solution uses `Nat.dvd_add_iff_left` and `Nat.dvd_sub'` along the way.

```
theorem primes_mod_4_eq_3_infinite : ∀ n, ∃ p > n, Nat.Prime p ∧ p % 4 = 3 := by
  by_contra h
  push_neg at h
  rcases h with ⟨n, hn⟩
  have : ∃ s : Finset Nat, ∀ p : ℕ, p.Prime ∧ p % 4 = 3 ↔ p ∈ s := by
    apply ex_finset_of_bounded
    use n
    contrapose! hn
    rcases hn with ⟨p, ⟨pp, p4⟩, pltn⟩
    exact ⟨p, pltn, pp, p4⟩
  rcases this with ⟨s, hs⟩
  have h₁ : ((4 * ∏ i ∈ erase s 3, i) + 3) % 4 = 3 := by
    sorry
  rcases exists_prime_factor_mod_4_eq_3 h₁ with ⟨p, pp, pdvd, p4eq⟩
  have ps : p ∈ s := by
    sorry
  have pne3 : p ≠ 3 := by
    sorry
  have : p ∣ 4 * ∏ i ∈ erase s 3, i := by
    sorry
  have : p ∣ 3 := by
    sorry
  have : p = 3 := by
    sorry
  contradiction
```

If you managed to complete the proof, congratulations! This has been a serious feat of formalization.

CHAPTER

SIX

DISCRETE MATHEMATICS

Discrete Mathematics is the study of finite sets, objects, and structures. We can count the elements of a finite set, and we can compute finite sums or products over its elements, we can compute maximums and minimums, and so on. We can also study objects that are generated by finitely many applications of certain generating functions, we can define functions by structural recursion, and prove theorems by structural induction. This chapters describes parts of Mathlib that support these activities.

6.1 More Induction

In Section 5.2, we saw how to define the factorial function by recursion on the natural numbers.

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

We also saw how to prove theorems using the `induction'` tactic.

```
theorem fac_pos (n : ℕ) : 0 < fac n := by
  induction' n with n ih
  · rw [fac]
    exact zero_lt_one
  rw [fac]
  exact mul_pos n.succ_pos ih
```

The `induction` tactic (without the prime tick mark) allows for more structured syntax.

```
example (n : ℕ) : 0 < fac n := by
  induction n
  case zero =>
    rw [fac]
    exact zero_lt_one
  case succ n ih =>
    rw [fac]
    exact mul_pos n.succ_pos ih

example (n : ℕ) : 0 < fac n := by
  induction n with
  | zero =>
    rw [fac]
    exact zero_lt_one
  | succ n ih =>
    rw [fac]
    exact mul_pos n.succ_pos ih
```

The names of the cases, `zero` and `succ`, are taken from the definition of the induction principle. Notice that the `succ` case allows you to choose whatever names you want for the induction variable and the inductive hypothesis, here `n` and `ih`. You can even prove a theorem with the same notation used to define a recursive function.

```
theorem fac_pos' : ∀ n, 0 < fac n
  | 0 => by
    rw [fac]
    exact zero_lt_one
  | n + 1 => by
    rw [fac]
    exact mul_pos n.succ_pos (fac_pos' n)
```

Notice also the absence of the `:=`, the `∀ n` after the colon, the `by` keyword in each case, and the inductive appeal to `fac_pos' n`. It is as though the theorem is a recursive function of `n` and in the inductive step we make a recursive call.

This style of definition is remarkably flexible. Lean's designers have built in elaborate means of defining recursive functions, and these extend to doing proofs by induction. For example, we can define the Fibonacci function with multiple base cases.

```
@[simp] def fib : ℕ → ℕ
  | 0 => 0
  | 1 => 1
  | n + 2 => fib n + fib (n + 1)
```

The `@[simp]` annotation means that the simplifier will use the defining equations. You can also apply them by writing `rw [fib]`. Below it will be helpful to give a name to the `n + 2` case.

```
theorem fib_add_two (n : ℕ) : fib (n + 2) = fib n + fib (n + 1) := rfl

example (n : ℕ) : fib (n + 2) = fib n + fib (n + 1) := by rw [fib]
```

Using Lean's notation for recursive functions, you can carry out proofs by induction on the natural numbers that mirror the recursive definition of `fib`. The following example provides an explicit formula for the nth Fibonacci number in terms of the golden mean, φ, and its conjugate, φ'. We have to tell Lean that we don't expect our definitions to generate code because the arithmetic operations on the real numbers are not computable.

```
noncomputable section

def phi  : ℝ := (1 + √5) / 2
def phi' : ℝ := (1 - √5) / 2

theorem phi_sq : phi^2 = phi + 1 := by
  field_simp [phi, add_sq]; ring

theorem phi'_sq : phi'^2 = phi' + 1 := by
  field_simp [phi', sub_sq]; ring

theorem fib_eq : ∀ n, fib n = (phi^n - phi'^n) / √5
  | 0 => by simp
  | 1 => by field_simp [phi, phi']
  | n+2 => by
    field_simp [fib_eq, pow_add, phi_sq, phi'_sq]
    ring

end
```

Induction proofs involving the Fibonacci function do not have to be of that form. Below we reproduce the `Mathlib` proof that consecutive Fibonacci numbers are coprime.

```
theorem fib_coprime_fib_succ (n : ℕ) : Nat.Coprime (fib n) (fib (n + 1)) := by
  induction n with
  | zero => simp
  | succ n ih =>
    simp only [fib, Nat.coprime_add_self_right]
    exact ih.symm
```

Using Lean's computational interpretation, we can evaluate the Fibonacci numbers.

```
#eval fib 6
#eval List.range 20 |>.map fib
```

The straightforward implementation of `fib` is computationally inefficient. In fact, it runs in time exponential in its argument. (You should think about why.) In Lean, we can implement the following tail-recursive version, whose running time is linear in `n`, and prove that it computes the same function.

```
def fib' (n : Nat) : Nat :=
  aux n 0 1
where aux
  | 0, x, _ => x
  | n+1, x, y => aux n y (x + y)

theorem fib'.aux_eq (m n : ℕ) : fib'.aux n (fib m) (fib (m + 1)) = fib (n + m) := by
  induction n generalizing m with
  | zero => simp [fib'.aux]
  | succ n ih => rw [fib'.aux, ←fib_add_two, ih, add_assoc, add_comm 1]

theorem fib'_eq_fib : fib' = fib := by
  ext n
  erw [fib', fib'.aux_eq 0 n]; rfl

#eval fib' 10000
```

Notice the `generalizing` keyword in the proof of `fib'.aux_eq`. It serves to insert a `∀ m` in front of the inductive hypothesis, so that in the induction step, `m` can take a different value. You can step through the proof and check that in this case, `m` needs to be instantiated to `m + 1`. As usual, you can hover over the `induction` keyword to read the documentation.

Notice also the use of `erw` (for "extended rewrite") instead of `rw`. This is used because to rewrite the goal `fib'.aux_eq`, `fib 0` and `fib 1` have to be reduced to `0` and `1`, respectively. The tactic `erw` is simply more aggressive than `rw` in unfolding definitions to match parameters. This isn't always a good idea; it can waste a lot of time in some cases, so use `erw` sparingly.

Here is another example of the `generalizing` keyword in use, in the proof of another identity that is found in `Mathlib`. An informal proof of the identity can be found here. We provide two variants of the formal proof.

```
theorem fib_add (m n : ℕ) : fib (m + n + 1) = fib m * fib n + fib (m + 1) * fib (n +␣
↪1) := by
  induction n generalizing m with
  | zero => simp
  | succ n ih =>
    specialize ih (m + 1)
    rw [add_assoc m 1 n, add_comm 1 n] at ih
    simp only [fib_add_two, Nat.succ_eq_add_one, ih]
    ring

theorem fib_add' : ∀ m n, fib (m + n + 1) = fib m * fib n + fib (m + 1) * fib (n + 1)
```

(continues on next page)

(continued from previous page)

```
  | _, 0     => by simp
  | m, n + 1 => by
    have := fib_add' (m + 1) n
    rw [add_assoc m 1 n, add_comm 1 n] at this
    simp only [fib_add_two, Nat.succ_eq_add_one, this]
    ring
```

As an exercise, use `fib_add` to prove the following.

```
example (n : ℕ): (fib n)^2 + (fib (n + 1))^2 = fib (2 * n + 1) := by sorry
```

Lean's mechanisms for defining recursive functions are flexible enough to allow arbitrary recursive calls, as long the complexity of the arguments decrease according to some well-founded measure. In the next example, we show that every natural number `n ≠ 1` has a prime divisor, using the fact that if `n` is itself nonzero and not prime, it has a smaller divisor. (You can check that Mathlib has a theorem of the same name in the `Nat` namespace, though it has a different proof than the one we give here.)

```
#check (@Nat.not_prime_iff_exists_dvd_lt :
  ∀ {n : ℕ}, 2 ≤ n → (¬Nat.Prime n ↔ ∃ m, m ∣ n ∧ 2 ≤ m ∧ m < n))

theorem ne_one_iff_exists_prime_dvd : ∀ {n}, n ≠ 1 ↔ ∃ p : ℕ, p.Prime ∧ p ∣ n
  | 0 => by simpa using Exists.intro 2 Nat.prime_two
  | 1 => by simp [Nat.not_prime_one]
  | n + 2 => by
    have hn : n+2 ≠ 1 := by omega
    simp only [Ne, not_false_iff, true_iff, hn]
    by_cases h : Nat.Prime (n + 2)
    · use n+2, h
    · have : 2 ≤ n + 2 := by omega
      rw [Nat.not_prime_iff_exists_dvd_lt this] at h
      rcases h with ⟨m, mdvdn, mge2, -⟩
      have : m ≠ 1 := by omega
      rw [ne_one_iff_exists_prime_dvd] at this
      rcases this with ⟨p, primep, pdvdm⟩
      use p, primep
      exact pdvdm.trans mdvdn
```

The line `rw [ne_one_iff_exists_prime_dvd] at this` is like a magic trick: we are using the very theorem we are proving in its own proof. What makes it work is that the inductive call is instantiated at `m`, the current case is `n + 2`, and the context has `m < n + 2`. Lean can find the hypothesis and use it to show that the induction is well-founded. Lean is pretty good at figuring out what is decreasing; in this case, the choice of `n` in the statement of the theorem and the less-than relation is obvious. In more complicated cases, Lean provides mechanisms to provide this information explicitly. See the section on well-founded recursion in the Lean Reference Manual.

Sometimes, in a proof, you need to split on cases depending on whether a natural number `n` is zero or a successor, without requiring an inductive hypothesis in the successor case. For that, you can use the `cases` and `rcases` tactics.

```
theorem zero_lt_of_mul_eq_one (m n : ℕ) : n*m = 1 → 0 < n ∧ 0 < m := by
  cases n <;> cases m <;> simp

example (m n : ℕ) : n*m = 1 → 0 < n ∧ 0 < m := by
  rcases m with (_ | m); simp
  rcases n with (_ | n) <;> simp
```

This is a useful trick. Often you have a theorem about a natural number `n` for which the zero case is easy. If you case on `n` and take care of the zero case quickly, you are left with the original goal with `n` replaced by `n + 1`.

6.2 Finsets and Fintypes

Dealing with finite sets and types in Mathlib can be confusing, because the library offers multiple ways of handling them. In this section we will discuss the most common ones.

We have already come across the type `Finset` in Section 5.2 and Section 5.3. As the name suggests, an element of type `Finset α` is a finite set of elements of type α. We will these "finsets." The `Finset` data type is designed to have a computational interpretation, and many basic operations on `Finset α` assume that α has decidable equality, which guarantees that there is an algorithm for testing whether `a : α` is an element of a finset `s`.

```
section
variable {α : Type*} [DecidableEq α] (a : α) (s t : Finset α)

#check a ∈ s
#check s ∩ t

end
```

If you remove the declaration `[DecidableEq α]`, Lean will complain on the line `#check s ∩ t` because it cannot compute the intersection. All of the data types that you should expect to be able to compute with have decidable equality, however, and if you work classically by opening the `Classical` namespace and declaring `noncomputable section`, you can reason about finsets of elements of any type at all.

Finsets support most of the set-theoretic operations that sets do:

```
open Finset

variable (a b c : Finset ℕ)
variable (n : ℕ)

#check a ∩ b
#check a ∪ b
#check a \ b
#check (∅ : Finset ℕ)

example : a ∩ (b ∪ c) = (a ∩ b) ∪ (a ∩ c) := by
  ext x; simp only [mem_inter, mem_union]; tauto

example : a ∩ (b ∪ c) = (a ∩ b) ∪ (a ∩ c) := by rw [inter_union_distrib_left]
```

Note that we have opened the `Finset` namespace, where theorems specific to finsets are found. If you step through the last example below, you will see applying `ext` followed by `simp` reduces the identity to a problem in propositional logic. As an exercise, you can try proving some of set identities from Chapter 4, transported to finsets.

You have already seen the notation `Finset.range n` for the finite set of natural numbers $\{0, 1, \ldots, n-1\}$. `Finset` also allows you to define finite sets by enumerating the elements:

```
#check ({0, 2, 5} : Finset Nat)

def example1 : Finset ℕ := {0, 1, 2}
```

There are various ways to get Lean to recognize that order of elements and duplicates do not matter in a set presented in this way.

```
example : ({0, 1, 2} : Finset ℕ) = {1, 2, 0} := by decide

example : ({0, 1, 2} : Finset ℕ) = {0, 1, 1, 2} := by decide
```

(continues on next page)

(continued from previous page)

```
example : ({0, 1} : Finset ℕ) = {1, 0} := by rw [Finset.pair_comm]

example (x : Nat) : ({x, x} : Finset ℕ) = {x} := by simp

example (x y z : Nat) : ({x, y, z, y, z, x} : Finset ℕ) = {x, y, z} := by
  ext i; simp [or_comm, or_assoc, or_left_comm]

example (x y z : Nat) : ({x, y, z, y, z, x} : Finset ℕ) = {x, y, z} := by
  ext i; simp; tauto
```

You can use `insert` to add a single element to a Finset, and `Finset.erase` to delete a single element. Note that `erase` is in the `Finset` namespace, but `insert` is in the root namespace.

```
example (s : Finset ℕ) (a : ℕ) (h : a ∉ s) : (insert a s |>.erase a) = s :=
  Finset.erase_insert h

example (s : Finset ℕ) (a : ℕ) (h : a ∈ s) : insert a (s.erase a) = s :=
  Finset.insert_erase h
```

In fact, `{0, 1, 2}` is just notation for `insert 0 (insert 1 (singleton 2))`.

```
set_option pp.notation false in
#check ({0, 1, 2} : Finset ℕ)
```

Given a finset `s` and a predicate `P`, we can use set-builder notation `{x ∈ s | P x}` to define the set of elements of `s` that satisfy `P`. This is notation for `Finset.filter P s`, which can also be written `s.filter P`.

```
example : {m ∈ range n | Even m} = (range n).filter Even := rfl
example : {m ∈ range n | Even m ∧ m ≠ 3} = (range n).filter (fun m ↦ Even m ∧ m ≠ 3)␣
↪:= rfl

example : {m ∈ range 10 | Even m} = {0, 2, 4, 6, 8} := by decide
```

Mathlib knows that the image of a finset under a function is a finset.

```
#check (range 5).image (fun x ↦ x * 2)

example : (range 5).image (fun x ↦ x * 2) = {x ∈ range 10 | Even x} := by decide
```

Lean also knows that the cartesian product `s ×ˢ t` of two finsets is a finset, and that the powerset of a finset is a finset. (Note that the notation `s ×ˢ t` also works for sets.)

```
#check s ×ˢ t
#check s.powerset
```

Defining operations on finsets in terms of their elements is tricky, because any such definition has to be independent of the order in which the elements are presented. Of course, you can always define functions by composing existing operations. Another thing you can do is use `Finset.fold` to fold a binary operation over the elements, provided that the operation is associative and commutative, since these properties guarantee that the result is independent of the order that the operation is applied. Finite sums, products, and unions are defined in that way. In the last example below, `biUnion` stands for "bounded indexed union." With conventional mathematical notation, the expression would be written $\bigcup_{i \in s} g(i)$.

```
#check Finset.fold

def f (n : ℕ) : Int := (↑n)^2
```

(continues on next page)

(continued from previous page)

```
#check (range 5).fold (fun x y : Int ↦ x + y) 0 f
#eval (range 5).fold (fun x y : Int ↦ x + y) 0 f

#check Σ i ∈ range 5, i^2
#check Π i ∈ range 5, i + 1

variable (g : Nat → Finset Int)

#check (range 5).biUnion g
```

There is a natural principle of induction on finsets: to prove that every finset has a property, show that the empty set has the property and that the property is preserved when we add one new element to a finset. (The `@` symbol in `@insert` is needed in the induction step of the next example to give names to the parameters `a` and `s` because they have been marked implicit.)

```
#check Finset.induction

example {α : Type*} [DecidableEq α] (f : α → ℕ) (s : Finset α) (h : ∀ x ∈ s, f x ≠␣
↪0) :
    Π x ∈ s, f x ≠ 0 := by
  induction s using Finset.induction_on with
  | empty => simp
  | @insert a s anins ih =>
      rw [prod_insert anins]
      apply mul_ne_zero
      · apply h; apply mem_insert_self
      apply ih
      intros x xs
      exact h x (mem_insert_of_mem xs)
```

If `s` is a finset, `Finset.Nonempty s` is defined to be `∃ x, x ∈ s`. You can use classical choice to pick an element of a nonempty finset. Similarly, the library defines `Finset.toList s` which uses choice to pick the elements of `s` in some order.

```
noncomputable example (s : Finset ℕ) (h : s.Nonempty) : ℕ := Classical.choose h

example (s : Finset ℕ) (h : s.Nonempty) : Classical.choose h ∈ s := Classical.choose_
↪spec h

noncomputable example (s : Finset ℕ) : List ℕ := s.toList

example (s : Finset ℕ) (a : ℕ) : a ∈ s.toList ↔ a ∈ s := mem_toList
```

You can use `Finset.min` and `Finset.max` to choose the minimum or maximum element of a finset of elements of a linear order, and similarly you can use `Finset.inf` and `Finset.sup` with finsets of elements of a lattice, but there is a catch. What should the minimum element of an empty finset be? You can check that the primed versions of the functions below add a precondition that the finset is nonempty. The non-primed versions `Finset.min` and `Finset.max` add a top or bottom element, respectively, to the output type, to handle the case where the finset is empty. The non-primed versions `Finset.inf` and `Finset.sup` assume that the lattice comes equipped with a top or bottom element, respectively.

```
#check Finset.min
#check Finset.min'
#check Finset.max
#check Finset.max'
```

(continues on next page)

(continued from previous page)

```
#check Finset.inf
#check Finset.inf'
#check Finset.sup
#check Finset.sup'

example : Finset.Nonempty {2, 6, 7} := ⟨6, by trivial⟩
example : Finset.min' {2, 6, 7} ⟨6, by trivial⟩ = 2 := by trivial
```

Every finset `s` has a finite cardinality, `Finset.card s`, which can be written `#s` when the `Finset` namespace is open.

```
#check Finset.card

#eval (range 5).card

example (s : Finset ℕ) : s.card = #s := by rfl

example (s : Finset ℕ) : s.card = Σ i ∈ s, 1 := by rw [card_eq_sum_ones]

example (s : Finset ℕ) : s.card = Σ i ∈ s, 1 := by simp
```

The next section is all about reasoning about cardinality.

When formalizing mathematics, one often has to make a decision as to whether to express one's definitions and theorems in terms of sets or types. Using types often simplifies notation and proofs, but working with subsets of a type can be more flexible. The type-based analogue of a finset is a *fintype*, that is, a type `Fintype α` for some α. By definition, a fintype is just a data type that comes equipped with a finset `univ` that contains all its elements.

```
variable {α : Type*} [Fintype α]

example : ∀ x : α, x ∈ Finset.univ := by
  intro x; exact mem_univ x
```

`Fintype.card α` is equal to the cardinality of the corresponding finset.

```
example : Fintype.card α = (Finset.univ : Finset α).card := rfl
```

We have already seen a prototypical example of a fintype, namely, the types `Fin n` for each `n`. Lean recognizes that the fintypes are closed under operations like the product operation.

```
example : Fintype.card (Fin 5) = 5 := by simp
example : Fintype.card ((Fin 5) × (Fin 3)) = 15 := by simp
```

Any element `s` of `Finset α` can be coercied to a type `(↑s : Finset α)`, namely, the subtype of elements of α that are contained in `s`.

```
variable (s : Finset ℕ)

example : (↑s : Type) = {x : ℕ // x ∈ s} := rfl
example : Fintype.card ↑s = s.card := by simp
```

Lean and Mathlib use *type class inference* to track the additional structure on fintypes, namely, the universal finset that contains all the elements. In other words, you can think of a fintype as an algebraic structure equipped with that extra data. Chapter 7 explains how this works.

6.3 Counting Arguments

The art of counting things is a central part of combinatorics. Mathlib contains several basic identities for counting elements of finsets:

```
open Finset

variable {α β : Type*} [DecidableEq α] [DecidableEq β] (s t : Finset α) (f : α → β)

example : #(s ×ˢ t) = #s * #t := by simp

example : #(s ∪ t) = #s + #t - #(s ∩ t) := card_union _ _

example (h : Disjoint s t) : #(s ∪ t) = #s + #t := by simp [h]

example (h : Function.Injective f) : #(s.image f) = #s := by rw [card_image_of_
↪injective _ h]

example (h : Set.InjOn f s) : #(s.image f) = #s := by rw [card_image_of_injOn h]
```

Opening the `Finset` namespace allows us to use the notation `#s` for `s.card`, as well as to use the shortened names *card_union* and so on.

Mathlib can also count elements of fintypes:

```
open Fintype

variable {α β : Type*} [Fintype α] [Fintype β]

example : card (α × β) = card α * card β := by simp

example : card (α ⊕ β) = card α + card β := by simp

example (n : ℕ) : card (Fin n → α) = (card α)^n := by simp

variable {n : ℕ} {γ : Fin n → Type*} [∀ i, Fintype (γ i)]

example : card ((i : Fin n) → γ i) = Π i, card (γ i) := by simp

example : card (Σ i, γ i) = Σ i, card (γ i) := by simp
```

When the `Fintype` namespace is not open, we have to use `Fintype.card` instead of *card*.

The following is an example of calculating the cardinality of a finset, namely, *range n* together with a copy of range *n* shifted by more than *n*. The calculation requires showing the the two sets in the union are disjoint; the first line of the proof yields the side condition `Disjoint (range n) (image (fun i ↦ m + i) (range n))`, which is established at the end of the proof. The `Disjoint` predicate is too general to be directly useful to us, but the theorem `disjoint_iff_ne` puts it in a form we can use.

```
#check Disjoint

example (m n : ℕ) (h : m ≥ n) :
    card (range n ∪ (range n).image (fun i ↦ m + i)) = 2 * n := by
  rw [card_union_of_disjoint, card_range, card_image_of_injective, card_range]; omega
  . apply add_right_injective
  . simp [disjoint_iff_ne]; omega
```

Throughout this section, `omega` will be a workhorse for us, for dealing with arithmetic calculations and inequalities.

Here is a more interesting example. Consider the subset of $\{0, \ldots, n\} \times \{0, \ldots, n\}$ consisting of pairs (i, j) such that $i < j$. If you think of these as lattice points in the coordinate plane, they constitute an upper triangle of the square with corners $(0, 0)$ and (n, n), not including the diagonal. The cardinality of the full square is $(n+1)^2$, and removing the size of the diagonal and halving the result shows us that the cardinality of the triangle is $n(n+1)/2$.

Alternatively, we note that the rows of the triangle have sizes $0, 1, \ldots, n$, so the cardinality is the sum of the first n natural numbers. The first `have` of the proof below describes the triangle as the union of the rows, where row j consists of the numbers $0, 1, ..., j-1$ paired with j. In the proof below, the notation `(., j)` abbreviates the function `fun i ↦ (i, j)`. The rest of the proof is just a calculation with finset cardinalities.

```
def triangle (n : ℕ) : Finset (ℕ × ℕ) := {p ∈ range (n+1) ×ˢ range (n+1) | p.1 < p.2}

example (n : ℕ) : #(triangle n) = (n + 1) * n / 2 := by
  have : triangle n = (range (n+1)).biUnion (fun j ↦ (range j).image (., j)) := by
    ext p
    simp only [triangle, mem_filter, mem_product, mem_range, mem_biUnion, mem_image]
    constructor
    . rintro ⟨⟨hp1, hp2⟩, hp3⟩
      use p.2, hp2, p.1, hp3
    . rintro ⟨p1, hp1, p2, hp2, rfl⟩
      omega
  rw [this, card_biUnion]; swap
  · -- take care of disjointness first
    intro x _ y _ xney
    simp [disjoint_iff_ne, xney]
  -- continue the calculation
  transitivity (Σ i in range (n+1), i)
  · congr; ext i
    rw [card_image_of_injective, card_range]
    intros i1 i2; simp
  rw [sum_range_id]; rfl
```

The following variation on the proof does the calculation with fintypes instead of finsets. The type $\alpha \simeq \beta$ is the type of equivalences between α and β, consisting of a map in the forward direction, the map in the backward direction, and proofs that these two are inverse to one another. The first `have` in the proof shows that `triangle n` is equivalent to the disjoint union of `Fin i` as `i` ranges over `Fin (n + 1)`. Interestingly, the forward function and the reverse function are constructed with tactics, rather than written explicitly. Since they do nothing more than move data and information around, `rfl` establishes that they are inverses.

After that, `rw [←Fintype.card_coe]` rewrites `#(triangle n)` as the cardinality of the subtype `{ x // x ∈ triangle n }`, and the rest of the proof is a calculation.

```
example (n : ℕ) : #(triangle n) = (n + 1) * n / 2 := by
  have : triangle n ≃ Σ i : Fin (n + 1), Fin i.val :=
    { toFun := by
        rintro ⟨⟨i, j⟩, hp⟩
        simp [triangle] at hp
        exact ⟨⟨j, hp.1.2⟩, ⟨i, hp.2⟩⟩
      invFun := by
        rintro ⟨i, j⟩
        use ⟨j, i⟩
        simp [triangle]
        exact j.isLt.trans i.isLt
      left_inv := by intro i; rfl
      right_inv := by intro i; rfl }
  rw [←Fintype.card_coe]
  trans; apply (Fintype.card_congr this)
  rw [Fintype.card_sigma, sum_fin_eq_sum_range]
```

(continues on next page)

(continued from previous page)

```
  convert Finset.sum_range_id (n + 1)
  simp_all
```

Here is yet another approach. The first line of the proof below reduces the problem to showing `2 * #(triangle n) = (n + 1) * n`. We can do that by showing that two copies of the triangle exactly fill the rectangle `range n ×ˢ range (n + 1)`. As an exercise, see if you can fill in the steps of the calculation. In the solutions, we rely on `omega` extensively in the second-to-last step, but we unfortunately have to do a fair amount of work by hand.

```
example (n : ℕ) : #(triangle n) = (n + 1) * n / 2 := by
  apply Nat.eq_div_of_mul_eq_right (by norm_num)
  let turn (p : ℕ × ℕ) : ℕ × ℕ := (n - 1 - p.1, n - p.2)
  calc 2 * #(triangle n)
      = #(triangle n) + #(triangle n) := by
          sorry
    _ = #(triangle n) + #(triangle n |>.image turn) := by
          sorry
    _ = #(range n ×ˢ range (n + 1)) := by
          sorry
    _ = (n + 1) * n := by
          sorry
```

You can convince yourself that we get the same triangle, shifted down, if we replace `n` by `n + 1` and replace `<` by `≤` in the definition of `triangle`. The exercise below asks you to use this fact to show that the two triangles have the same size.

```
def triangle' (n : ℕ) : Finset (ℕ × ℕ) := {p ∈ range n ×ˢ range n | p.1 ≤ p.2}

example (n : ℕ) : #(triangle' n) = #(triangle n) := by sorry
```

Let us close this section with an example and an exercise from a tutorial on combinatorics given by Bhavik Mehta at *Lean for the Curious Mathematician* in 2023. Suppose we have a bipartite graph with vertex sets `s` and `t`, such that for every `a` in `s`, there are at least three edges leaving `a`, and for every `b` in `t`, there is at most one edge entering `b`. Then the total number of edges in the graph is at least three times the cardinality of `s` and at most the cardinality of `t`, from which is follows that the cardinality of `t` is at least three times the cardinality of `s`. The following theorem implements this argument, where we use the relation `r` to represent the edges of the graph. The proof is an elegant calculation.

```
open Classical
variable (s t : Finset Nat) (a b : Nat)

theorem doubleCounting {α β : Type*} (s : Finset α) (t : Finset β)
    (r : α → β → Prop)
    (h_left : ∀ a ∈ s, 3 ≤ #{b ∈ t | r a b})
    (h_right : ∀ b ∈ t, #{a ∈ s | r a b} ≤ 1) :
    3 * #(s) ≤ #(t) := by
  calc
    3 * #(s) = Σ a ∈ s, 3 := by simp [sum_const_nat, mul_comm]
    _ ≤ Σ a ∈ s, #({b ∈ t | r a b}) := sum_le_sum h_left
    _ = Σ a ∈ s, Σ b ∈ t, if r a b then 1 else 0 := by simp
    _ = Σ b ∈ t, Σ a ∈ s, if r a b then 1 else 0 := sum_comm
    _ = Σ b ∈ t, #({a ∈ s | r a b}) := by simp
    _ ≤ Σ b ∈ t, 1 := sum_le_sum h_right
    _ ≤ #(t) := by simp
```

The following exercise is also taken from Mehta's tutorial. Suppose `A` is a subset of `range (2 * n)` with `n + 1` elements. It's easy to see that `A` must contain two consecutive integers, and hence two elements that are coprime. If you watch the tutorial, you will see that a good deal of effort was spent in establishing the following fact, which is now proved

automatically by `omega`.

```
example (m k : ℕ) (h : m ≠ k) (h' : m / 2 = k / 2) : m = k + 1 ∨ k = m + 1 := by omega
```

The proof of the claim uses the pigeonhole principle, in the form `exists_lt_card_fiber_of_mul_lt_card_of_maps_to`, to show that there are two distinct elements `m` and `k` in `A` such that `m / 2 = k / 2`. See if you can complete the justification of that fact and then use it to finish the proof.

```
example {n : ℕ} (A : Finset ℕ)
    (hA : #(A) = n + 1)
    (hA' : A ⊆ range (2 * n)) :
    ∃ m ∈ A, ∃ k ∈ A, Nat.Coprime m k := by
  have : ∃ t ∈ range n, 1 < #({u ∈ A | u / 2 = t}) := by
    apply exists_lt_card_fiber_of_mul_lt_card_of_maps_to
    · sorry
    · sorry
  rcases this with ⟨t, ht, ht'⟩
  simp only [one_lt_card, mem_filter] at ht'
  sorry
```

6.4 Inductively Defined Types

Lean's foundation allows us to define inductive types, that is, data types whose instances are generated from the bottom up. For example, the data type `List α` of lists of elements of α is generated by starting with the empty list, `nil`, and successively adding elements to the front the list. Below we will define a type of binary trees, `BinTree`, whose elements are generated by starting with the empty tree and building new trees by attaching a new node to two existing trees.

In Lean, one can define inductive types whose objects are infinite, like countably branching well-founded trees. Finite inductive definitions are commonly used in discrete mathematics, however, especially in those branches of discrete mathematics that are relevant to computer science. Lean provides not only the means to define such types, but also principles of induction and definition by recursion. For example, the data type `List α` is defined inductively:

```
namespace MyListSpace

inductive List (α : Type*) where
  | nil  : List α
  | cons : α → List α → List α

end MyListSpace
```

The inductive definition says that every element of `List α` is either `nil`, the empty list, or `cons a as`, where `a` is an element of α and `as` is a list of elements of α. The constructors are properly named `List.nil` and `List.cons`, but you can use the shorter notation with the `List` namespace is open. When the `List` namespace is *not* open, you can write `.nil` and `.cons a as` anywhere that `Lean` expects a list, and Lean will automatically insert the `List` qualifier. Throughout this section, we will put temporary definitions in separate namespaces like `MyListSpace` to avoid conflicts with the standard library. Outside the temporary namespace, we revert to using the standard library definitions.

Lean defines the notation `[]` for `nil` and `::` for `cons`, and you can write `[a, b, c]` for `a :: b :: c :: []`. The append and map functions are defined recursively as follows:

```
def append {α : Type*} : List α → List α → List α
  | [],      bs => bs
  | a :: as, bs => a :: (append as bs)
```

(continues on next page)

(continued from previous page)

```
def map {α β : Type*} (f : α → β) : List α → List β
  | []      => []
  | a :: as => f a :: map f as

#eval append [1, 2, 3] [4, 5, 6]
#eval map (fun n => n^2) [1, 2, 3, 4, 5]
```

Notice that there is a base case and a recursive case. In each case, the two defining clauses hold definitionally:

```
theorem nil_append {α : Type*} (as : List α) : append [] as = as := rfl

theorem cons_append {α : Type*} (a : α) (as : List α) (bs : List α) :
    append (a :: as) bs = a :: (append as bs) := rfl

theorem map_nil {α β : Type*} (f : α → β) : map f [] = [] := rfl

theorem map_cons {α β : Type*} (f : α → β) (a : α) (as : List α) :
    map f (a :: as) = f a :: map f as := rfl
```

The functions `append` and `map` are defined in the standard library, and `append as bs` can be written as `as ++ bs`.

Lean allows you to write proofs by induction following the structure of the definition.

```
variable {α β γ : Type*}
variable (as bs cs : List α)
variable (a b c : α)

open List

theorem append_nil : ∀ as : List α, as ++ [] = as
  | [] => rfl
  | a :: as => by rw [cons_append, append_nil as]

theorem map_map (f : α → β) (g : β → γ) :
    ∀ as : List α, map g (map f as) = map (g ∘ f) as
  | [] => rfl
  | a :: as => by rw [map_cons, map_cons, map_cons, map_map f g as]; rfl
```

You can also use the `induction'` tactic.

Of course, these theorems are already in the standard library. As an exercise, try defining a function `reverse` in the `MyListSpace3` namespace (to avoid conflicting with the standard `List.reverse`) that reverses a list. You can use `#eval reverse [1, 2, 3, 4, 5]` to test it out. The most straightforward definition of `reverse` requires quadratic time, but don't worry about that. You can jump to the definition of `List.reverse` in the standard library to see a linear time implementation. Try proving `reverse (as ++ bs) = reverse bs ++ reverse as` and `reverse (reverse as) = as`. You can use `cons_append` and `append_assoc`, but you You may need to come up with auxiliary lemmas and prove them.

```
def reverse : List α → List α := sorry

theorem reverse_append (as bs : List α) : reverse (as ++ bs) = reverse bs ++ reverse␣
↪as := by
  sorry

theorem reverse_reverse (as : List α) : reverse (reverse as) = as := by sorry
```

For another example, consider the following inductive definition of binary trees together with functions to compute the size and depth of a binary tree.

```
inductive BinTree where
  | empty : BinTree
  | node : BinTree → BinTree → BinTree

namespace BinTree

def size : BinTree → ℕ
  | empty => 0
  | node l r => size l + size r + 1

def depth : BinTree → ℕ
  | empty => 0
  | node l r => max (depth l) (depth r) + 1
```

It is convenient to count the empty binary tree as a binary tree of size 0 and depth 0. In the literature, this data type is sometimes called the *extended binary trees*. Including the empty tree means, for example, that we can define the tree `node empty (node empty empty)` consisting of a root node, and empty left subtree, and a right subtree consisting of a single node.

Here is an important inequality relating the size and the depth:

```
theorem size_le : ∀ t : BinTree, size t ≤ 2^depth t - 1
  | empty => Nat.zero_le _
  | node l r => by
    simp only [depth, size]
    calc l.size + r.size + 1
      ≤ (2^l.depth - 1) + (2^r.depth - 1) + 1 := by
          gcongr <;> apply size_le
    _ ≤ (2 ^ max l.depth r.depth - 1) + (2 ^ max l.depth r.depth - 1) + 1 := by
        gcongr <;> simp
    _ ≤ 2 ^ (max l.depth r.depth + 1) - 1 := by
        have : 0 < 2 ^ max l.depth r.depth := by simp
        omega
```

Try proving the following inequality, which is somewhat easier. Remember, if you do a proof by induction as in the previous theorem, you have to delete the `:= by`.

```
theorem depth_le_size : ∀ t : BinTree, depth t ≤ size t := by sorry
```

Also define the `flip` operation on binary trees, which recursively swaps the left and right subtrees.

```
def flip : BinTree → BinTree := sorry
```

If you did it right, the proof of the following should be *rfl*.

```
example: flip  (node (node empty (node empty empty)) (node empty empty)) =
    node (node empty empty) (node (node empty empty) empty) := sorry
```

Prove the following:

```
theorem size_flip : ∀ t, size (flip t) = size t := by sorry
```

We close this section with some formal logic. The following is an inductive definition of propositional formulas.

```
inductive PropForm : Type where
  | var (n : ℕ)           : PropForm
  | fls                   : PropForm
  | conj (A B : PropForm) : PropForm
  | disj (A B : PropForm) : PropForm
  | impl (A B : PropForm) : PropForm
```

Every propositional formula is either a variable `var n`, the constant false `fls`, or a compound formula of the form `conj A B`, `disj A B`, or `impl A B`. With ordinary mathematical notation, these are commonly written p_n, $\bot$, $A \wedge B$, $A \vee B$, and $A \to B$, respectively. The other propositional connectives can be defined in terms of these; for example, we can define $\neg A$ as $A \to \bot$ and $A \leftrightarrow B$ as $(A \to B) \wedge (B \to A)$.

Having defined the data type of propositional formulas, we define what it means to evaluate a propositional formula with respect to an assignment `v` of Boolean truth values to the variables.

```
def eval : PropForm → (ℕ → Bool) → Bool
  | var n,    v => v n
  | fls,      _ => false
  | conj A B, v => A.eval v && B.eval v
  | disj A B, v => A.eval v || B.eval v
  | impl A B, v => ! A.eval v || B.eval v
```

The next definition specifies the set of variables occurring in a formula, and the subsequent theorem shows that evaluating a formula on two truth assignments that agree on its variables yields that same value.

```
def vars : PropForm → Finset ℕ
  | var n    => {n}
  | fls      => ∅
  | conj A B => A.vars ∪ B.vars
  | disj A B => A.vars ∪ B.vars
  | impl A B => A.vars ∪ B.vars

theorem eval_eq_eval : ∀ (A : PropForm) (v1 v2 : ℕ → Bool),
    (∀ n ∈ A.vars, v1 n = v2 n) → A.eval v1 = A.eval v2
  | var n, v1, v2, h => by simp_all [vars, eval, h]
  | fls, v1, v2, h => by simp_all [eval]
  | conj A B, v1, v2, h => by
    simp_all [vars, eval, eval_eq_eval A v1 v2, eval_eq_eval B v1 v2]
  | disj A B, v1, v2, h => by
    simp_all [vars, eval, eval_eq_eval A v1 v2, eval_eq_eval B v1 v2]
  | impl A B, v1, v2, h => by
    simp_all [vars, eval, eval_eq_eval A v1 v2, eval_eq_eval B v1 v2]
```

Noticing the repetition, we can be clever about the use of automation.

```
theorem eval_eq_eval' (A : PropForm) (v1 v2 : ℕ → Bool) (h : ∀ n ∈ A.vars, v1 n = v2␣
↪n) :
    A.eval v1 = A.eval v2 := by
  cases A <;> simp_all [eval, vars, fun A => eval_eq_eval' A v1 v2]
```

The function `subst A m C` describes the result of substituting the formula `C` for every occurrence of the variable `var m` in the formula `A`.

```
def subst : PropForm → ℕ → PropForm → PropForm
  | var n,    m, C => if n = m then C else var n
  | fls,      _, _ => fls
  | conj A B, m, C => conj (A.subst m C) (B.subst m C)
```

(continues on next page)

(continued from previous page)

```
  | disj A B, m, C => disj (A.subst m C) (B.subst m C)
  | impl A B, m, C => impl (A.subst m C) (B.subst m C)
```

As an example, show that substituting for a variable that does not occur in a formula has no effect:

```
theorem subst_eq_of_not_mem_vars :
    ∀ (A : PropForm) (n : ℕ) (C : PropForm), n ∉ A.vars → A.subst n C = A := sorry
```

The following theorem says something more subtle and interesting: evaluating `A.subst n C` on a truth assignment `v` is the same as evaluating `A` on a truth assignment that assigns the value of `C` to `var n`. See if you can prove it.

```
theorem subst_eval_eq : ∀ (A : PropForm) (n : ℕ) (C : PropForm) (v : ℕ → Bool),
  (A.subst n C).eval v = A.eval (fun m => if m = n then C.eval v else v m) := sorry
```

CHAPTER

SEVEN

STRUCTURES

Modern mathematics makes essential use of algebraic structures, which encapsulate patterns that can be instantiated in multiple settings. The subject provides various ways of defining such structures and constructing particular instances.

Lean therefore provides corresponding ways of defining structures formally and working with them. You have already seen examples of algebraic structures in Lean, such as rings and lattices, which were discussed in Chapter 2. This chapter will explain the mysterious square bracket annotations that you saw there, `[Ring α]` and `[Lattice α]`. It will also show you how to define and use algebraic structures on your own.

For more technical detail, you can consult Theorem Proving in Lean, and a paper by Anne Baanen, Use and abuse of instance parameters in the Lean mathematical library.

7.1 Defining structures

In the broadest sense of the term, a *structure* is a specification of a collection of data, possibly with constraints that the data is required to satisfy. An *instance* of the structure is a particular bundle of data satisfying the constraints. For example, we can specify that a point is a tuple of three real numbers:

```
@[ext]
structure Point where
  x : ℝ
  y : ℝ
  z : ℝ
```

The `@[ext]` annotation tells Lean to automatically generate theorems that can be used to prove that two instances of a structure are equal when their components are equal, a property known as *extensionality*.

```
#check Point.ext

example (a b : Point) (hx : a.x = b.x) (hy : a.y = b.y) (hz : a.z = b.z) : a = b := by
  ext
  repeat' assumption
```

We can then define particular instances of the `Point` structure. Lean provides multiple ways of doing that.

```
def myPoint1 : Point where
  x := 2
  y := -1
  z := 4

def myPoint2 : Point :=
  ⟨2, -1, 4⟩
```

(continues on next page)

(continued from previous page)

```
def myPoint3 :=
  Point.mk 2 (-1) 4
```

In the first example, the fields of the structure are named explicitly. The function `Point.mk` referred to in the definition of `myPoint3` is known as the *constructor* for the `Point` structure, because it serves to construct elements. You can specify a different name if you want, like `build`.

```
structure Point' where build ::
  x : ℝ
  y : ℝ
  z : ℝ

#check Point'.build 2 (-1) 4
```

The next two examples show how to define functions on structures. Whereas the second example makes the `Point.mk` constructor explicit, the first example uses an anonymous constructor for brevity. Lean can infer the relevant constructor from the indicated type of `add`. It is conventional to put definitions and theorems associated with a structure like `Point` in a namespace with the same name. In the example below, because we have opened the `Point` namespace, the full name of `add` is `Point.add`. When the namespace is not open, we have to use the full name. But remember that it is often convenient to use anonymous projection notation, which allows us to write `a.add b` instead of `Point.add a b`. Lean interprets the former as the latter because `a` has type `Point`.

```
namespace Point

def add (a b : Point) : Point :=
  ⟨a.x + b.x, a.y + b.y, a.z + b.z⟩

def add' (a b : Point) : Point where
  x := a.x + b.x
  y := a.y + b.y
  z := a.z + b.z

#check add myPoint1 myPoint2
#check myPoint1.add myPoint2

end Point

#check Point.add myPoint1 myPoint2
#check myPoint1.add myPoint2
```

Below we will continue to put definitions in the relevant namespace, but we will leave the namespacing commands out of the quoted snippets. To prove properties of the addition function, we can use `rw` to expand the definition and `ext` to reduce an equation between two elements of the structure to equations between the components. Below we use the `protected` keyword so that the name of the theorem is `Point.add_comm`, even when the namespace is open. This is helpful when we want to avoid ambiguity with a generic theorem like `add_comm`.

```
protected theorem add_comm (a b : Point) : add a b = add b a := by
  rw [add, add]
  ext <;> dsimp
  repeat' apply add_comm

example (a b : Point) : add a b = add b a := by simp [add, add_comm]
```

Because Lean can unfold definitions and simplify projections internally, sometimes the equations we want hold definitionally.

```
theorem add_x (a b : Point) : (a.add b).x = a.x + b.x :=
  rfl
```

It is also possible to define functions on structures using pattern matching, in a manner similar to the way we defined recursive functions in Section 5.2. The definitions `addAlt` and `addAlt'` below are essentially the same; the only difference is that we use anonymous constructor notation in the second. Although it is sometimes convenient to define functions this way, and structural eta-reduction makes this alternative definitionally equivalent, it can make things less convenient in later proofs. In particular, `rw [addAlt]` leaves us with a messier goal view containing a `match` statement.

```
def addAlt : Point → Point → Point
  | Point.mk x₁ y₁ z₁, Point.mk x₂ y₂ z₂ => ⟨x₁ + x₂, y₁ + y₂, z₁ + z₂⟩

def addAlt' : Point → Point → Point
  | ⟨x₁, y₁, z₁⟩, ⟨x₂, y₂, z₂⟩ => ⟨x₁ + x₂, y₁ + y₂, z₁ + z₂⟩

theorem addAlt_x (a b : Point) : (a.addAlt b).x = a.x + b.x := by
  rfl

theorem addAlt_comm (a b : Point) : addAlt a b = addAlt b a := by
  rw [addAlt, addAlt]
  -- the same proof still works, but the goal view here is harder to read
  ext <;> dsimp
  repeat' apply add_comm
```

Mathematical constructions often involve taking apart bundled information and putting it together again in different ways. It therefore makes sense that Lean and Mathlib offer so many ways of doing this efficiently. As an exercise, try proving that `Point.add` is associative. Then define scalar multiplication for a point and show that it distributes over addition.

```
protected theorem add_assoc (a b c : Point) : (a.add b).add c = a.add (b.add c) := by
  sorry

def smul (r : ℝ) (a : Point) : Point :=
  sorry

theorem smul_distrib (r : ℝ) (a b : Point) :
    (smul r a).add (smul r b) = smul r (a.add b) := by
  sorry
```

Using structures is only the first step on the road to algebraic abstraction. We don't yet have a way to link `Point.add` to the generic + symbol, or to connect `Point.add_comm` and `Point.add_assoc` to the generic `add_comm` and `add_assoc` theorems. These tasks belong to the *algebraic* aspect of using structures, and we will explain how to carry them out in the next section. For now, just think of a structure as a way of bundling together objects and information.

It is especially useful that a structure can specify not only data types but also constraints that the data must satisfy. In Lean, the latter are represented as fields of type `Prop`. For example, the *standard 2-simplex* is defined to be the set of points (x, y, z) satisfying $x \geq 0$, $y \geq 0$, $z \geq 0$, and $x + y + z = 1$. If you are not familiar with the notion, you should draw a picture, and convince yourself that this set is the equilateral triangle in three-space with vertices $(1, 0, 0)$, $(0, 1, 0)$, and $(0, 0, 1)$, together with its interior. We can represent it in Lean as follows:

```
structure StandardTwoSimplex where
  x : ℝ
  y : ℝ
  z : ℝ
  x_nonneg : 0 ≤ x
  y_nonneg : 0 ≤ y
  z_nonneg : 0 ≤ z
  sum_eq : x + y + z = 1
```

Notice that the last four fields refer to `x`, `y`, and `z`, that is, the first three fields. We can define a map from the two-simplex to itself that swaps `x` and `y`:

```
def swapXy (a : StandardTwoSimplex) : StandardTwoSimplex
    where
  x := a.y
  y := a.x
  z := a.z
  x_nonneg := a.y_nonneg
  y_nonneg := a.x_nonneg
  z_nonneg := a.z_nonneg
  sum_eq := by rw [add_comm a.y a.x, a.sum_eq]
```

More interestingly, we can compute the midpoint of two points on the simplex. We have added the phrase `noncomputable section` at the beginning of this file in order to use division on the real numbers.

```
noncomputable section

def midpoint (a b : StandardTwoSimplex) : StandardTwoSimplex
    where
  x := (a.x + b.x) / 2
  y := (a.y + b.y) / 2
  z := (a.z + b.z) / 2
  x_nonneg := div_nonneg (add_nonneg a.x_nonneg b.x_nonneg) (by norm_num)
  y_nonneg := div_nonneg (add_nonneg a.y_nonneg b.y_nonneg) (by norm_num)
  z_nonneg := div_nonneg (add_nonneg a.z_nonneg b.z_nonneg) (by norm_num)
  sum_eq := by field_simp; linarith [a.sum_eq, b.sum_eq]
```

Here we have established `x_nonneg`, `y_nonneg`, and `z_nonneg` with concise proof terms, but establish `sum_eq` in tactic mode, using `by`.

Given a parameter λ satisfying $0 \le \lambda \le 1$, we can take the weighted average $\lambda a + (1 - \lambda)b$ of two points a and b in the standard 2-simplex. We challenge you to define that function, in analogy to the `midpoint` function above.

```
def weightedAverage (lambda : Real) (lambda_nonneg : 0 ≤ lambda) (lambda_le : lambda␣
↪≤ 1)
    (a b : StandardTwoSimplex) : StandardTwoSimplex :=
  sorry
```

Structures can depend on parameters. For example, we can generalize the standard 2-simplex to the standard n-simplex for any n. At this stage, you don't have to know anything about the type `Fin n` except that it has n elements, and that Lean knows how to sum over it.

```
open BigOperators

structure StandardSimplex (n : ℕ) where
  V : Fin n → ℝ
  NonNeg : ∀ i : Fin n, 0 ≤ V i
  sum_eq_one : (Σ i, V i) = 1

namespace StandardSimplex

def midpoint (n : ℕ) (a b : StandardSimplex n) : StandardSimplex n
    where
  V i := (a.V i + b.V i) / 2
  NonNeg := by
    intro i
    apply div_nonneg
```

(continues on next page)

(continued from previous page)

```
    · linarith [a.NonNeg i, b.NonNeg i]
    norm_num
  sum_eq_one := by
    simp [div_eq_mul_inv, ← Finset.sum_mul, Finset.sum_add_distrib,
      a.sum_eq_one, b.sum_eq_one]
    field_simp

end StandardSimplex
```

As an exercise, see if you can define the weighted average of two points in the standard n-simplex. You can use `Finset.sum_add_distrib` and `Finset.mul_sum` to manipulate the relevant sums.

We have seen that structures can be used to bundle together data and properties. Interestingly, they can also be used to bundle together properties without the data. For example, the next structure, `IsLinear`, bundles together the two components of linearity.

```
structure IsLinear (f : ℝ → ℝ) where
  is_additive : ∀ x y, f (x + y) = f x + f y
  preserves_mul : ∀ x c, f (c * x) = c * f x

section
variable (f : ℝ → ℝ) (linf : IsLinear f)

#check linf.is_additive
#check linf.preserves_mul

end
```

It is worth pointing out that structures are not the only way to bundle together data. The `Point` data structure can be defined using the generic type product, and `IsLinear` can be defined with a simple `and`.

```
def Point'' :=
  ℝ × ℝ × ℝ

def IsLinear' (f : ℝ → ℝ) :=
  (∀ x y, f (x + y) = f x + f y) ∧ ∀ x c, f (c * x) = c * f x
```

Generic type constructions can even be used in place of structures with dependencies between their components. For example, the *subtype* construction combines a piece of data with a property. You can think of the type `PReal` in the next example as being the type of positive real numbers. Any `x : PReal` has two components: the value, and the property of being positive. You can access these components as `x.val`, which has type $\mathbb{R}$, and `x.property`, which represents the fact `0 < x.val`.

```
def PReal :=
  { y : ℝ // 0 < y }

section
variable (x : PReal)

#check x.val
#check x.property
#check x.1
#check x.2

end
```

We could have used subtypes to define the standard 2-simplex, as well as the standard n-simplex for an arbitrary n.

```
def StandardTwoSimplex' :=
  { p : ℝ × ℝ × ℝ // 0 ≤ p.1 ∧ 0 ≤ p.2.1 ∧ 0 ≤ p.2.2 ∧ p.1 + p.2.1 + p.2.2 = 1 }

def StandardSimplex' (n : ℕ) :=
  { v : Fin n → ℝ // (∀ i : Fin n, 0 ≤ v i) ∧ (Σ i, v i) = 1 }
```

Similarly, *Sigma types* are generalizations of ordered pairs, whereby the type of the second component depends on the type of the first.

```
def StdSimplex := Σ n : ℕ, StandardSimplex n

section
variable (s : StdSimplex)

#check s.fst
#check s.snd

#check s.1
#check s.2

end
```

Given `s : StdSimplex`, the first component `s.fst` is a natural number, and the second component is an element of the corresponding simplex `StandardSimplex s.fst`. The difference between a Sigma type and a subtype is that the second component of a Sigma type is data rather than a proposition.

But even though we can use products, subtypes, and Sigma types instead of structures, using structures has a number of advantages. Defining a structure abstracts away the underlying representation and provides custom names for the functions that access the components. This makes proofs more robust: proofs that rely only on the interface to a structure will generally continue to work when we change the definition, as long as we redefine the old accessors in terms of the new definition. Moreover, as we are about to see, Lean provides support for weaving structures together into a rich, interconnected hierarchy, and for managing the interactions between them.

7.2 Algebraic Structures

To clarify what we mean by the phrase *algebraic structure*, it will help to consider some examples.

1. A *partially ordered set* consists of a set P and a binary relation $\leq$ on P that is transitive and reflexive.
2. A *group* consists of a set G with an associative binary operation, an identity element 1, and a function $g \mapsto g^{-1}$ that returns an inverse for each g in G. A group is *abelian* or *commutative* if the operation is commutative.
3. A *lattice* is a partially ordered set with meets and joins.
4. A *ring* consists of an (additively written) abelian group $(R, +, 0, x \mapsto -x)$ together with an associative multiplication operation $\cdot$ and an identity 1, such that multiplication distributes over addition. A ring is *commutative* if the multiplication is commutative.
5. An *ordered ring* $(R, +, 0, -, \cdot, 1, \leq)$ consists of a ring together with a partial order on its elements, such that $a \leq b$ implies $a + c \leq b + c$ for every a, b, and c in R, and $0 \leq a$ and $0 \leq b$ implies $0 \leq ab$ for every a and b in R.
6. A *metric space* consists of a set X and a function $d : X \times X \to \mathbb{R}$ such that the following hold:
   - $d(x, y) \geq 0$ for every x and y in X.
   - $d(x, y) = 0$ if and only if $x = y$.
   - $d(x, y) = d(y, x)$ for every x and y in X.

- $d(x, z) \leq d(x, y) + d(y, z)$ for every x, y, and z in X.

7. A *topological space* consists of a set X and a collection $\mathcal{T}$ of subsets of X, called the *open subsets of* X, such that the following hold:
   - The empty set and X are open.
   - The intersection of two open sets is open.
   - An arbitrary union of open sets is open.

In each of these examples, the elements of the structure belong to a set, the *carrier set*, that sometimes stands proxy for the entire structure. For example, when we say "let G be a group" and then "let $g \in G$," we are using G to stand for both the structure and its carrier. Not every algebraic structure is associated with a single carrier set in this way. For example, a *bipartite graph* involves a relation between two sets, as does a *Galois connection*, A *category* also involves two sets of interest, commonly called the *objects* and the *morphisms*.

The examples indicate some of the things that a proof assistant has to do in order to support algebraic reasoning. First, it needs to recognize concrete instances of structures. The number systems $\mathbb{Z}$, $\mathbb{Q}$, and $\mathbb{R}$ are all ordered rings, and we should be able to apply a generic theorem about ordered rings in any of these instances. Sometimes a concrete set may be an instance of a structure in more than one way. For example, in addition to the usual topology on $\mathbb{R}$, which forms the basis for real analysis, we can also consider the *discrete* topology on $\mathbb{R}$, in which every set is open.

Second, a proof assistant needs to support generic notation on structures. In Lean, the notation `*` is used for multiplication in all the usual number systems, as well as for multiplication in generic groups and rings. When we use an expression like `f x * y`, Lean has to use information about the types of `f`, `x`, and `y` to determine which multiplication we have in mind.

Third, it needs to deal with the fact that structures can inherit definitions, theorems, and notation from other structures in various ways. Some structures extend others by adding more axioms. A commutative ring is still a ring, so any definition that makes sense in a ring also makes sense in a commutative ring, and any theorem that holds in a ring also holds in a commutative ring. Some structures extend others by adding more data. For example, the additive part of any ring is an additive group. The ring structure adds a multiplication and an identity, as well as axioms that govern them and relate them to the additive part. Sometimes we can define one structure in terms of another. Any metric space has a canonical topology associated with it, the *metric space topology*, and there are various topologies that can be associated with any linear ordering.

Finally, it is important to keep in mind that mathematics allows us to use functions and operations to define structures in the same way we use functions and operations to define numbers. Products and powers of groups are again groups. For every n, the integers modulo n form a ring, and for every $k > 0$, the $k \times k$ matrices of polynomials with coefficients in that ring again form a ring. Thus we can calculate with structures just as easily as we can calculate with their elements. This means that algebraic structures lead dual lives in mathematics, as containers for collections of objects and as objects in their own right. A proof assistant has to accommodate this dual role.

When dealing with elements of a type that has an algebraic structure associated with it, a proof assistant needs to recognize the structure and find the relevant definitions, theorems, and notation. All this should sound like a lot of work, and it is. But Lean uses a small collection of fundamental mechanisms to carry out these tasks. The goal of this section is to explain these mechanisms and show you how to use them.

The first ingredient is almost too obvious to mention: formally speaking, algebraic structures are structures in the sense of Section 7.1. An algebraic structure is a specification of a bundle of data satisfying some axiomatic hypotheses, and we saw in Section 7.1 that this is exactly what the `structure` command is designed to accommodate. It's a marriage made in heaven!

Given a data type α, we can define the group structure on α as follows.

```
structure Group₁ (α : Type*) where
  mul : α → α → α
  one : α
  inv : α → α
```

(continues on next page)

(continued from previous page)

```
  mul_assoc : ∀ x y z : α, mul (mul x y) z = mul x (mul y z)
  mul_one : ∀ x : α, mul x one = x
  one_mul : ∀ x : α, mul one x = x
  inv_mul_cancel : ∀ x : α, mul (inv x) x = one
```

Notice that the type α is a *parameter* in the definition of `Group₁`. So you should think of an object `struc : Group₁ α` as being a group structure on α. We saw in Section 2.2 that the counterpart `mul_inv_cancel` to `inv_mul_cancel` follows from the other group axioms, so there is no need to add it to the definition.

This definition of a group is similar to the definition of `Group` in Mathlib, and we have chosen the name `Group₁` to distinguish our version. If you write `#check Group` and ctrl-click on the definition, you will see that the Mathlib version of `Group` is defined to extend another structure; we will explain how to do that later. If you type `#print Group` you will also see that the Mathlib version of `Group` has a number of extra fields. For reasons we will explain later, sometimes it is useful to add redundant information to a structure, so that there are additional fields for objects and functions that can be defined from the core data. Don't worry about that for now. Rest assured that our simplified version `Group₁` is morally the same as the definition of a group that Mathlib uses.

It is sometimes useful to bundle the type together with the structure, and Mathlib also contains a definition of a `Grp` structure that is equivalent to the following:

```
structure Grp₁ where
  α : Type*
  str : Group₁ α
```

The Mathlib version is found in `Mathlib.Algebra.Category.Grp.Basic`, and you can `#check` it if you add this to the imports at the beginning of the examples file.

For reasons that will become clearer below, it is more often useful to keep the type α separate from the structure `Group` α. We refer to the two objects together as a *partially bundled structure*, since the representation combines most, but not all, of the components into one structure. It is common in Mathlib to use capital roman letters like `G` for a type when it is used as the carrier type for a group.

Let's construct a group, which is to say, an element of the `Group₁` type. For any pair of types α and β, Mathlib defines the type `Equiv α β` of *equivalences* between α and β. Mathlib also defines the suggestive notation $\alpha \simeq \beta$ for this type. An element `f : α ≃ β` is a bijection between α and β represented by four components: a function `f.toFun` from α to β, the inverse function `f.invFun` from β to α, and two properties that specify these functions are indeed inverse to one another.

```
variable (α β γ : Type*)
variable (f : α ≃ β) (g : β ≃ γ)

#check Equiv α β
#check (f.toFun : α → β)
#check (f.invFun : β → α)
#check (f.right_inv : ∀ x : β, f (f.invFun x) = x)
#check (f.left_inv : ∀ x : α, f.invFun (f x) = x)
#check (Equiv.refl α : α ≃ α)
#check (f.symm : β ≃ α)
#check (f.trans g : α ≃ γ)
```

Notice the creative naming of the last three constructions. We think of the identity function `Equiv.refl`, the inverse operation `Equiv.symm`, and the composition operation `Equiv.trans` as explicit evidence that the property of being in bijective correspondence is an equivalence relation.

Notice also that `f.trans g` requires composing the forward functions in reverse order. Mathlib has declared a *coercion* from `Equiv α β` to the function type $\alpha \to \beta$, so we can omit writing `.toFun` and have Lean insert it for us.

```
example (x : α) : (f.trans g).toFun x = g.toFun (f.toFun x) :=
  rfl

example (x : α) : (f.trans g) x = g (f x) :=
  rfl

example : (f.trans g : α → γ) = g ∘ f :=
  rfl
```

Mathlib also defines the type `perm α` of equivalences between α and itself.

```
example (α : Type*) : Equiv.Perm α = (α ≃ α) :=
  rfl
```

It should be clear that `Equiv.Perm α` forms a group under composition of equivalences. We orient things so that `mul f g` is equal to `g.trans f`, whose forward function is `f ∘ g`. In other words, multiplication is what we ordinarily think of as composition of the bijections. Here we define this group:

```
def permGroup {α : Type*} : Group₁ (Equiv.Perm α)
    where
  mul f g := Equiv.trans g f
  one := Equiv.refl α
  inv := Equiv.symm
  mul_assoc f g h := (Equiv.trans_assoc _ _ _).symm
  one_mul := Equiv.trans_refl
  mul_one := Equiv.refl_trans
  inv_mul_cancel := Equiv.self_trans_symm
```

In fact, Mathlib defines exactly this `Group` structure on `Equiv.Perm α` in the file `Algebra.Group.End`. As always, you can hover over the theorems used in the definition of `permGroup` to see their statements, and you can jump to their definitions in the original file to learn more about how they are implemented.

In ordinary mathematics, we generally think of notation as independent of structure. For example, we can consider groups $(G_1, \cdot, 1, \cdot^{-1})$, $(G_2, \circ, e, i(\cdot))$, and $(G_3, +, 0, -)$. In the first case, we write the binary operation as $\cdot$, the identity as 1, and the inverse function as $x \mapsto x^{-1}$. In the second and third cases, we use the notational alternatives shown. When we formalize the notion of a group in Lean, however, the notation is more tightly linked to the structure. In Lean, the components of any `Group` are named `mul`, `one`, and `inv`, and in a moment we will see how multiplicative notation is set up to refer to them. If we want to use additive notation, we instead use an isomorphic structure `AddGroup` (the structure underlying additive groups). Its components are named `add`, `zero`, and `neg`, and the associated notation is what you would expect it to be.

Recall the type `Point` that we defined in Section 7.1, and the addition function that we defined there. These definitions are reproduced in the examples file that accompanies this section. As an exercise, define an `AddGroup₁` structure that is similar to the `Group₁` structure we defined above, except that it uses the additive naming scheme just described. Define negation and a zero on the `Point` data type, and define the `AddGroup₁` structure on `Point`.

```
structure AddGroup₁ (α : Type*) where
  (add : α → α → α)
  -- fill in the rest
@[ext]
structure Point where
  x : ℝ
  y : ℝ
  z : ℝ

namespace Point
```

(continues on next page)

(continued from previous page)

```
def add (a b : Point) : Point :=
  ⟨a.x + b.x, a.y + b.y, a.z + b.z⟩

def neg (a : Point) : Point := sorry

def zero : Point := sorry

def addGroupPoint : AddGroup₁ Point := sorry

end Point
```

We are making progress. Now we know how to define algebraic structures in Lean, and we know how to define instances of those structures. But we also want to associate notation with structures so that we can use it with each instance. Moreover, we want to arrange it so that we can define an operation on a structure and use it with any particular instance, and we want to arrange it so that we can prove a theorem about a structure and use it with any instance.

In fact, Mathlib is already set up to use generic group notation, definitions, and theorems for `Equiv.Perm α`.

```
variable {α : Type*} (f g : Equiv.Perm α) (n : ℕ)

#check f * g
#check mul_assoc f g g⁻¹

-- group power, defined for any group
#check g ^ n

example : f * g * g⁻¹ = f := by rw [mul_assoc, mul_inv_cancel, mul_one]

example : f * g * g⁻¹ = f :=
  mul_inv_cancel_right f g

example {α : Type*} (f g : Equiv.Perm α) : g.symm.trans (g.trans f) = f :=
  mul_inv_cancel_right f g
```

You can check that this is not the case for the additive group structure on `Point` that we asked you to define above. Our task now is to understand that magic that goes on under the hood in order to make the examples for `Equiv.Perm α` work the way they do.

The issue is that Lean needs to be able to *find* the relevant notation and the implicit group structure, using the information that is found in the expressions that we type. Similarly, when we write `x + y` with expressions `x` and `y` that have type $\mathbb{R}$, Lean needs to interpret the + symbol as the relevant addition function on the reals. It also has to recognize the type $\mathbb{R}$ as an instance of a commutative ring, so that all the definitions and theorems for a commutative ring are available. For another example, continuity is defined in Lean relative to any two topological spaces. When we have `f : ℝ → ℂ` and we write `Continuous f`, Lean has to find the relevant topologies on $\mathbb{R}$ and $\mathbb{C}$.

The magic is achieved with a combination of three things.

1. *Logic.* A definition that should be interpreted in any group takes, as arguments, the type of the group and the group structure as arguments. Similarly, a theorem about the elements of an arbitrary group begins with universal quantifiers over the type of the group and the group structure.
2. *Implicit arguments.* The arguments for the type and the structure are generally left implicit, so that we do not have to write them or see them in the Lean information window. Lean fills the information in for us silently.
3. *Type class inference.* Also known as *class inference*, this is a simple but powerful mechanism that enables us to register information for Lean to use later on. When Lean is called on to fill in implicit arguments to a definition, theorem, or piece of notation, it can make use of information that has been registered.

Whereas an annotation `(grp : Group G)` tells Lean that it should expect to be given that argument explicitly and the annotation `{grp : Group G}` tells Lean that it should try to figure it out from contextual cues in the expression, the annotation `[grp : Group G]` tells Lean that the corresponding argument should be synthesized using type class inference. Since the whole point to the use of such arguments is that we generally do not need to refer to them explicitly, Lean allows us to write `[Group G]` and leave the name anonymous. You have probably already noticed that Lean chooses names like `_inst_1` automatically. When we use the anonymous square-bracket annotation with the `variables` command, then as long as the variables are still in scope, Lean automatically adds the argument `[Group G]` to any definition or theorem that mentions `G`.

How do we register the information that Lean needs to use to carry out the search? Returning to our group example, we need only make two changes. First, instead of using the `structure` command to define the group structure, we use the keyword `class` to indicate that it is a candidate for class inference. Second, instead of defining particular instances with `def`, we use the keyword `instance` to register the particular instance with Lean. As with the names of class variables, we are allowed to leave the name of an instance definition anonymous, since in general we intend Lean to find it and put it to use without troubling us with the details.

```
class Group₂ (α : Type*) where
  mul : α → α → α
  one : α
  inv : α → α
  mul_assoc : ∀ x y z : α, mul (mul x y) z = mul x (mul y z)
  mul_one : ∀ x : α, mul x one = x
  one_mul : ∀ x : α, mul one x = x
  inv_mul_cancel : ∀ x : α, mul (inv x) x = one

instance {α : Type*} : Group₂ (Equiv.Perm α) where
  mul f g := Equiv.trans g f
  one := Equiv.refl α
  inv := Equiv.symm
  mul_assoc f g h := (Equiv.trans_assoc _ _ _).symm
  one_mul := Equiv.trans_refl
  mul_one := Equiv.refl_trans
  inv_mul_cancel := Equiv.self_trans_symm
```

The following illustrates their use.

```
#check Group₂.mul

def mySquare {α : Type*} [Group₂ α] (x : α) :=
  Group₂.mul x x

#check mySquare

section
variable {β : Type*} (f g : Equiv.Perm β)

example : Group₂.mul f g = g.trans f :=
  rfl

example : mySquare f = f.trans f :=
  rfl

end
```

The `#check` command shows that `Group₂.mul` has an implicit argument `[Group₂ α]` that we expect to be found by class inference, where α is the type of the arguments to `Group₂.mul`. In other words, `{α : Type*}` is the implicit argument for the type of the group elements and `[Group₂ α]` is the implicit argument for the group structure on α. Similarly, when we define a generic squaring function `my_square` for `Group₂`, we use an implicit argument `{α`

: Type*} for the type of the elements and an implicit argument [Group₂ α] for the Group₂ structure.

In the first example, when we write Group₂.mul f g, the type of f and g tells Lean that in the argument α to Group₂.mul has to be instantiated to Equiv.Perm β. That means that Lean has to find an element of Group₂ (Equiv.Perm β). The previous instance declaration tells Lean exactly how to do that. Problem solved!

This simple mechanism for registering information so that Lean can find it when it needs it is remarkably useful. Here is one way it comes up. In Lean's foundation, a data type α may be empty. In a number of applications, however, it is useful to know that a type has at least one element. For example, the function List.headI, which returns the first element of a list, can return the default value when the list is empty. To make that work, the Lean library defines a class Inhabited α, which does nothing more than store a default value. We can show that the Point type is an instance:

```
instance : Inhabited Point where default := ⟨0, 0, 0⟩

#check (default : Point)

example : ([] : List Point).headI = default :=
  rfl
```

The class inference mechanism is also used for generic notation. The expression x + y is an abbreviation for Add.add x y where—you guessed it—Add α is a class that stores a binary function on α. Writing x + y tells Lean to find a registered instance of [Add.add α] and use the corresponding function. Below, we register the addition function for Point.

```
instance : Add Point where add := Point.add

section
variable (x y : Point)

#check x + y

example : x + y = Point.add x y :=
  rfl

end
```

In this way, we can assign the notation + to binary operations on other types as well.

But we can do even better. We have seen that * can be used in any group, + can be used in any additive group, and both can be used in any ring. When we define a new instance of a ring in Lean, we don't have to define + and * for that instance, because Lean knows that these are defined for every ring. We can use this method to specify notation for our Group₂ class:

```
instance {α : Type*} [Group₂ α] : Mul α :=
  ⟨Group₂.mul⟩

instance {α : Type*} [Group₂ α] : One α :=
  ⟨Group₂.one⟩

instance {α : Type*} [Group₂ α] : Inv α :=
  ⟨Group₂.inv⟩

section
variable {α : Type*} (f g : Equiv.Perm α)

#check f * 1 * g⁻¹

def foo : f * 1 * g⁻¹ = g.symm.trans ((Equiv.refl α).trans f) :=
```

(continues on next page)

(continued from previous page)

```
  rfl

end
```

What makes this approach work is that Lean carries out a recursive search. According to the instances we have declared, Lean can find an instance of `Mul (Equiv.Perm α)` by finding an instance of `Group₂ (Equiv.Perm α)`, and it can find an instance of `Group₂ (Equiv.Perm α)` because we have provided one. Lean is capable of finding these two facts and chaining them together.

The example we have just given is dangerous, because Lean's library also has an instance of `Group (Equiv.Perm α)`, and multiplication is defined on any group. So it is ambiguous as to which instance is found. In fact, Lean favors more recent declarations unless you explicitly specify a different priority. Also, there is another way to tell Lean that one structure is an instance of another, using the `extends` keyword. This is how Mathlib specifies that, for example, every commutative ring is a ring. You can find more information in Section 8 and in a section on class inference in *Theorem Proving in Lean*.

In general, it is a bad idea to specify a value of `*` for an instance of an algebraic structure that already has the notation defined. Redefining the notion of `Group` in Lean is an artificial example. In this case, however, both interpretations of the group notation unfold to `Equiv.trans`, `Equiv.refl`, and `Equiv.symm`, in the same way.

As a similarly artificial exercise, define a class `AddGroup₂` in analogy to `Group₂`. Define the usual notation for addition, negation, and zero on any `AddGroup₂` using the classes `Add`, `Neg`, and `Zero`. Then show `Point` is an instance of `AddGroup₂`. Try it out and make sure that the additive group notation works for elements of `Point`.

```
class AddGroup₂ (α : Type*) where
  add : α → α → α
  -- fill in the rest
```

It is not a big problem that we have already declared instances `Add`, `Neg`, and `Zero` for `Point` above. Once again, the two ways of synthesizing the notation should come up with the same answer.

Class inference is subtle, and you have to be careful when using it, because it configures automation that invisibly governs the interpretation of the expressions we type. When used wisely, however, class inference is a powerful tool. It is what makes algebraic reasoning possible in Lean.

7.3 Building the Gaussian Integers

We will now illustrate the use of the algebraic hierarchy in Lean by building an important mathematical object, the *Gaussian integers*, and showing that it is a Euclidean domain. In other words, according to the terminology we have been using, we will define the Gaussian integers and show that they are an instance of the Euclidean domain structure.

In ordinary mathematical terms, the set of Gaussian integers $\mathbb{Z}[i]$ is the set of complex numbers $\{a + bi \mid a, b \in \mathbb{Z}\}$. But rather than define them as a subset of the complex numbers, our goal here is to define them as a data type in their own right. We do this by representing a Gaussian integer as a pair of integers, which we think of as the *real* and *imaginary* parts.

```
@[ext]
structure GaussInt where
  re : ℤ
  im : ℤ
```

We first show that the Gaussian integers have the structure of a ring, with `0` defined to be `⟨0, 0⟩`, `1` defined to be `⟨1, 0⟩`, and addition defined pointwise. To work out the definition of multiplication, remember that we want the element i,

represented by ⟨0, 1⟩, to be a square root of -1. Thus we want

$$\begin{aligned}(a + bi)(c + di) &= ac + bci + adi + bdi^2 \\ &= (ac - bd) + (bc + ad)i.\end{aligned}$$

This explains the definition of `Mul` below.

```
instance : Zero GaussInt :=
  ⟨⟨0, 0⟩⟩

instance : One GaussInt :=
  ⟨⟨1, 0⟩⟩

instance : Add GaussInt :=
  ⟨fun x y ↦ ⟨x.re + y.re, x.im + y.im⟩⟩

instance : Neg GaussInt :=
  ⟨fun x ↦ ⟨-x.re, -x.im⟩⟩

instance : Mul GaussInt :=
  ⟨fun x y ↦ ⟨x.re * y.re - x.im * y.im, x.re * y.im + x.im * y.re⟩⟩
```

As noted in Section 7.1, it is a good idea to put all the definitions related to a data type in a namespace with the same name. Thus in the Lean files associated with this chapter, these definitions are made in the `GaussInt` namespace.

Notice that here we are defining the interpretations of the notation `0`, `1`, `+`, `-`, and `*` directly, rather than naming them `GaussInt.zero` and the like and assigning the notation to those. It is often useful to have an explicit name for the definitions, for example, to use with `simp` and `rw`.

```
theorem zero_def : (0 : GaussInt) = ⟨0, 0⟩ :=
  rfl

theorem one_def : (1 : GaussInt) = ⟨1, 0⟩ :=
  rfl

theorem add_def (x y : GaussInt) : x + y = ⟨x.re + y.re, x.im + y.im⟩ :=
  rfl

theorem neg_def (x : GaussInt) : -x = ⟨-x.re, -x.im⟩ :=
  rfl

theorem mul_def (x y : GaussInt) :
    x * y = ⟨x.re * y.re - x.im * y.im, x.re * y.im + x.im * y.re⟩ :=
  rfl
```

It is also useful to name the rules that compute the real and imaginary parts, and to declare them to the simplifier.

```
@[simp]
theorem zero_re : (0 : GaussInt).re = 0 :=
  rfl

@[simp]
theorem zero_im : (0 : GaussInt).im = 0 :=
  rfl

@[simp]
theorem one_re : (1 : GaussInt).re = 1 :=
  rfl
```

(continues on next page)

(continued from previous page)

```
@[simp]
theorem one_im : (1 : GaussInt).im = 0 :=
  rfl

@[simp]
theorem add_re (x y : GaussInt) : (x + y).re = x.re + y.re :=
  rfl

@[simp]
theorem add_im (x y : GaussInt) : (x + y).im = x.im + y.im :=
  rfl

@[simp]
theorem neg_re (x : GaussInt) : (-x).re = -x.re :=
  rfl

@[simp]
theorem neg_im (x : GaussInt) : (-x).im = -x.im :=
  rfl

@[simp]
theorem mul_re (x y : GaussInt) : (x * y).re = x.re * y.re - x.im * y.im :=
  rfl

@[simp]
theorem mul_im (x y : GaussInt) : (x * y).im = x.re * y.im + x.im * y.re :=
  rfl
```

It is now surprisingly easy to show that the Gaussian integers are an instance of a commutative ring. We are putting the structure concept to good use. Each particular Gaussian integer is an instance of the `GaussInt` structure, whereas the type `GaussInt` itself, together with the relevant operations, is an instance of the `CommRing` structure. The `CommRing` structure, in turn, extends the notational structures `Zero`, `One`, `Add`, `Neg`, and `Mul`.

If you type `instance : CommRing GaussInt := _`, click on the light bulb that appears in VS Code, and then ask Lean to fill in a skeleton for the structure definition, you will see a scary number of entries. Jumping to the definition of the structure, however, shows that many of the fields have default definitions that Lean will fill in for you automatically. The essential ones appear in the definition below. A special case are `nsmul` and `zsmul` which should be ignored for now and will be explained in the next chapter. In each case, the relevant identity is proved by unfolding definitions, using the `ext` tactic to reduce the identities to their real and imaginary components, simplifying, and, if necessary, carrying out the relevant ring calculation in the integers. Note that we could easily avoid repeating all this code, but this is not the topic of the current discussion.

```
instance instCommRing : CommRing GaussInt where
  zero := 0
  one := 1
  add := (· + ·)
  neg x := -x
  mul := (· * ·)
  nsmul := nsmulRec
  zsmul := zsmulRec
  add_assoc := by
    intros
    ext <;> simp <;> ring
  zero_add := by
    intro
```

(continues on next page)

(continued from previous page)

```
    ext <;> simp
  add_zero := by
    intro
    ext <;> simp
  neg_add_cancel := by
    intro
    ext <;> simp
  add_comm := by
    intros
    ext <;> simp <;> ring
  mul_assoc := by
    intros
    ext <;> simp <;> ring
  one_mul := by
    intro
    ext <;> simp
  mul_one := by
    intro
    ext <;> simp
  left_distrib := by
    intros
    ext <;> simp <;> ring
  right_distrib := by
    intros
    ext <;> simp <;> ring
  mul_comm := by
    intros
    ext <;> simp <;> ring
  zero_mul := by
    intros
    ext <;> simp
  mul_zero := by
    intros
    ext <;> simp
```

Lean's library defines the class of *nontrivial* types to be types with at least two distinct elements. In the context of a ring, this is equivalent to saying that the zero is not equal to the one. Since some common theorems depend on that fact, we may as well establish it now.

```
instance : Nontrivial GaussInt := by
  use 0, 1
  rw [Ne, GaussInt.ext_iff]
  simp
```

We will now show that the Gaussian integers have an important additional property. A *Euclidean domain* is a ring R equipped with a *norm* function $N : R \to \mathbb{N}$ with the following two properties:

- For every a and $b \neq 0$ in R, there are q and r in R such that $a = bq + r$ and either $r = 0$ or $N(r) < N(b)$.
- For every a and $b \neq 0$, $N(a) \leq N(ab)$.

The ring of integers $\mathbb{Z}$ with $N(a) = |a|$ is an archetypal example of a Euclidean domain. In that case, we can take q to be the result of integer division of a by b and r to be the remainder. These functions are defined in Lean so that the satisfy the following:

```
example (a b : ℤ) : a = b * (a / b) + a % b :=
  Eq.symm (Int.ediv_add_emod a b)
```

(continues on next page)

(continued from previous page)

```
example (a b : ℤ) : b ≠ 0 → 0 ≤ a % b :=
  Int.emod_nonneg a

example (a b : ℤ) : b ≠ 0 → a % b < |b| :=
  Int.emod_lt_abs a
```

In an arbitrary ring, an element a is said to be a *unit* if it divides 1. A nonzero element a is said to be *irreducible* if it cannot be written in the form $a = bc$ where neither b nor c is a unit. In the integers, every irreducible element a is *prime*, which is to say, whenever a divides a product bc, it divides either b or c. But in other rings this property can fail. In the ring $\mathbb{Z}[\sqrt{-5}]$, we have

$$6 = 2 \cdot 3 = (1 + \sqrt{-5})(1 - \sqrt{-5}),$$

and the elements 2, 3, $1 + \sqrt{-5}$, and $1 - \sqrt{-5}$ are all irreducible, but they are not prime. For example, 2 divides the product $(1 + \sqrt{-5})(1 - \sqrt{-5})$, but it does not divide either factor. In particular, we no longer have unique factorization: the number 6 can be factored into irreducible elements in more than one way.

In contrast, every Euclidean domain is a unique factorization domain, which implies that every irreducible element is prime. The axioms for a Euclidean domain imply that one can write any nonzero element as a finite product of irreducible elements. They also imply that one can use the Euclidean algorithm to find a greatest common divisor of any two nonzero elements `a` and `b`, i.e. an element that is divisible by any other common divisor. This, in turn, implies that factorization into irreducible elements is unique up to multiplication by units.

We now show that the Gaussian integers are a Euclidean domain with the norm defined by $N(a+bi) = (a+bi)(a-bi) = a^2+b^2$. The Gaussian integer $a-bi$ is called the *conjugate* of $a+bi$. It is not hard to check that for any complex numbers x and y, we have $N(xy) = N(x)N(y)$.

To see that this definition of the norm makes the Gaussian integers a Euclidean domain, only the first property is challenging. Suppose we want to write $a + bi = (c + di)q + r$ for suitable q and r. Treating $a + bi$ and $c + di$ as complex numbers, carry out the division

$$\frac{a + bi}{c + di} = \frac{(a + bi)(c - di)}{(c + di)(c - di)} = \frac{ac + bd}{c^2 + d^2} + \frac{bc - ad}{c^2 + d^2} i.$$

The real and imaginary parts might not be integers, but we can round them to the nearest integers u and v. We can then express the right-hand side as $(u + vi) + (u' + v'i)$, where $u' + v'i$ is the part left over. Note that we have $|u'| \leq 1/2$ and $|v'| \leq 1/2$, and hence

$$N(u' + v'i) = (u')^2 + (v')^2 \leq 1/4 + 1/4 \leq 1/2.$$

Multiplying through by $c + di$, we have

$$a + bi = (c + di)(u + vi) + (c + di)(u' + v'i).$$

Setting $q = u + vi$ and $r = (c + di)(u' + v'i)$, we have $a + bi = (c + di)q + r$, and we only need to bound $N(r)$:

$$N(r) = N(c + di)N(u' + v'i) \leq N(c + di) \cdot 1/2 < N(c + di).$$

The argument we just carried out requires viewing the Gaussian integers as a subset of the complex numbers. One option for formalizing it in Lean is therefore to embed the Gaussian integers in the complex numbers, embed the integers in the Gaussian integers, define the rounding function from the real numbers to the integers, and take great care to pass back and forth between these number systems appropriately. In fact, this is exactly the approach that is followed in Mathlib, where the Gaussian integers themselves are constructed as a special case of a ring of *quadratic integers*. See the file GaussianInt.lean.

Here we will instead carry out an argument that stays in the integers. This illustrates a choice one commonly faces when formalizing mathematics. Given an argument that requires concepts or machinery that is not already in the library, one

has two choices: either formalize the concepts and machinery needed, or adapt the argument to make use of concepts and machinery you already have. The first choice is generally a good investment of time when the results can be used in other contexts. Pragmatically speaking, however, sometimes seeking a more elementary proof is more efficient.

The usual quotient-remainder theorem for the integers says that for every a and nonzero b, there are q and r such that $a = bq + r$ and $0 \leq r < b$. Here we will make use of the following variation, which says that there are q' and r' such that $a = bq' + r'$ and $|r'| \leq b/2$. You can check that if the value of r in the first statement satisfies $r \leq b/2$, we can take $q' = q$ and $r' = r$, and otherwise we can take $q' = q + 1$ and $r' = r - b$. We are grateful to Heather Macbeth for suggesting the following more elegant approach, which avoids definition by cases. We simply add `b / 2` to `a` before dividing and then subtract it from the remainder.

```
def div' (a b : ℤ) :=
  (a + b / 2) / b

def mod' (a b : ℤ) :=
  (a + b / 2) % b - b / 2

theorem div'_add_mod' (a b : ℤ) : b * div' a b + mod' a b = a := by
  rw [div', mod']
  linarith [Int.ediv_add_emod (a + b / 2) b]

theorem abs_mod'_le (a b : ℤ) (h : 0 < b) : |mod' a b| ≤ b / 2 := by
  rw [mod', abs_le]
  constructor
  · linarith [Int.emod_nonneg (a + b / 2) h.ne']
  have := Int.emod_lt_of_pos (a + b / 2) h
  have := Int.ediv_add_emod b 2
  have := Int.emod_lt_of_pos b zero_lt_two
  linarith
```

Note the use of our old friend, `linarith`. We will also need to express `mod'` in terms of `div'`.

```
theorem mod'_eq (a b : ℤ) : mod' a b = a - b * div' a b := by linarith [div'_add_mod'␣
↪a b]
```

We will use the fact that $x^2 + y^2$ is equal to zero if and only if x and y are both zero. As an exercise, we ask you to prove that this holds in any ordered ring.

```
theorem sq_add_sq_eq_zero {α : Type*} [Ring α] [LinearOrder α] [IsStrictOrderedRing α]
    (x y : α) : x ^ 2 + y ^ 2 = 0 ↔ x = 0 ∧ y = 0 := by
  sorry
```

We will put all the remaining definitions and theorems in this section in the `GaussInt` namespace. First, we define the `norm` function and ask you to establish some of its properties. The proofs are all short.

```
def norm (x : GaussInt) :=
  x.re ^ 2 + x.im ^ 2

@[simp]
theorem norm_nonneg (x : GaussInt) : 0 ≤ norm x := by
  sorry
theorem norm_eq_zero (x : GaussInt) : norm x = 0 ↔ x = 0 := by
  sorry
theorem norm_pos (x : GaussInt) : 0 < norm x ↔ x ≠ 0 := by
  sorry
theorem norm_mul (x y : GaussInt) : norm (x * y) = norm x * norm y := by
  sorry
```

Next we define the conjugate function:

```
def conj (x : GaussInt) : GaussInt :=
  ⟨x.re, -x.im⟩

@[simp]
theorem conj_re (x : GaussInt) : (conj x).re = x.re :=
  rfl

@[simp]
theorem conj_im (x : GaussInt) : (conj x).im = -x.im :=
  rfl

theorem norm_conj (x : GaussInt) : norm (conj x) = norm x := by simp [norm]
```

Finally, we define division for the Gaussian integers with the notation `x / y`, that rounds the complex quotient to the nearest Gaussian integer. We use our bespoke `Int.div'` for that purpose. As we calculated above, if `x` is $a + bi$ and `y` is $c + di$, then the real and imaginary parts of `x / y` are the nearest integers to

$$\frac{ac + bd}{c^2 + d^2} \quad \text{and} \quad \frac{bc - ad}{c^2 + d^2},$$

respectively. Here the numerators are the real and imaginary parts of $(a + bi)(c - di)$, and the denominators are both equal to the norm of $c + di$.

```
instance : Div GaussInt :=
  ⟨fun x y ↦ ⟨Int.div' (x * conj y).re (norm y), Int.div' (x * conj y).im (norm y)⟩⟩
```

Having defined `x / y`, We define `x % y` to be the remainder, `x - (x / y) * y`. As above, we record the definitions in the theorems `div_def` and `mod_def` so that we can use them with `simp` and `rw`.

```
instance : Mod GaussInt :=
  ⟨fun x y ↦ x - y * (x / y)⟩

theorem div_def (x y : GaussInt) :
    x / y = ⟨Int.div' (x * conj y).re (norm y), Int.div' (x * conj y).im (norm y)⟩ :=
  rfl

theorem mod_def (x y : GaussInt) : x % y = x - y * (x / y) :=
  rfl
```

These definitions immediately yield `x = y * (x / y) + x % y` for every `x` and `y`, so all we need to do is show that the norm of `x % y` is less than the norm of `y` when `y` is not zero.

We just defined the real and imaginary parts of `x / y` to be `div' (x * conj y).re (norm y)` and `div' (x * conj y).im (norm y)`, respectively. Calculating, we have

```
(x % y) * conj y = (x - x / y * y) * conj y = x * conj y - x / y * (y *
conj y)
```

The real and imaginary parts of the right-hand side are exactly `mod' (x * conj y).re (norm y)` and `mod' (x * conj y).im (norm y)`. By the properties of `div'` and `mod'`, these are guaranteed to be less than or equal to `norm y / 2`. So we have

```
norm ((x % y) * conj y) ≤ (norm y / 2)^2 + (norm y / 2)^2 ≤ (norm y /
2) * norm y.
```

On the other hand, we have

```
norm ((x % y) * conj y) = norm (x % y) * norm (conj y) = norm (x % y) *
norm y.
```

Dividing through by `norm y` we have `norm (x % y) ≤ (norm y) / 2 < norm y`, as required.

This messy calculation is carried out in the next proof. We encourage you to step through the details and see if you can find a nicer argument.

```
theorem norm_mod_lt (x : GaussInt) {y : GaussInt} (hy : y ≠ 0) :
    (x % y).norm < y.norm := by
  have norm_y_pos : 0 < norm y := by rwa [norm_pos]
  have H1 : x % y * conj y = ⟨Int.mod' (x * conj y).re (norm y), Int.mod' (x * conj y).
↪im (norm y)⟩
  · ext <;> simp [Int.mod'_eq, mod_def, div_def, norm] <;> ring
  have H2 : norm (x % y) * norm y ≤ norm y / 2 * norm y
  · calc
      norm (x % y) * norm y = norm (x % y * conj y) := by simp only [norm_mul, norm_
↪conj]
      _ = |Int.mod' (x.re * y.re + x.im * y.im) (norm y)| ^ 2
          + |Int.mod' (-(x.re * y.im) + x.im * y.re) (norm y)| ^ 2 := by simp [H1,␣
↪norm, sq_abs]
      _ ≤ (y.norm / 2) ^ 2 + (y.norm / 2) ^ 2 := by gcongr <;> apply Int.abs_mod'_le _
↪ _ norm_y_pos
      _ = norm y / 2 * (norm y / 2 * 2) := by ring
      _ ≤ norm y / 2 * norm y := by gcongr; apply Int.ediv_mul_le; norm_num
  calc norm (x % y) ≤ norm y / 2 := le_of_mul_le_mul_right H2 norm_y_pos
    _ < norm y := by
      apply Int.ediv_lt_of_lt_mul
      · norm_num
      · linarith
```

We are in the home stretch. Our `norm` function maps Gaussian integers to nonnegative integers. We need a function that maps Gaussian integers to natural numbers, and we obtain that by composing `norm` with the function `Int.natAbs`, which maps integers to the natural numbers. The first of the next two lemmas establishes that mapping the norm to the natural numbers and back to the integers does not change the value. The second one re-expresses the fact that the norm is decreasing.

```
theorem coe_natAbs_norm (x : GaussInt) : (x.norm.natAbs : ℤ) = x.norm :=
  Int.natAbs_of_nonneg (norm_nonneg _)

theorem natAbs_norm_mod_lt (x y : GaussInt) (hy : y ≠ 0) :
    (x % y).norm.natAbs < y.norm.natAbs := by
  apply Int.ofNat_lt.1
  simp only [Int.natCast_natAbs, abs_of_nonneg, norm_nonneg]
  exact norm_mod_lt x hy
```

We also need to establish the second key property of the norm function on a Euclidean domain.

```
theorem not_norm_mul_left_lt_norm (x : GaussInt) {y : GaussInt} (hy : y ≠ 0) :
    ¬(norm (x * y)).natAbs < (norm x).natAbs := by
  apply not_lt_of_ge
  rw [norm_mul, Int.natAbs_mul]
  apply le_mul_of_one_le_right (Nat.zero_le _)
  apply Int.ofNat_le.1
  rw [coe_natAbs_norm]
  exact Int.add_one_le_of_lt ((norm_pos _).mpr hy)
```

We can now put it together to show that the Gaussian integers are an instance of a Euclidean domain. We use the quotient and remainder function we have defined. The Mathlib definition of a Euclidean domain is more general than the one above in that it allows us to show that remainder decreases with respect to any well-founded measure. Comparing the values of a norm function that returns natural numbers is just one instance of such a measure, and in that case, the required

properties are the theorems `natAbs_norm_mod_lt` and `not_norm_mul_left_lt_norm`.

```
instance : EuclideanDomain GaussInt :=
  { GaussInt.instCommRing with
    quotient := (· / ·)
    remainder := (· % ·)
    quotient_mul_add_remainder_eq :=
      fun x y ↦ by rw [mod_def, add_comm] ; ring
    quotient_zero := fun x ↦ by
      simp [div_def, norm, Int.div']
      rfl
    r := (measure (Int.natAbs ∘ norm)).1
    r_wellFounded := (measure (Int.natAbs ∘ norm)).2
    remainder_lt := natAbs_norm_mod_lt
    mul_left_not_lt := not_norm_mul_left_lt_norm }
```

An immediate payoff is that we now know that, in the Gaussian integers, the notions of being prime and being irreducible coincide.

```
example (x : GaussInt) : Irreducible x ↔ Prime x :=
  irreducible_iff_prime
```

CHAPTER

EIGHT

HIERARCHIES

We have seen in Chapter 7 how to define the class of groups and build instances of this class, and then how to build an instance of the commutative ring class. But of course there is a hierarchy here: a commutative ring is in particular an additive group. In this chapter we will study how to build such hierarchies. They appear in all branches of mathematics but in this chapter the emphasis will be on algebraic examples.

It may seem premature to discuss how to build hierarchies before more discussions about using existing hierarchies. But some understanding of the technology underlying hierarchies is required to use them. So you should probably still read this chapter, but without trying too hard to remember everything on your first read, then read the following chapters and come back here for a second reading.

In this chapter, we will redefine (simpler versions of) many things that appear in Mathlib so we will used indices to distinguish our version. For instance we will have `Ring₁` as our version of `Ring`. Since we will gradually explain more powerful ways of formalizing structures, those indices will sometimes grow beyond one.

8.1 Basics

At the very bottom of all hierarchies in Lean, we find data-carrying classes. The following class records that the given type α is endowed with a distinguished element called `one`. At this stage, it has no property at all.

```
class One₁ (α : Type) where
  /-- The element one -/
  one : α
```

Since we'll make a much heavier use of classes in this chapter, we need to understand some more details about what the `class` command is doing. First, the `class` command above defines a structure `One₁` with parameter `α : Type` and a single field `one`. It also mark this structure as a class so that arguments of type `One₁ α` for some type α will be inferrable using the instance resolution procedure, as long as they are marked as instance-implicit, i.e. appear between square brackets. Those two effects could also have been achieved using the `structure` command with `class` attribute, i.e. writing `@[class] structure` instance of `class`. But the class command also ensures that `One₁ α` appears as an instance-implicit argument in its own fields. Compare:

```
#check One₁.one -- One₁.one {α : Type} [self : One₁ α] : α

@[class] structure One₂ (α : Type) where
  /-- The element one -/
  one : α

#check One₂.one
```

In the second check, we can see that `self : One₂ α` is an explicit argument. Let us make sure the first version is indeed usable without any explicit argument.

```
example (α : Type) [One₁ α] : α := One₁.one
```

Remark: in the above example, the argument `One₁ α` is marked as instance-implicit, which is a bit silly since this affects only *uses* of the declaration and declaration created by the `example` command cannot be used. However it allows us to avoid giving a name to that argument and, more importantly, it starts installing the good habit of marking `One₁ α` arguments as instance-implicit.

Another remark is that all this will work only when Lean knows what is α. In the above example, leaving out the type ascription `: α` would generate an error message like: `typeclass instance problem is stuck, it is often due to metavariables One₁ (?m.263 α)` where `?m.263 α` means "some type depending on α" (and 263 is simply an auto-generated index that would be useful to distinguish between several unknown things). Another way to avoid this issue would be to use a type annotation, as in:

```
example (α : Type) [One₁ α] := (One₁.one : α)
```

You may have already encountered that issue when playing with limits of sequences in Section 3.6 if you tried to state for instance that `0 < 1` without telling Lean whether you meant this inequality to be about natural numbers or real numbers.

Our next task is to assign a notation to `One₁.one`. Since we don't want collisions with the builtin notation for `1`, we will use `𝟙`. This is achieved by the following command where the first line tells Lean to use the documentation of `One₁.one` as documentation for the symbol `𝟙`.

```
@[inherit_doc]
notation "𝟙" => One₁.one

example {α : Type} [One₁ α] : α := 𝟙

example {α : Type} [One₁ α] : (𝟙 : α) = 𝟙 := rfl
```

We now want a data-carrying class recording a binary operation. We don't want to choose between addition and multiplication for now so we'll use diamond.

```
class Dia₁ (α : Type) where
  dia : α → α → α

infixl:70 " ◇ "   => Dia₁.dia
```

As in the `One₁` example, the operation has no property at all at this stage. Let us now define the class of semigroup structures where the operation is denoted by $\diamond$. For now, we define it by hand as a structure with two fields, a `Dia₁` instance and some `Prop`-valued field `dia_assoc` asserting associativity of $\diamond$.

```
class Semigroup₀ (α : Type) where
  toDia₁ : Dia₁ α
  /-- Diamond is associative -/
  dia_assoc : ∀ a b c : α, a ◇ b ◇ c = a ◇ (b ◇ c)
```

Note that while stating *dia_assoc*, the previously defined field *toDia₁* is in the local context hence can be used when Lean searches for an instance of *Dia₁ α* to make sense of *a ◇ b*. However this *toDia₁* field does not become part of the type class instances database. Hence doing `example {α : Type} [Semigroup₁ α] (a b : α) : α := a ◇ b` would fail with error message `failed to synthesize instance Dia₁ α`.

We can fix this by adding the `instance` attribute later.

```
attribute [instance] Semigroup₀.toDia₁

example {α : Type} [Semigroup₀ α] (a b : α) : α := a ◇ b
```

Before building up, we need to use a different syntax to add this *toDia*$_1$ field, to tell Lean that *Dia*$_1$ α should be treated as if its fields were fields of *Semigroup*$_1$ itself. This also conveniently adds the *toDia*$_1$ instance automatically. The `class` command supports this using the `extends` syntax as in:

```
class Semigroup₁ (α : Type) extends toDia₁ : Dia₁ α where
  /-- Diamond is associative -/
  dia_assoc : ∀ a b c : α, a ◇ b ◇ c = a ◇ (b ◇ c)

example {α : Type} [Semigroup₁ α] (a b : α) : α := a ◇ b
```

Note this syntax is also available in the `structure` command, although it that case it fixes only the hurdle of writing fields such as *toDia*$_1$ since there is no instance to define in that case.

The field name *toDia*$_1$ is optional in the *extends* syntax. By default it takes the name of the class being extended and prefixes it with "to".

```
class Semigroup₂ (α : Type) extends Dia₁ α where
  /-- Diamond is associative -/
  dia_assoc : ∀ a b c : α, a ◇ b ◇ c = a ◇ (b ◇ c)
```

Let us now try to combine a diamond operation and a distinguished one element with axioms saying this element is neutral on both sides.

```
class DiaOneClass₁ (α : Type) extends One₁ α, Dia₁ α where
  /-- One is a left neutral element for diamond. -/
  one_dia : ∀ a : α, 𝟙 ◇ a = a
  /-- One is a right neutral element for diamond -/
  dia_one : ∀ a : α, a ◇ 𝟙 = a
```

In the next example, we tell Lean that α has a `DiaOneClass`$_1$ structure and state a property that uses both a *Dia*$_1$ instance and a *One*$_1$ instance. In order to see how Lean finds those instances we set a tracing option whose result can be seen in the Infoview. This result is rather terse by default but it can be expanded by clicking on lines ending with black arrows. It includes failed attempts where Lean tried to find instances before having enough type information to succeed. The successful attempts do involve the instances generated by the `extends` syntax.

```
set_option trace.Meta.synthInstance true in
example {α : Type} [DiaOneClass₁ α] (a b : α) : Prop := a ◇ b = 𝟙
```

Note that we don't need to include extra fields where combining existing classes. Hence we can define monoids as:

```
class Monoid₁ (α : Type) extends Semigroup₁ α, DiaOneClass₁ α
```

While the above definition seems straightforward, it hides an important subtlety. Both `Semigroup`$_1$ α and `DiaOneClass`$_1$ α extend `Dia`$_1$ α, so one could fear that having a `Monoid`$_1$ α instance gives two unrelated diamond operations on α, one coming from a field `Monoid`$_1$`.toSemigroup`$_1$ and one coming from a field `Monoid`$_1$`.toDiaOneClass`$_1$.

Indeed if we try to build a monoid class by hand using:

```
class Monoid₂ (α : Type) where
  toSemigroup₁ : Semigroup₁ α
  toDiaOneClass₁ : DiaOneClass₁ α
```

then we get two completely unrelated diamond operations `Monoid`$_2$`.toSemigroup`$_1$`.toDia`$_1$`.dia` and `Monoid`$_2$`.toDiaOneClass`$_1$`.toDia`$_1$`.dia`.

The version generated using the `extends` syntax does not have this defect.

```
example {α : Type} [Monoid₁ α] :
  (Monoid₁.toSemigroup₁.toDia₁.dia : α → α → α) = Monoid₁.toDiaOneClass₁.toDia₁.dia
↪:= rfl
```

So the `class` command did some magic for us (and the `structure` command would have done it too). An easy way to see what are the fields of our classes is to check their constructor. Compare:

```
/- Monoid₂.mk {α : Type} (toSemigroup₁ : Semigroup₁ α) (toDiaOneClass₁ : DiaOneClass₁
↪α) : Monoid₂ α -/
#check Monoid₂.mk

/- Monoid₁.mk {α : Type} [toSemigroup₁ : Semigroup₁ α] [toOne₁ : One₁ α] (one_dia : ∀
↪(a : α), 𝟙 ◇ a = a) (dia_one : ∀ (a : α), a ◇ 𝟙 = a) : Monoid₁ α -/
#check Monoid₁.mk
```

So we see that `Monoid₁` takes `Semigroup₁ α` argument as expected but then it won't take a would-be overlapping `DiaOneClass₁ α` argument but instead tears it apart and includes only the non-overlapping parts. And it also auto-generated an instance `Monoid₁.toDiaOneClass₁` which is *not* a field but has the expected signature which, from the end-user point of view, restores the symmetry between the two extended classes `Semigroup₁` and `DiaOneClass₁`.

```
#check Monoid₁.toSemigroup₁
#check Monoid₁.toDiaOneClass₁
```

We are now very close to defining groups. We could add to the monoid structure a field asserting the existence of an inverse for every element. But then we would need to work to access these inverses. In practice it is more convenient to add it as data. To optimize reusability, we define a new data-carrying class, and then give it some notation.

```
class Inv₁ (α : Type) where
  /-- The inversion function -/
  inv : α → α

@[inherit_doc]
postfix:max "⁻¹" => Inv₁.inv

class Group₁ (G : Type) extends Monoid₁ G, Inv₁ G where
  inv_dia : ∀ a : G, a⁻¹ ◇ a = 𝟙
```

The above definition may seem too weak, we only ask that `a⁻¹` is a left-inverse of `a`. But the other side is automatic. In order to prove that, we need a preliminary lemma.

```
lemma left_inv_eq_right_inv₁ {M : Type} [Monoid₁ M] {a b c : M} (hba : b ◇ a = 𝟙) (hac
↪: a ◇ c = 𝟙) : b = c := by
  rw [← DiaOneClass₁.one_dia c, ← hba, Semigroup₁.dia_assoc, hac, DiaOneClass₁.dia_one
↪b]
```

In this lemma, it is pretty annoying to give full names, especially since it requires knowing which part of the hierarchy provides those facts. One way to fix this is to use the `export` command to copy those facts as lemmas in the root name space.

```
export DiaOneClass₁ (one_dia dia_one)
export Semigroup₁ (dia_assoc)
export Group₁ (inv_dia)
```

We can then rewrite the above proof as:

```
example {M : Type} [Monoid₁ M] {a b c : M} (hba : b ◇ a = 𝟙) (hac : a ◇ c = 𝟙) : b = c␣
↪:= by
  rw [← one_dia c, ← hba, dia_assoc, hac, dia_one b]
```

It is now your turn to prove things about our algebraic structures.

```
lemma inv_eq_of_dia [Group₁ G] {a b : G} (h : a ◇ b = 𝟙) : a⁻¹ = b :=
  sorry

lemma dia_inv [Group₁ G] (a : G) : a ◇ a⁻¹ = 𝟙 :=
  sorry
```

At this stage we would like to move on to define rings, but there is a serious issue. A ring structure on a type contains both an additive group structure and a multiplicative monoid structure, and some properties about their interaction. But so far we hard-coded a notation ◇ for all our operations. More fundamentally, the type class system assumes every type has only one instance of each type class. There are various ways to solve this issue. Surprisingly Mathlib uses the naive idea to duplicate everything for additive and multiplicative theories with the help of some code-generating attribute. Structures and classes are defined in both additive and multiplicative notation with an attribute `to_additive` linking them. In case of multiple inheritance like for semi-groups, the auto-generated "symmetry-restoring" instances need also to be marked. This is a bit technical; you don't need to understand details. The important point is that lemmas are then only stated in multiplicative notation and marked with the attribute `to_additive` to generate the additive version as `left_inv_eq_right_inv'` with its auto-generated additive version `left_neg_eq_right_neg'`. In order to check the name of this additive version we used the `whatsnew in` command on top of `left_inv_eq_right_inv'`.

```
class AddSemigroup₃ (α : Type) extends Add α where
  /-- Addition is associative -/
  add_assoc₃ : ∀ a b c : α, a + b + c = a + (b + c)

@[to_additive AddSemigroup₃]
class Semigroup₃ (α : Type) extends Mul α where
  /-- Multiplication is associative -/
  mul_assoc₃ : ∀ a b c : α, a * b * c = a * (b * c)

class AddMonoid₃ (α : Type) extends AddSemigroup₃ α, AddZeroClass α

@[to_additive AddMonoid₃]
class Monoid₃ (α : Type) extends Semigroup₃ α, MulOneClass α

export Semigroup₃ (mul_assoc₃)
export AddSemigroup₃ (add_assoc₃)

whatsnew in
@[to_additive]
lemma left_inv_eq_right_inv' {M : Type} [Monoid₃ M] {a b c : M} (hba : b * a = 1) (hac␣
↪: a * c = 1) : b = c := by
  rw [← one_mul c, ← hba, mul_assoc₃, hac, mul_one b]

#check left_neg_eq_right_neg'
```

Equipped with this technology, we can easily define also commutative semigroups, monoids and groups, and then define rings.

```
class AddCommSemigroup₃ (α : Type) extends AddSemigroup₃ α where
  add_comm : ∀ a b : α, a + b = b + a
```

(continues on next page)

(continued from previous page)

```
@[to_additive AddCommSemigroup₃]
class CommSemigroup₃ (α : Type) extends Semigroup₃ α where
  mul_comm : ∀ a b : α, a * b = b * a

class AddCommMonoid₃ (α : Type) extends AddMonoid₃ α, AddCommSemigroup₃ α

@[to_additive AddCommMonoid₃]
class CommMonoid₃ (α : Type) extends Monoid₃ α, CommSemigroup₃ α

class AddGroup₃ (G : Type) extends AddMonoid₃ G, Neg G where
  neg_add : ∀ a : G, -a + a = 0

@[to_additive AddGroup₃]
class Group₃ (G : Type) extends Monoid₃ G, Inv G where
  inv_mul : ∀ a : G, a⁻¹ * a = 1
```

We should remember to tag lemmas with `simp` when appropriate.

```
attribute [simp] Group₃.inv_mul AddGroup₃.neg_add
```

Then we need to repeat ourselves a bit since we switch to standard notations, but at least `to_additive` does the work of translating from the multiplicative notation to the additive one.

```
@[to_additive]
lemma inv_eq_of_mul [Group₃ G] {a b : G} (h : a * b = 1) : a⁻¹ = b :=
  sorry
```

Note that `to_additive` can be asked to tag a lemma with `simp` and propagate that attribute to the additive version as follows.

```
@[to_additive (attr := simp)]
lemma Group₃.mul_inv {G : Type} [Group₃ G] {a : G} : a * a⁻¹ = 1 := by
  sorry

@[to_additive]
lemma mul_left_cancel₃ {G : Type} [Group₃ G] {a b c : G} (h : a * b = a * c) : b = c␣
↪:= by
  sorry

@[to_additive]
lemma mul_right_cancel₃ {G : Type} [Group₃ G] {a b c : G} (h : b*a = c*a) : b = c := by
  sorry

class AddCommGroup₃ (G : Type) extends AddGroup₃ G, AddCommMonoid₃ G

@[to_additive AddCommGroup₃]
class CommGroup₃ (G : Type) extends Group₃ G, CommMonoid₃ G
```

We are now ready for rings. For demonstration purposes we won't assume that addition is commutative, and then immediately provide an instance of `AddCommGroup₃`. Mathlib does not play this game, first because in practice this does not make any ring instance easier and also because Mathlib's algebraic hierarchy goes through semirings which are like rings but without opposites so that the proof below does not work for them. What we gain here, besides a nice exercise if you have never seen it, is an example of building an instance using the syntax that allows to provide a parent structure as an instance parameter and then supply the extra fields. Here the *Ring₃ R* argument supplies anything *AddCommGroup₃ R* wants except for *add_comm*.

```
class Ring₃ (R : Type) extends AddGroup₃ R, Monoid₃ R, MulZeroClass R where
  /-- Multiplication is left distributive over addition -/
  left_distrib : ∀ a b c : R, a * (b + c) = a * b + a * c
  /-- Multiplication is right distributive over addition -/
  right_distrib : ∀ a b c : R, (a + b) * c = a * c + b * c

instance {R : Type} [Ring₃ R] : AddCommGroup₃ R :=
{ add_comm := by
    sorry }
```

Of course we can also build concrete instances, such as a ring structure on integers (of course the instance below uses that all the work is already done in Mathlib).

```
instance : Ring₃ ℤ where
  add := (· + ·)
  add_assoc₃ := add_assoc
  zero := 0
  zero_add := by simp
  add_zero := by simp
  neg := (- ·)
  neg_add := by simp
  mul := (· * ·)
  mul_assoc₃ := mul_assoc
  one := 1
  one_mul := by simp
  mul_one := by simp
  zero_mul := by simp
  mul_zero := by simp
  left_distrib := Int.mul_add
  right_distrib := Int.add_mul
```

As an exercise you can now set up a simple hierarchy for order relations, including a class for ordered commutative monoids, which have both a partial order and a commutative monoid structure such that `∀ a b : α, a ≤ b → ∀ c : α, c * a ≤ c * b`. Of course you need to add fields and maybe `extends` clauses to the following classes.

```
class LE₁ (α : Type) where
  /-- The Less-or-Equal relation. -/
  le : α → α → Prop

@[inherit_doc] infix:50 " ≤₁ " => LE₁.le

class Preorder₁ (α : Type)

class PartialOrder₁ (α : Type)

class OrderedCommMonoid₁ (α : Type)

instance : OrderedCommMonoid₁ ℕ where
```

We now want to discuss algebraic structures involving several types. The prime example is modules over rings. If you don't know what is a module, you can pretend it means vector space and think that all our rings are fields. Those structures are commutative additive groups equipped with a scalar multiplication by elements of some ring.

We first define the data-carrying type class of scalar multiplication by some type α on some type β, and give it a right associative notation.

```
class SMul₃ (α : Type) (β : Type) where
  /-- Scalar multiplication -/
  smul : α → β → β

infixr:73 " · " => SMul₃.smul
```

Then we can define modules (again think about vector spaces if you don't know what is a module).

```
class Module₁ (R : Type) [Ring₃ R] (M : Type) [AddCommGroup₃ M] extends SMul₃ R M where
  zero_smul : ∀ m : M, (0 : R) · m = 0
  one_smul : ∀ m : M, (1 : R) · m = m
  mul_smul : ∀ (a b : R) (m : M), (a * b) · m = a · b · m
  add_smul : ∀ (a b : R) (m : M), (a + b) · m = a · m + b · m
  smul_add : ∀ (a : R) (m n : M), a · (m + n) = a · m + a · n
```

There is something interesting going on here. While it isn't too surprising that the ring structure on `R` is a parameter in this definition, you probably expected `AddCommGroup₃ M` to be part of the `extends` clause just as `SMul₃ R M` is. Trying to do that would lead to a mysterious sounding error message: `cannot find synthesization order for instance Module₁.toAddCommGroup₃ with type (R : Type) → [inst : Ring₃ R] → {M : Type} → [self : Module₁ R M] → AddCommGroup₃ M all remaining arguments have metavariables: Ring₃ ?R @Module₁ ?R ?inst✝ M`. In order to understand this message, you need to remember that such an `extends` clause would lead to a field `Module₃.toAddCommGroup₃` marked as an instance. This instance would have the signature appearing in the error message: `(R : Type) → [inst : Ring₃ R] → {M : Type} → [self : Module₁ R M] → AddCommGroup₃ M`. With such an instance in the type class database, each time Lean would look for a `AddCommGroup₃ M` instance for some `M`, it would need to go hunting for a completely unspecified type `R` and a `Ring₃ R` instance before embarking on the main quest of finding a `Module₁ R M` instance. Those two side-quests are represented by the meta-variables mentioned in the error message and denoted by `?R` and `?inst✝` there. Such a `Module₃.toAddCommGroup₃` instance would then be a huge trap for the instance resolution procedure and then `class` command refuses to set it up.

What about `extends SMul₃ R M` then? That one creates a field `Module₁.toSMul₃ : {R : Type} → [inst : Ring₃ R] → {M : Type} → [inst_1 : AddCommGroup₃ M] → [self : Module₁ R M] → SMul₃ R M` whose end result `SMul₃ R M` mentions both `R` and `M` so this field can safely be used as an instance. The rule is easy to remember: each class appearing in the `extends` clause should mention every type appearing in the parameters.

Let us create our first module instance: a ring is a module over itself using its multiplication as a scalar multiplication.

```
instance selfModule (R : Type) [Ring₃ R] : Module₁ R R where
  smul := fun r s ↦ r*s
  zero_smul := zero_mul
  one_smul := one_mul
  mul_smul := mul_assoc₃
  add_smul := Ring₃.right_distrib
  smul_add := Ring₃.left_distrib
```

As a second example, every abelian group is a module over $\mathbb{Z}$ (this is one of the reason to generalize the theory of vector spaces by allowing non-invertible scalars). First one can define scalar multiplication by a natural number for any type equipped with a zero and an addition: `n · a` is defined as `a + ⋯ + a` where `a` appears `n` times. Then this is extended to scalar multiplication by an integer by ensuring `(-1) · a = -a`.

```
def nsmul₁ {M : Type*} [Zero M] [Add M] : ℕ → M → M
  | 0, _ => 0
  | n + 1, a => a + nsmul₁ n a

def zsmul₁ {M : Type*} [Zero M] [Add M] [Neg M] : ℤ → M → M
```

(continues on next page)

(continued from previous page)

```
  | Int.ofNat n, a => nsmul₁ n a
  | Int.negSucc n, a => -nsmul₁ n.succ a
```

Proving this gives rise to a module structure is a bit tedious and not interesting for the current discussion, so we will sorry all axioms. You are *not* asked to replace those sorries with proofs. If you insist on doing it then you will probably want to state and prove several intermediate lemmas about `nsmul₁` and `zsmul₁`.

```
instance abGrpModule (A : Type) [AddCommGroup₃ A] : Module₁ ℤ A where
  smul := zsmul₁
  zero_smul := sorry
  one_smul := sorry
  mul_smul := sorry
  add_smul := sorry
  smul_add := sorry
```

A much more important issue is that we now have two module structures over the ring $\mathbb{Z}$ for $\mathbb{Z}$ itself: `abGrpModule ℤ` since $\mathbb{Z}$ is a abelian group, and `selfModule ℤ` since $\mathbb{Z}$ is a ring. Those two module structure correspond to the same abelian group structure, but it is not obvious that they have the same scalar multiplication. They actually do, but this isn't true by definition, it requires a proof. This is very bad news for the type class instance resolution procedure and will lead to very frustrating failures for users of this hierarchy. When directly asked to find an instance, Lean will pick one, and we can see which one using:

```
#synth Module₁ ℤ ℤ -- abGrpModule ℤ
```

But in a more indirect context it can happen that Lean infers the other one and then gets confused. This situation is known as a bad diamond. This has nothing to do with the diamond operation we used above, it refers to the way one can draw the paths from $\mathbb{Z}$ to its `Module₁ ℤ` going through either `AddCommGroup₃ ℤ` or `Ring₃ ℤ`.

It is important to understand that not all diamonds are bad. In fact there are diamonds everywhere in Mathlib, and also in this chapter. Already at the very beginning we saw one can go from `Monoid₁ α` to `Dia₁ α` through either `Semigroup₁ α` or `DiaOneClass₁ α` and thanks to the work done by the `class` command, the resulting two `Dia₁ α` instances are definitionally equal. In particular a diamond having a `Prop`-valued class at the bottom cannot be bad since any two proofs of the same statement are definitionally equal.

But the diamond we created with modules is definitely bad. The offending piece is the `smul` field which is data, not a proof, and we have two constructions that are not definitionally equal. The robust way of fixing this issue is to make sure that going from a rich structure to a poor structure is always done by forgetting data, not by defining data. This well-known pattern has been named "forgetful inheritance" and extensively discussed in https://inria.hal.science/hal-02463336v2.

In our concrete case, we can modify the definition of `AddMonoid₃` to include a `nsmul` data field and some `Prop`-valued fields ensuring this operation is provably the one we constructed above. Those fields are given default values using `:=` after their type in the definition below. Thanks to these default values, most instances would be constructed exactly as with our previous definitions. But in the special case of $\mathbb{Z}$ we will be able to provide specific values.

```
class AddMonoid₄ (M : Type) extends AddSemigroup₃ M, AddZeroClass M where
  /-- Multiplication by a natural number. -/
  nsmul : ℕ → M → M := nsmul₁
  /-- Multiplication by `(0 : ℕ)` gives `0`. -/
  nsmul_zero : ∀ x, nsmul 0 x = 0 := by intros; rfl
  /-- Multiplication by `(n + 1 : ℕ)` behaves as expected. -/
  nsmul_succ : ∀ (n : ℕ) (x), nsmul (n + 1) x = x + nsmul n x := by intros; rfl

instance mySMul {M : Type} [AddMonoid₄ M] : SMul ℕ M := ⟨AddMonoid₄.nsmul⟩
```

Let us check we can still construct a product monoid instance without providing the `nsmul` related fields.

```
instance (M N : Type) [AddMonoid₄ M] [AddMonoid₄ N] : AddMonoid₄ (M × N) where
  add := fun p q ↦ (p.1 + q.1, p.2 + q.2)
  add_assoc₃ := fun a b c ↦ by ext <;> apply add_assoc₃
  zero := (0, 0)
  zero_add := fun a ↦ by ext <;> apply zero_add
  add_zero := fun a ↦ by ext <;> apply add_zero
```

And now let us handle the special case of $\mathbb{Z}$ where we want to build `nsmul` using the coercion of $\mathbb{N}$ to $\mathbb{Z}$ and the multiplication on $\mathbb{Z}$. Note in particular how the proof fields contain more work than in the default value above.

```
instance : AddMonoid₄ ℤ where
  add := (· + ·)
  add_assoc₃ := Int.add_assoc
  zero := 0
  zero_add := Int.zero_add
  add_zero := Int.add_zero
  nsmul := fun n m ↦ (n : ℤ) * m
  nsmul_zero := Int.zero_mul
  nsmul_succ := fun n m ↦ show (n + 1 : ℤ) * m = m + n * m
    by rw [Int.add_mul, Int.add_comm, Int.one_mul]
```

Let us check we solved our issue. Because Lean already has a definition of scalar multiplication of a natural number and an integer, and we want to make sure our instance is used, we won't use the · notation but call `SMul.mul` and explicitly provide our instance defined above.

```
example (n : ℕ) (m : ℤ) : SMul.smul (self := mySMul) n m = n * m := rfl
```

This story then continues with incorporating a `zsmul` field into the definition of groups and similar tricks. You are now ready to read the definition of monoids, groups, rings and modules in Mathlib. There are more complicated than what we have seen here, because they are part of a huge hierarchy, but all principles have been explained above.

As an exercise, you can come back to the order relation hierarchy you built above and try to incorporate a type class $\texttt{LT}_1$ carrying the Less-Than notation $<_1$ and make sure that every preorder comes with a $<_1$ which has a default value built from $\leq_1$ and a `Prop`-valued field asserting the natural relation between those two comparison operators.

8.2 Morphisms

So far in this chapter, we discussed how to create a hierarchy of mathematical structures. But defining structures is not really completed until we have morphisms. There are two main approaches here. The most obvious one is to define a predicate on functions.

```
def isMonoidHom₁ [Monoid G] [Monoid H] (f : G → H) : Prop :=
  f 1 = 1 ∧ ∀ g g', f (g * g') = f g * f g'
```

In this definition, it is a bit unpleasant to use a conjunction. In particular users will need to remember the ordering we chose when they want to access the two conditions. So we could use a structure instead.

```
structure isMonoidHom₂ [Monoid G] [Monoid H] (f : G → H) : Prop where
  map_one : f 1 = 1
  map_mul : ∀ g g', f (g * g') = f g * f g'
```

Once we are here, it is even tempting to make it a class and use the type class instance resolution procedure to automatically infer $\texttt{isMonoidHom}_2$ for complicated functions out of instances for simpler functions. For instance a composition of monoid morphisms is a monoid morphism and this seems like a useful instance. However such an instance would be very tricky for the resolution procedure since it would need to hunt down `g ∘ f` everywhere. Seeing it failing in `g (f x)`

would be very frustrating. More generally one must always keep in mind that recognizing which function is applied in a given expression is a very difficult problem, called the "higher-order unification problem". So Mathlib does not use this class approach.

A more fundamental question is whether we use predicates as above (using either a `def` or a `structure`) or use structures bundling a function and predicates. This is partly a psychological issue. It is extremely rare to consider a function between monoids that is not a morphism. It really feels like "monoid morphism" is not an adjective you can assign to a bare function, it is a noun. On the other hand one can argue that a continuous function between topological spaces is really a function that happens to be continuous. This is one reason why Mathlib has a `Continuous` predicate. For instance you can write:

```
example : Continuous (id : ℝ → ℝ) := continuous_id
```

We still have bundles of continuous functions, which are convenient for instance to put a topology on a space of continuous functions, but they are not the primary tool to work with continuity.

By contrast, morphisms between monoids (or other algebraic structures) are bundled as in:

```
@[ext]
structure MonoidHom₁ (G H : Type) [Monoid G] [Monoid H]  where
  toFun : G → H
  map_one : toFun 1 = 1
  map_mul : ∀ g g', toFun (g * g') = toFun g * toFun g'
```

Of course we don't want to type `toFun` everywhere so we register a coercion using the `CoeFun` type class. Its first argument is the type we want to coerce to a function. The second argument describes the target function type. In our case it is always `G → H` for every `f : MonoidHom₁ G H`. We also tag `MonoidHom₁.toFun` with the `coe` attribute to make sure it is displayed almost invisibly in the tactic state, simply by a ↑ prefix.

```
instance [Monoid G] [Monoid H] : CoeFun (MonoidHom₁ G H) (fun _ ↦ G → H) where
  coe := MonoidHom₁.toFun

attribute [coe] MonoidHom₁.toFun
```

Let us check we can indeed apply a bundled monoid morphism to an element.

```
example [Monoid G] [Monoid H] (f : MonoidHom₁ G H) : f 1 = 1 :=  f.map_one
```

We can do the same with other kind of morphisms until we reach ring morphisms.

```
@[ext]
structure AddMonoidHom₁ (G H : Type) [AddMonoid G] [AddMonoid H]  where
  toFun : G → H
  map_zero : toFun 0 = 0
  map_add : ∀ g g', toFun (g + g') = toFun g + toFun g'

instance [AddMonoid G] [AddMonoid H] : CoeFun (AddMonoidHom₁ G H) (fun _ ↦ G → H)␣
↪where
  coe := AddMonoidHom₁.toFun

attribute [coe] AddMonoidHom₁.toFun

@[ext]
structure RingHom₁ (R S : Type) [Ring R] [Ring S] extends MonoidHom₁ R S,␣
↪AddMonoidHom₁ R S
```

There are a couple of issues about this approach. A minor one is we don't quite know where to put the `coe` attribute since the `RingHom₁.toFun` does not exist, the relevant function is `MonoidHom₁.toFun ∘ RingHom₁.`

`toMonoidHom₁` which is not a declaration that can be tagged with an attribute (but we could still define a `CoeFun (RingHom₁ R S) (fun _ ↦ R → S)` instance). A much more important one is that lemmas about monoid morphisms won't directly apply to ring morphisms. This leaves the alternative of either juggling with `RingHom₁.toMonoidHom₁` each time we want to apply a monoid morphism lemma or restate every such lemmas for ring morphisms. Neither option is appealing so Mathlib uses a new hierarchy trick here. The idea is to define a type class for objects that are at least monoid morphisms, instantiate that class with both monoid morphisms and ring morphisms and use it to state every lemma. In the definition below, `F` could be `MonoidHom₁ M N`, or `RingHom₁ M N` if `M` and `N` have a ring structure.

```
class MonoidHomClass₁ (F : Type) (M N : Type) [Monoid M] [Monoid N] where
  toFun : F → M → N
  map_one : ∀ f : F, toFun f 1 = 1
  map_mul : ∀ f g g', toFun f (g * g') = toFun f g * toFun f g'
```

However there is a problem with the above implementation. We haven't registered a coercion to function instance yet. Let us try to do it now.

```
def badInst [Monoid M] [Monoid N] [MonoidHomClass₁ F M N] : CoeFun F (fun _ ↦ M → N)␣
↪where
  coe := MonoidHomClass₁.toFun
```

Making this an instance would be bad. When faced with something like `f x` where the type of `f` is not a function type, Lean will try to find a `CoeFun` instance to coerce `f` into a function. The above function has type: `{M N F : Type} → [Monoid M] → [Monoid N] → [MonoidHomClass₁ F M N] → CoeFun F (fun x ↦ M → N)` so, when it trying to apply it, it wouldn't be a priori clear to Lean in which order the unknown types `M`, `N` and `F` should be inferred. This is a kind of bad instance that is slightly different from the one we saw already, but it boils down to the same issue: without knowing `M`, Lean would have to search for a monoid instance on an unknown type, hence hopelessly try *every* monoid instance in the database. If you are curious to see the effect of such an instance you can type `set_option synthInstance.checkSynthOrder false in` on top of the above declaration, replace `def badInst` with `instance`, and look for random failures in this file.

Here the solution is easy, we need to tell Lean to first search what is `F` and then deduce `M` and `N`. This is done using the `outParam` function. This function is defined as the identity function, but is still recognized by the type class machinery and triggers the desired behavior. Hence we can retry defining our class, paying attention to the `outParam` function:

```
class MonoidHomClass₂ (F : Type) (M N : outParam Type) [Monoid M] [Monoid N] where
  toFun : F → M → N
  map_one : ∀ f : F, toFun f 1 = 1
  map_mul : ∀ f g g', toFun f (g * g') = toFun f g * toFun f g'

instance [Monoid M] [Monoid N] [MonoidHomClass₂ F M N] : CoeFun F (fun _ ↦ M → N)␣
↪where
  coe := MonoidHomClass₂.toFun

attribute [coe] MonoidHomClass₂.toFun
```

Now we can proceed with our plan to instantiate this class.

```
instance (M N : Type) [Monoid M] [Monoid N] : MonoidHomClass₂ (MonoidHom₁ M N) M N␣
↪where
  toFun := MonoidHom₁.toFun
  map_one := fun f ↦ f.map_one
  map_mul := fun f ↦ f.map_mul

instance (R S : Type) [Ring R] [Ring S] : MonoidHomClass₂ (RingHom₁ R S) R S where
  toFun := fun f ↦ f.toMonoidHom₁.toFun
```

(continues on next page)

(continued from previous page)

```
  map_one := fun f ↦ f.toMonoidHom₁.map_one
  map_mul := fun f ↦ f.toMonoidHom₁.map_mul
```

As promised every lemma we prove about `f : F` assuming an instance of `MonoidHomClass₁ F` will apply both to monoid morphisms and ring morphisms. Let us see an example lemma and check it applies to both situations.

```
lemma map_inv_of_inv [Monoid M] [Monoid N] [MonoidHomClass₂ F M N] (f : F) {m m' : M}␣
↪(h : m*m' = 1) :
    f m * f m' = 1 := by
  rw [← MonoidHomClass₂.map_mul, h, MonoidHomClass₂.map_one]

example [Monoid M] [Monoid N] (f : MonoidHom₁ M N) {m m' : M} (h : m*m' = 1) : f m * f␣
↪m' = 1 :=
map_inv_of_inv f h

example [Ring R] [Ring S] (f : RingHom₁ R S) {r r' : R} (h : r*r' = 1) : f r * f r' =␣
↪1 :=
map_inv_of_inv f h
```

At first sight, it may look like we got back to our old bad idea of making `MonoidHom₁` a class. But we haven't. Everything is shifted one level of abstraction up. The type class resolution procedure won't be looking for functions, it will be looking for either `MonoidHom₁` or `RingHom₁`.

One remaining issue with our approach is the presence of repetitive code around the `toFun` field and the corresponding `CoeFun` instance and `coe` attribute. It would also be better to record that this pattern is used only for functions with extra properties, meaning that the coercion to functions should be injective. So Mathlib adds one more layer of abstraction with the base class `DFunLike` (where "DFun" stands for dependent function). Let us redefine our `MonoidHomClass` on top of this base layer.

```
class MonoidHomClass₃ (F : Type) (M N : outParam Type) [Monoid M] [Monoid N] extends
    DFunLike F M (fun _ ↦ N) where
  map_one : ∀ f : F, f 1 = 1
  map_mul : ∀ (f : F) g g', f (g * g') = f g * f g'

instance (M N : Type) [Monoid M] [Monoid N] : MonoidHomClass₃ (MonoidHom₁ M N) M N␣
↪where
  coe := MonoidHom₁.toFun
  coe_injective' _ _ := MonoidHom₁.ext
  map_one := MonoidHom₁.map_one
  map_mul := MonoidHom₁.map_mul
```

Of course the hierarchy of morphisms does not stop here. We could go on and define a class `RingHomClass₃` extending `MonoidHomClass₃` and instantiate it on `RingHom` and then later on `AlgebraHom` (algebras are rings with some extra structure). But we've covered the main formalization ideas used in Mathlib for morphisms and you should be ready to understand how morphisms are defined in Mathlib.

As an exercise, you should try to define your class of bundled order-preserving function between ordered types, and then order preserving monoid morphisms. This is for training purposes only. Like continuous functions, order preserving functions are primarily unbundled in Mathlib where they are defined by the `Monotone` predicate. Of course you need to complete the class definitions below.

```
@[ext]
structure OrderPresHom (α β : Type) [LE α] [LE β] where
  toFun : α → β
  le_of_le : ∀ a a', a ≤ a' → toFun a ≤ toFun a'
```

(continues on next page)

(continued from previous page)

```
@[ext]
structure OrderPresMonoidHom (M N : Type) [Monoid M] [LE M] [Monoid N] [LE N] extends
MonoidHom₁ M N, OrderPresHom M N

class OrderPresHomClass (F : Type) (α β : outParam Type) [LE α] [LE β]

instance (α β : Type) [LE α] [LE β] : OrderPresHomClass (OrderPresHom α β) α β where

instance (α β : Type) [LE α] [Monoid α] [LE β] [Monoid β] :
    OrderPresHomClass (OrderPresMonoidHom α β) α β where

instance (α β : Type) [LE α] [Monoid α] [LE β] [Monoid β] :
    MonoidHomClass₃ (OrderPresMonoidHom α β) α β
  := sorry
```

8.3 Sub-objects

After defining some algebraic structure and its morphisms, the next step is to consider sets that inherit this algebraic structure, for instance subgroups or subrings. This largely overlaps with our previous topic. Indeed a set in `X` is implemented as a function from `X` to `Prop` so sub-objects are function satisfying a certain predicate. Hence we can reuse of lot of the ideas that led to the `DFunLike` class and its descendants. We won't reuse `DFunLike` itself because this would break the abstraction barrier from `Set X` to `X → Prop`. Instead there is a `SetLike` class. Instead of wrapping an injection into a function type, that class wraps an injection into a `Set` type and defines the corresponding coercion and `Membership` instance.

```
@[ext]
structure Submonoid₁ (M : Type) [Monoid M] where
  /-- The carrier of a submonoid. -/
  carrier : Set M
  /-- The product of two elements of a submonoid belongs to the submonoid. -/
  mul_mem {a b} : a ∈ carrier → b ∈ carrier → a * b ∈ carrier
  /-- The unit element belongs to the submonoid. -/
  one_mem : 1 ∈ carrier

/-- Submonoids in `M` can be seen as sets in `M`. -/
instance [Monoid M] : SetLike (Submonoid₁ M) M where
  coe := Submonoid₁.carrier
  coe_injective' _ _ := Submonoid₁.ext
```

Equipped with the above `SetLike` instance, we can already state naturally that a submonoid `N` contains `1` without using `N.carrier`. We can also silently treat `N` as a set in `M` as take its direct image under a map.

```
example [Monoid M] (N : Submonoid₁ M) : 1 ∈ N := N.one_mem

example [Monoid M] (N : Submonoid₁ M) (α : Type) (f : M → α) := f '' N
```

We also have a coercion to `Type` which uses `Subtype` so, given a submonoid `N` we can write a parameter `(x : N)` which can be coerced to an element of `M` belonging to `N`.

```
example [Monoid M] (N : Submonoid₁ M) (x : N) : (x : M) ∈ N := x.property
```

Using this coercion to `Type` we can also tackle the task of equipping a submonoid with a monoid structure. We will use the coercion from the type associated to `N` as above, and the lemma `SetCoe.ext` asserting this coercion is injective. Both are provided by the `SetLike` instance.

```
instance SubMonoid₁Monoid [Monoid M] (N : Submonoid₁ M) : Monoid N where
  mul := fun x y ↦ ⟨x*y, N.mul_mem x.property y.property⟩
  mul_assoc := fun x y z ↦ SetCoe.ext (mul_assoc (x : M) y z)
  one := ⟨1, N.one_mem⟩
  one_mul := fun x ↦ SetCoe.ext (one_mul (x : M))
  mul_one := fun x ↦ SetCoe.ext (mul_one (x : M))
```

Note that, in the above instance, instead of using the coercion to `M` and calling the `property` field, we could have used destructuring binders as follows.

```
example [Monoid M] (N : Submonoid₁ M) : Monoid N where
  mul := fun ⟨x, hx⟩ ⟨y, hy⟩ ↦ ⟨x*y, N.mul_mem hx hy⟩
  mul_assoc := fun ⟨x, _⟩ ⟨y, _⟩ ⟨z, _⟩ ↦ SetCoe.ext (mul_assoc x y z)
  one := ⟨1, N.one_mem⟩
  one_mul := fun ⟨x, _⟩ ↦ SetCoe.ext (one_mul x)
  mul_one := fun ⟨x, _⟩ ↦ SetCoe.ext (mul_one x)
```

In order to apply lemmas about submonoids to subgroups or subrings, we need a class, just like for morphisms. Note this class take a `SetLike` instance as a parameter so it does not need a carrier field and can use the membership notation in its fields.

```
class SubmonoidClass₁ (S : Type) (M : Type) [Monoid M] [SetLike S M] : Prop where
  mul_mem : ∀ (s : S) {a b : M}, a ∈ s → b ∈ s → a * b ∈ s
  one_mem : ∀ s : S, 1 ∈ s

instance [Monoid M] : SubmonoidClass₁ (Submonoid₁ M) M where
  mul_mem := Submonoid₁.mul_mem
  one_mem := Submonoid₁.one_mem
```

As an exercise you should define a `Subgroup₁` structure, endow it with a `SetLike` instance and a `SubmonoidClass₁` instance, put a `Group` instance on the subtype associated to a `Subgroup₁` and define a `SubgroupClass₁` class.

Another very important thing to know about subobjects of a given algebraic object in Mathlib always form a complete lattice, and this structure is used a lot. For instance you may look for the lemma saying that an intersection of submonoids is a submonoid. But this won't be a lemma, this will be an infimum construction. Let us do the case of two submonoids.

```
instance [Monoid M] : Min (Submonoid₁ M) :=
  ⟨fun S₁ S₂ ↦
    { carrier := S₁ ∩ S₂
      one_mem := ⟨S₁.one_mem, S₂.one_mem⟩
      mul_mem := fun ⟨hx, hx'⟩ ⟨hy, hy'⟩ ↦ ⟨S₁.mul_mem hx hy, S₂.mul_mem hx' hy'⟩ }⟩
```

This allows to get the intersections of two submonoids as a submonoid.

```
example [Monoid M] (N P : Submonoid₁ M) : Submonoid₁ M := N ⊓ P
```

You may think it's a shame that we had to use the inf symbol ⊓ in the above example instead of the intersection symbol ∩. But think about the supremum. The union of two submonoids is not a submonoid. However submonoids still form a lattice (even a complete one). Actually `N ⊔ P` is the submonoid generated by the union of `N` and `P` and of course it would be very confusing to denote it by `N ∪ P`. So you can see the use of `N ⊓ P` as much more consistent. It is also a lot more consistent across various kind of algebraic structures. It may look a bit weird at first to see the sum of two vector subspace `E` and `F` denoted by `E ⊔ F` instead of `E + F`. But you will get used to it. And soon you will consider the `E + F` notation as a distraction emphasizing the anecdotal fact that elements of `E ⊔ F` can be written as a sum of an element of `E` and an element of `F` instead of emphasizing the fundamental fact that `E ⊔ F` is the smallest vector subspace containing both `E` and `F`.

Our last topic for this chapter is that of quotients. Again we want to explain how convenient notation are built and code duplication is avoided in Mathlib. Here the main device is the `HasQuotient` class which allows notations like `M ⧸ N`. Beware the quotient symbol `⧸` is a special unicode character, not a regular ASCII division symbol.

As an example, we will build the quotient of a commutative monoid by a submonoid, leave proofs to you. In the last example, you can use `Setoid.refl` but it won't automatically pick up the relevant `Setoid` structure. You can fix this issue by providing all arguments using the `@` syntax, as in `@Setoid.refl M N.Setoid`.

```
def Submonoid.Setoid [CommMonoid M] (N : Submonoid M) : Setoid M  where
  r := fun x y ↦ ∃ w ∈ N, ∃ z ∈ N, x*w = y*z
  iseqv := {
    refl := fun x ↦ ⟨1, N.one_mem, 1, N.one_mem, rfl⟩
    symm := fun ⟨w, hw, z, hz, h⟩ ↦ ⟨z, hz, w, hw, h.symm⟩
    trans := by
      sorry
  }

instance [CommMonoid M] : HasQuotient M (Submonoid M) where
  quotient' := fun N ↦ Quotient N.Setoid

def QuotientMonoid.mk [CommMonoid M] (N : Submonoid M) : M → M ⧸ N := Quotient.mk N.
↪Setoid

instance [CommMonoid M] (N : Submonoid M) : Monoid (M ⧸ N) where
  mul := Quotient.map₂ (· * ·) (by
      sorry
        )
  mul_assoc := by
      sorry
  one := QuotientMonoid.mk N 1
  one_mul := by
      sorry
  mul_one := by
      sorry
```

CHAPTER

NINE

GROUPS AND RINGS

We saw in Section 2.2 how to reason about operations in groups and rings. Later, in Section 7.2, we saw how to define abstract algebraic structures, such as group structures, as well as concrete instances such as the ring structure on the Gaussian integers. Chapter 8 explained how hierarchies of abstract structures are handled in Mathlib.

In this chapter we work with groups and rings in more detail. We won't be able to cover every aspect of the treatment of these topics in Mathlib, especially since Mathlib is constantly growing. But we will provide entry points to the library and show how the essential concepts are used. There is some overlap with the discussion of Chapter 8, but here we will focus on how to use Mathlib instead of the design decisions behind the way the topics are treated. So making sense of some of the examples may require reviewing the background from Chapter 8.

9.1 Monoids and Groups

9.1.1 Monoids and their morphisms

Courses in abstract algebra often start with groups and then progress to rings, fields, and vector spaces. This involves some contortions when discussing multiplication on rings since the multiplication operation does not come from a group structure but many of the proofs carry over verbatim from group theory to this new setting. The most common fix, when doing mathematics with pen and paper, is to leave those proofs as exercises. A less efficient but safer and more formalization-friendly way of proceeding is to use monoids. A *monoid* structure on a type M is an internal composition law that is associative and has a neutral element. Monoids are used primarily to accommodate both groups and the multiplicative structure of rings. But there are also a number of natural examples; for instance, the set of natural numbers equipped with addition forms a monoid.

From a practical point of view, you can mostly ignore monoids when using Mathlib. But you need to know they exist when you are looking for a lemma by browsing Mathlib files. Otherwise, you might end up looking for a statement in the group theory files when it is actually in the found with monoids because it does not require elements to be invertible.

The type of monoid structures on a type `M` is written `Monoid M`. The function `Monoid` is a type class so it will almost always appear as an instance implicit argument (in other words, in square brackets). By default, `Monoid` uses multiplicative notation for the operation; for additive notation use `AddMonoid` instead. The commutative versions of these structures add the prefix `Comm` before `Monoid`.

```
example {M : Type*} [Monoid M] (x : M) : x * 1 = x := mul_one x

example {M : Type*} [AddCommMonoid M] (x y : M) : x + y = y + x := add_comm x y
```

Note that although `AddMonoid` is found in the library, it is generally confusing to use additive notation with a non-commutative operation.

The type of morphisms between monoids `M` and `N` is called `MonoidHom M N` and written `M →* N`. Lean will automatically see such a morphism as a function from `M` to `N` when we apply it to elements of `M`. The additive version is

called `AddMonoidHom` and written `M →+ N`.

```
example {M N : Type*} [Monoid M] [Monoid N] (x y : M) (f : M →* N) : f (x * y) = f x␣
↪* f y :=
  f.map_mul x y

example {M N : Type*} [AddMonoid M] [AddMonoid N] (f : M →+ N) : f 0 = 0 :=
  f.map_zero
```

These morphisms are bundled maps, i.e. they package together a map and some of its properties. Remember that Section 8.2 explains bundled maps; here we simply note the slightly unfortunate consequence that we cannot use ordinary function composition to compose maps. Instead, we need to use `MonoidHom.comp` and `AddMonoidHom.comp`.

```
example {M N P : Type*} [AddMonoid M] [AddMonoid N] [AddMonoid P]
    (f : M →+ N) (g : N →+ P) : M →+ P := g.comp f
```

9.1.2 Groups and their morphisms

We will have much more to say about groups, which are monoids with the extra property that every element has an inverse.

```
example {G : Type*} [Group G] (x : G) : x * x⁻¹ = 1 := mul_inv_cancel x
```

Similar to the `ring` tactic that we saw earlier, there is a `group` tactic that proves any identity that holds in any group. (Equivalently, it proves the identities that hold in free groups.)

```
example {G : Type*} [Group G] (x y z : G) : x * (y * z) * (x * z)⁻¹ * (x * y * x⁻¹)⁻¹ =␣
↪1 := by
  group
```

There is also a tactic for identities in commutative additive groups called `abel`.

```
example {G : Type*} [AddCommGroup G] (x y z : G) : z + x + (y - z - x) = y := by
  abel
```

Interestingly, a group morphism is nothing more than a monoid morphism between groups. So we can copy and paste one of our earlier examples, replacing `Monoid` with `Group`.

```
example {G H : Type*} [Group G] [Group H] (x y : G) (f : G →* H) : f (x * y) = f x *␣
↪f y :=
  f.map_mul x y
```

Of course we do get some new properties, such as this one:

```
example {G H : Type*} [Group G] [Group H] (x : G) (f : G →* H) : f (x⁻¹) = (f x)⁻¹ :=
  f.map_inv x
```

You may be worried that constructing group morphisms will require us to do unnecessary work since the definition of monoid morphism enforces that neutral elements are sent to neutral elements while this is automatic in the case of group morphisms. In practice the extra work is not hard, but, to avoid it, there is a function building a group morphism from a function between groups that is compatible with the composition laws.

```
example {G H : Type*} [Group G] [Group H] (f : G → H) (h : ∀ x y, f (x * y) = f x * f␣
↪y) :
    G →* H :=
  MonoidHom.mk' f h
```

There is also a type `MulEquiv` of group (or monoid) isomorphisms denoted by `≃*` (and `AddEquiv` denoted by `≃+` in additive notation). The inverse of `f : G ≃* H` is `MulEquiv.symm f : H ≃* G`, composition of `f` and `g` is `MulEquiv.trans f g`, and the identity isomorphism of `G` is `MulEquiv.refl G`. Using anonymous projector notation, the first two can be written `f.symm` and `f.trans g` respectively. Elements of this type are automatically coerced to morphisms and functions when necessary.

```
example {G H : Type*} [Group G] [Group H] (f : G ≃* H) :
    f.trans f.symm = MulEquiv.refl G :=
  f.self_trans_symm
```

One can use `MulEquiv.ofBijective` to build an isomorphism from a bijective morphism. Doing so makes the inverse function noncomputable.

```
noncomputable example {G H : Type*} [Group G] [Group H]
    (f : G →* H) (h : Function.Bijective f) :
    G ≃* H :=
  MulEquiv.ofBijective f h
```

9.1.3 Subgroups

Just as group morphisms are bundled, a subgroup of `G` is also a bundled structure consisting of a set in `G` with the relevant closure properties.

```
example {G : Type*} [Group G] (H : Subgroup G) {x y : G} (hx : x ∈ H) (hy : y ∈ H) :
    x * y ∈ H :=
  H.mul_mem hx hy

example {G : Type*} [Group G] (H : Subgroup G) {x : G} (hx : x ∈ H) :
    x⁻¹ ∈ H :=
  H.inv_mem hx
```

In the example above, it is important to understand that `Subgroup G` is the type of subgroups of `G`, rather than a predicate `IsSubgroup H` where `H` is an element of `Set G`. `Subgroup G` is endowed with a coercion to `Set G` and a membership predicate on `G`. See Section 8.3 for an explanation of how and why this is done.

Of course, two subgroups are the same if and only if they have the same elements. This fact is registered for use with the `ext` tactic, which can be used to prove two subgroups are equal in the same way it is used to prove that two sets are equal.

To state and prove, for example, that $\mathbb{Z}$ is an additive subgroup of $\mathbb{Q}$, what we really want is to construct a term of type `AddSubgroup ℚ` whose projection to `Set ℚ` is $\mathbb{Z}$, or, more precisely, the image of $\mathbb{Z}$ in $\mathbb{Q}$.

```
example : AddSubgroup ℚ where
  carrier := Set.range ((↑) : ℤ → ℚ)
  add_mem' := by
    rintro _ _ ⟨n, rfl⟩ ⟨m, rfl⟩
    use n + m
    simp
  zero_mem' := by
    use 0
    simp
  neg_mem' := by
    rintro _ ⟨n, rfl⟩
    use -n
    simp
```

Using type classes, Mathlib knows that a subgroup of a group inherits a group structure.

```
example {G : Type*} [Group G] (H : Subgroup G) : Group H := inferInstance
```

This example is subtle. The object `H` is not a type, but Lean automatically coerces it to a type by interpreting it as a subtype of `G`. So the above example can be restated more explicitly as:

```
example {G : Type*} [Group G] (H : Subgroup G) : Group {x : G // x ∈ H} :=␣
↪inferInstance
```

An important benefit of having a type `Subgroup G` instead of a predicate `IsSubgroup : Set G → Prop` is that one can easily endow `Subgroup G` with additional structure. Importantly, it has the structure of a complete lattice structure with respect to inclusion. For instance, instead of having a lemma stating that an intersection of two subgroups of `G` is again a subgroup, we have used the lattice operation `⊓` to construct the intersection. We can then apply arbitrary lemmas about lattices to the construction.

Let us check that the set underlying the infimum of two subgroups is indeed, by definition, their intersection.

```
example {G : Type*} [Group G] (H H' : Subgroup G) :
    ((H ⊓ H' : Subgroup G) : Set G) = (H : Set G) ∩ (H' : Set G) := rfl
```

It may look strange to have a different notation for what amounts to the intersection of the underlying sets, but the correspondence does not carry over to the supremum operation and set union, since a union of subgroups is not, in general, a subgroup. Instead one needs to use the subgroup generated by the union, which is done using `Subgroup.closure`.

```
example {G : Type*} [Group G] (H H' : Subgroup G) :
    ((H ⊔ H' : Subgroup G) : Set G) = Subgroup.closure ((H : Set G) ∪ (H' : Set G))␣
↪:= by
  rw [Subgroup.sup_eq_closure]
```

Another subtlety is that `G` itself does not have type `Subgroup G`, so we need a way to talk about `G` seen as a subgroup of `G`. This is also provided by the lattice structure: the full subgroup is the top element of this lattice.

```
example {G : Type*} [Group G] (x : G) : x ∈ (⊤ : Subgroup G) := trivial
```

Similarly the bottom element of this lattice is the subgroup whose only element is the neutral element.

```
example {G : Type*} [Group G] (x : G) : x ∈ (⊥ : Subgroup G) ↔ x = 1 := Subgroup.mem_
↪bot
```

As an exercise in manipulating groups and subgroups, you can define the conjugate of a subgroup by an element of the ambient group.

```
def conjugate {G : Type*} [Group G] (x : G) (H : Subgroup G) : Subgroup G where
  carrier := {a : G | ∃ h, h ∈ H ∧ a = x * h * x⁻¹}
  one_mem' := by
    dsimp
    sorry
  inv_mem' := by
    dsimp
    sorry
  mul_mem' := by
    dsimp
    sorry
```

Tying the previous two topics together, one can push forward and pull back subgroups using group morphisms. The naming convention in Mathlib is to call those operations `map` and `comap`. These are not the common mathematical terms, but they have the advantage of being shorter than "pushforward" and "direct image."

```
example {G H : Type*} [Group G] [Group H] (G' : Subgroup G) (f : G →* H) : Subgroup H :=
  Subgroup.map f G'

example {G H : Type*} [Group G] [Group H] (H' : Subgroup H) (f : G →* H) : Subgroup G :=
  Subgroup.comap f H'

#check Subgroup.mem_map
#check Subgroup.mem_comap
```

In particular, the preimage of the bottom subgroup under a morphism `f` is a subgroup called the *kernel* of `f`, and the range of `f` is also a subgroup.

```
example {G H : Type*} [Group G] [Group H] (f : G →* H) (g : G) :
    g ∈ MonoidHom.ker f ↔ f g = 1 :=
  f.mem_ker

example {G H : Type*} [Group G] [Group H] (f : G →* H) (h : H) :
    h ∈ MonoidHom.range f ↔ ∃ g : G, f g = h :=
  f.mem_range
```

As exercises in manipulating group morphisms and subgroups, let us prove some elementary properties. They are already proved in Mathlib, so do not use `exact?` too quickly if you want to benefit from these exercises.

```
section exercises
variable {G H : Type*} [Group G] [Group H]

open Subgroup

example (φ : G →* H) (S T : Subgroup H) (hST : S ≤ T) : comap φ S ≤ comap φ T := by
  sorry

example (φ : G →* H) (S T : Subgroup G) (hST : S ≤ T) : map φ S ≤ map φ T := by
  sorry

variable {K : Type*} [Group K]

-- Remember you can use the `ext` tactic to prove an equality of subgroups.
example (φ : G →* H) (ψ : H →* K) (U : Subgroup K) :
    comap (ψ.comp φ) U = comap φ (comap ψ U) := by
  sorry

-- Pushing a subgroup along one homomorphism and then another is equal to
-- pushing it forward along the composite of the homomorphisms.
example (φ : G →* H) (ψ : H →* K) (S : Subgroup G) :
    map (ψ.comp φ) S = map ψ (S.map φ) := by
  sorry

end exercises
```

Let us finish this introduction to subgroups in Mathlib with two very classical results. Lagrange theorem states the cardinality of a subgroup of a finite group divides the cardinality of the group. Sylow's first theorem is a famous partial converse to Lagrange's theorem.

While this corner of Mathlib is partly set up to allow computation, we can tell Lean to use nonconstructive logic anyway using the following `open scoped` command.

```
open scoped Classical


example {G : Type*} [Group G] (G' : Subgroup G) : Nat.card G' ∣ Nat.card G :=
  ⟨G'.index, mul_comm G'.index _ ▶ G'.index_mul_card.symm⟩

open Subgroup

example {G : Type*} [Group G] [Finite G] (p : ℕ) {n : ℕ} [Fact p.Prime]
    (hdvd : p ^ n ∣ Nat.card G) : ∃ K : Subgroup G, Nat.card K = p ^ n :=
  Sylow.exists_subgroup_card_pow_prime p hdvd
```

The next two exercises derive a corollary of Lagrange's lemma. (This is also already in Mathlib, so do not use `exact?` too quickly.)

```
lemma eq_bot_iff_card {G : Type*} [Group G] {H : Subgroup G} :
    H = ⊥ ↔ Nat.card H = 1 := by
  suffices (∀ x ∈ H, x = 1) ↔ ∃ x ∈ H, ∀ a ∈ H, a = x by
    simpa [eq_bot_iff_forall, Nat.card_eq_one_iff_exists]
  sorry

#check card_dvd_of_le

lemma inf_bot_of_coprime {G : Type*} [Group G] (H K : Subgroup G)
    (h : (Nat.card H).Coprime (Nat.card K)) : H ⊓ K = ⊥ := by
  sorry
```

9.1.4 Concrete groups

One can also manipulate concrete groups in Mathlib, although this is typically more complicated than working with the abstract theory. For instance, given any type `X`, the group of permutations of `X` is `Equiv.Perm X`. In particular the symmetric group $\mathfrak{S}_n$ is `Equiv.Perm (Fin n)`. One can state abstract results about this group, for instance saying that `Equiv.Perm X` is generated by cycles if `X` is finite.

```
open Equiv

example {X : Type*} [Finite X] : Subgroup.closure {σ : Perm X | Perm.IsCycle σ} = ⊤ :=
  Perm.closure_isCycle
```

One can be fully concrete and compute actual products of cycles. Below we use the `#simp` command, which calls the `simp` tactic on a given expression. The notation `c[]` is used to define a cyclic permutation. In the example, the result is a permutation of $\mathbb{N}$. One could use a type ascription such as `(1 : Fin 5)` on the first number appearing to make it a computation in `Perm (Fin 5)`.

```
#simp [mul_assoc] c[1, 2, 3] * c[2, 3, 4]
```

Another way to work with concrete groups is to use free groups and group presentations. The free group on a type α is `FreeGroup α` and the inclusion map is `FreeGroup.of : α → FreeGroup α`. For instance let us define a type `S` with three elements denoted by `a`, `b` and `c`, and the element ab^{-1} of the corresponding free group.

```
section FreeGroup

inductive S | a | b | c

open S
```

(continues on next page)

(continued from previous page)

```
def myElement : FreeGroup S := (.of a) * (.of b)⁻¹
```

Note that we gave the expected type of the definition so that Lean knows that `.of` means `FreeGroup.of`.

The universal property of free groups is embodied as the equivalence `FreeGroup.lift`. For example, let us define the group morphism from `FreeGroup S` to `Perm (Fin 5)` that sends `a` to `c[1, 2, 3]`, `b` to `c[2, 3, 1]`, and `c` to `c[2, 3]`,

```
def myMorphism : FreeGroup S →* Perm (Fin 5) :=
  FreeGroup.lift fun | .a => c[1, 2, 3]
                     | .b => c[2, 3, 1]
                     | .c => c[2, 3]
```

As a last concrete example, let us see how to define a group generated by a single element whose cube is one (so that group will be isomorphic to $\mathbb{Z}/3$) and build a morphism from that group to `Perm (Fin 5)`.

As a type with exactly one element, we will use `Unit` whose only element is denoted by `()`. The function `PresentedGroup` takes a set of relations, i.e. a set of elements of some free group, and returns a group that is this free group quotiented by a normal subgroup generated by relations. (We will see how to handle more general quotients in Section 9.1.6.) Since we somehow hide this behind a definition, we use `deriving Group` to force creation of a group instance on `myGroup`.

```
def myGroup := PresentedGroup {.of () ^ 3} deriving Group
```

The universal property of presented groups ensures that morphisms out of this group can be built from functions that send the relations to the neutral element of the target group. So we need such a function and a proof that the condition holds. Then we can feed this proof to `PresentedGroup.toGroup` to get the desired group morphism.

```
def myMap : Unit → Perm (Fin 5)
| () => c[1, 2, 3]

lemma compat_myMap :
    ∀ r ∈ ({.of () ^ 3} : Set (FreeGroup Unit)), FreeGroup.lift myMap r = 1 := by
  rintro _ rfl
  simp
  decide

def myNewMorphism : myGroup →* Perm (Fin 5) := PresentedGroup.toGroup compat_myMap

end FreeGroup
```

9.1.5 Group actions

One important way that group theory interacts with the rest of mathematics is through the use of group actions. An action of a group `G` on some type `X` is nothing more than a morphism from `G` to `Equiv.Perm X`. So in a sense group actions are already covered by the previous discussion. But we don't want to carry this morphism around; instead, we want it to be inferred automatically by Lean as much as possible. So we have a type class for this, which is `MulAction G X`. The downside of this setup is that having multiple actions of the same group on the same type requires some contortions, such as defining type synonyms, each of which carries different type class instances.

This allows us in particular to use `g • x` to denote the action of a group element `g` on a point `x`.

```
noncomputable section GroupActions

example {G X : Type*} [Group G] [MulAction G X] (g g': G) (x : X) :
    g • (g' • x) = (g * g') • x :=
  (mul_smul g g' x).symm
```

There is also a version for additive group called `AddAction`, where the action is denoted by $+_v$. This is used for instance in the definition of affine spaces.

```
example {G X : Type*} [AddGroup G] [AddAction G X] (g g' : G) (x : X) :
    g +ᵥ (g' +ᵥ x) = (g + g') +ᵥ x :=
  (add_vadd g g' x).symm
```

The underlying group morphism is called `MulAction.toPermHom`.

```
open MulAction

example {G X : Type*} [Group G] [MulAction G X] : G →* Equiv.Perm X :=
  toPermHom G X
```

As an illustration let us see how to define the Cayley isomorphism embedding of any group `G` into a permutation group, namely `Perm G`.

```
def CayleyIsoMorphism (G : Type*) [Group G] : G ≃* (toPermHom G G).range :=
  Equiv.Perm.subgroupOfMulAction G G
```

Note that nothing before the above definition required having a group rather than a monoid (or any type endowed with a multiplication operation really).

The group condition really enters the picture when we will want to partition `X` into orbits. The corresponding equivalence relation on `X` is called `MulAction.orbitRel`. It is not declared as a global instance.

```
example {G X : Type*} [Group G] [MulAction G X] : Setoid X := orbitRel G X
```

Using this we can state that `X` is partitioned into orbits under the action of `G`. More precisely, we get a bijection between `X` and the dependent product `(ω : orbitRel.Quotient G X) × (orbit G (Quotient.out' ω))` where `Quotient.out' ω` simply chooses an element that projects to ω. Recall that elements of this dependent product are pairs $\langle \omega,$ `x`$\rangle$ where the type `orbit G (Quotient.out' ω)` of `x` depends on ω.

```
example {G X : Type*} [Group G] [MulAction G X] :
    X ≃ (ω : orbitRel.Quotient G X) × (orbit G (Quotient.out ω)) :=
  MulAction.selfEquivSigmaOrbits G X
```

In particular, when X is finite, this can be combined with `Fintype.card_congr` and `Fintype.card_sigma` to deduce that the cardinality of `X` is the sum of the cardinalities of the orbits. Furthermore, the orbits are in bijection with the quotient of `G` under the action of the stabilizers by left translation. This action of a subgroup by left-translation is used to define quotients of a group by a subgroup with notation `/` so we can use the following concise statement.

```
example {G X : Type*} [Group G] [MulAction G X] (x : X) :
    orbit G x ≃ G ⧸ stabilizer G x :=
  MulAction.orbitEquivQuotientStabilizer G x
```

An important special case of combining the above two results is when `X` is a group `G` equipped with the action of a subgroup `H` by translation. In this case all stabilizers are trivial so every orbit is in bijection with `H` and we get:

```
example {G : Type*} [Group G] (H : Subgroup G) : G ≃ (G ⧸ H) × H :=
  groupEquivQuotientProdSubgroup
```

This is the conceptual variant of the version of Lagrange theorem that we saw above. Note this version makes no finiteness assumption.

As an exercise for this section, let us build the action of a group on its subgroup by conjugation, using our definition of `conjugate` from a previous exercise.

```
variable {G : Type*} [Group G]

lemma conjugate_one (H : Subgroup G) : conjugate 1 H = H := by
  sorry

instance : MulAction G (Subgroup G) where
  smul := conjugate
  one_smul := by
    sorry
  mul_smul := by
    sorry

end GroupActions
```

9.1.6 Quotient groups

In the above discussion of subgroups acting on groups, we saw the quotient `G ⧸ H` appear. In general this is only a type. It can be endowed with a group structure such that the quotient map is a group morphism if and only if `H` is a normal subgroup (and this group structure is then unique).

The normality assumption is a type class `Subgroup.Normal` so that type class inference can use it to derive the group structure on the quotient.

```
noncomputable section QuotientGroup

example {G : Type*} [Group G] (H : Subgroup G) [H.Normal] : Group (G ⧸ H) :=␣
↪inferInstance

example {G : Type*} [Group G] (H : Subgroup G) [H.Normal] : G →* G ⧸ H :=
  QuotientGroup.mk' H
```

The universal property of quotient groups is accessed through `QuotientGroup.lift`: a group morphism φ descends to `G ⧸ N` as soon as its kernel contains `N`.

```
example {G : Type*} [Group G] (N : Subgroup G) [N.Normal] {M : Type*}
    [Group M] (φ : G →* M) (h : N ≤ MonoidHom.ker φ) : G ⧸ N →* M :=
  QuotientGroup.lift N φ h
```

The fact that the target group is called `M` is the above snippet is a clue that having a monoid structure on `M` would be enough.

An important special case is when `N = ker φ`. In that case the descended morphism is injective and we get a group isomorphism onto its image. This result is often called the first isomorphism theorem.

```
example {G : Type*} [Group G] {M : Type*} [Group M] (φ : G →* M) :
    G ⧸ MonoidHom.ker φ →* MonoidHom.range φ :=
  QuotientGroup.quotientKerEquivRange φ
```

Applying the universal property to a composition of a morphism `φ : G →* G'` with a quotient group projection `Quotient.mk' N'`, we can also aim for a morphism from `G ⧸ N` to `G' ⧸ N'`. The condition required on φ is

usually formulated by saying "φ should send N inside N'." But this is equivalent to asking that φ should pull N' back over N, and the latter condition is nicer to work with since the definition of pullback does not involve an existential quantifier.

```
example {G G': Type*} [Group G] [Group G']
    {N : Subgroup G} [N.Normal] {N' : Subgroup G'} [N'.Normal]
    {φ : G →* G'} (h : N ≤ Subgroup.comap φ N') : G ⧸ N →* G' ⧸ N':=
  QuotientGroup.map N N' φ h
```

One subtle point to keep in mind is that the type G ⧸ N really depends on N (up to definitional equality), so having a proof that two normal subgroups N and M are equal is not enough to make the corresponding quotients equal. However the universal properties does give an isomorphism in this case.

```
example {G : Type*} [Group G] {M N : Subgroup G} [M.Normal]
    [N.Normal] (h : M = N) : G ⧸ M ≃* G ⧸ N := QuotientGroup.quotientMulEquivOfEq h
```

As a final series of exercises for this section, we will prove that if H and K are disjoint normal subgroups of a finite group G such that the product of their cardinalities is equal to the cardinality of G then G is isomorphic to H × K. Recall that disjoint in this context means H ⊓ K = ⊥.

We start with playing a bit with Lagrange's lemma, without assuming the subgroups are normal or disjoint.

```
section
variable {G : Type*} [Group G] {H K : Subgroup G}

open MonoidHom

#check Nat.card_pos -- The nonempty argument will be automatically inferred for␣
↪subgroups
#check Subgroup.index_eq_card
#check Subgroup.index_mul_card
#check Nat.eq_of_mul_eq_mul_right

lemma aux_card_eq [Finite G] (h' : Nat.card G = Nat.card H * Nat.card K) :
    Nat.card (G ⧸ H) = Nat.card K := by
  sorry
```

From now on, we assume that our subgroups are normal and disjoint, and we assume the cardinality condition. Now we construct the first building block of the desired isomorphism.

```
variable [H.Normal] [K.Normal] [Fintype G] (h : Disjoint H K)
  (h' : Nat.card G = Nat.card H * Nat.card K)

#check Nat.bijective_iff_injective_and_card
#check ker_eq_bot_iff
#check restrict
#check ker_restrict

def iso₁ : K ≃* G ⧸ H := by
  sorry
```

Now we can define our second building block. We will need MonoidHom.prod, which builds a morphism from G_0 to $G_1 \times G_2$ out of morphisms from G_0 to G_1 and G_2.

```
def iso₂ : G ≃* (G ⧸ K) × (G ⧸ H) := by
  sorry
```

We are ready to put all pieces together.

```
#check MulEquiv.prodCongr

def finalIso : G ≃* H × K :=
  sorry
```

9.2 Rings

9.2.1 Rings, their units, morphisms and subrings

The type of ring structures on a type `R` is `Ring R`. The variant where multiplication is assumed to be commutative is `CommRing R`. We have already seen that the `ring` tactic will prove any equality that follows from the axioms of a commutative ring.

```
example {R : Type*} [CommRing R] (x y : R) : (x + y) ^ 2 = x ^ 2 + y ^ 2 + 2 * x * y␣
↪:= by ring
```

More exotic variants do not require that the addition on `R` forms a group but only an additive monoid. The corresponding type classes are `Semiring R` and `CommSemiring R`. The type of natural numbers is an important instance of `CommSemiring R`, as is any type of functions taking values in the natural numbers. Another important example is the type of ideals in a ring, which will be discussed below. The name of the `ring` tactic is doubly misleading, since it assumes commutativity but works in semirings as well. In other words, it applies to any `CommSemiring`.

```
example (x y : ℕ) : (x + y) ^ 2 = x ^ 2 + y ^ 2 + 2 * x * y := by ring
```

There are also versions of the ring and semiring classes that do not assume the existence of a multiplicative unit or the associativity of multiplication. We will not discuss those here.

Some concepts that are traditionally taught in an introduction to ring theory are actually about the underlying multiplicative monoid. A prominent example is the definition of the units of a ring. Every (multiplicative) monoid `M` has a predicate `IsUnit : M → Prop` asserting existence of a two-sided inverse, a type `Units M` of units with notation $M^\times$, and a coercion to `M`. The type `Units M` bundles an invertible element with its inverse as well as properties than ensure that each is indeed the inverse of the other. This implementation detail is relevant mainly when defining computable functions. In most situations one can use `IsUnit.unit {x : M} : IsUnit x → Mˣ` to build a unit. In the commutative case, one also has `Units.mkOfMulEqOne (x y : M) : x * y = 1 → Mˣ` which builds `x` seen as unit.

```
example (x : ℤˣ) : x = 1 ∨ x = -1 := Int.units_eq_one_or x

example {M : Type*} [Monoid M] (x : Mˣ) : (x : M) * x⁻¹ = 1 := Units.mul_inv x

example {M : Type*} [Monoid M] : Group Mˣ := inferInstance
```

The type of ring morphisms between two (semi)-rings `R` and `S` is `RingHom R S`, with notation `R →+* S`.

```
example {R S : Type*} [Ring R] [Ring S] (f : R →+* S) (x y : R) :
    f (x + y) = f x + f y := f.map_add x y

example {R S : Type*} [Ring R] [Ring S] (f : R →+* S) : Rˣ →* Sˣ :=
  Units.map f
```

The isomorphism variant is `RingEquiv`, with notation `≃+*`.

As with submonoids and subgroups, there is a `Subring R` type for subrings of a ring `R`, but this type is a lot less useful than the type of subgroups since one cannot quotient a ring by a subring.

```
example {R : Type*} [Ring R] (S : Subring R) : Ring S := inferInstance
```

Also notice that `RingHom.range` produces a subring.

9.2.2 Ideals and quotients

For historical reasons, Mathlib only has a theory of ideals for commutative rings. (The ring library was originally developed to make quick progress toward the foundations of modern algebraic geometry.) So in this section we will work with commutative (semi)rings. Ideals of `R` are defined as submodules of `R` seen as `R`-modules. Modules will be covered later in a chapter on linear algebra, but this implementation detail can mostly be safely ignored since most (but not all) relevant lemmas are restated in the special context of ideals. But anonymous projection notation won't always work as expected. For instance, one cannot replace `Ideal.Quotient.mk I` by `I.Quotient.mk` in the snippet below because there are two `.`s and so it will parse as `(Ideal.Quotient I).mk`; but `Ideal.Quotient` by itself doesn't exist.

```
example {R : Type*} [CommRing R] (I : Ideal R) : R →+* R ⧸ I :=
  Ideal.Quotient.mk I

example {R : Type*} [CommRing R] {a : R} {I : Ideal R} :
    Ideal.Quotient.mk I a = 0 ↔ a ∈ I :=
  Ideal.Quotient.eq_zero_iff_mem
```

The universal property of quotient rings is `Ideal.Quotient.lift`.

```
example {R S : Type*} [CommRing R] [CommRing S] (I : Ideal R) (f : R →+* S)
    (H : I ≤ RingHom.ker f) : R ⧸ I →+* S :=
  Ideal.Quotient.lift I f H
```

In particular it leads to the first isomorphism theorem for rings.

```
example {R S : Type*} [CommRing R] [CommRing S](f : R →+* S) :
    R ⧸ RingHom.ker f ≃+* f.range :=
  RingHom.quotientKerEquivRange f
```

Ideals form a complete lattice structure with the inclusion relation, as well as a semiring structure. These two structures interact nicely.

```
variable {R : Type*} [CommRing R] {I J : Ideal R}

example : I + J = I ⊔ J := rfl

example {x : R} : x ∈ I + J ↔ ∃ a ∈ I, ∃ b ∈ J, a + b = x := by
  simp [Submodule.mem_sup]

example : I * J ≤ J := Ideal.mul_le_left

example : I * J ≤ I := Ideal.mul_le_right

example : I * J ≤ I ⊓ J := Ideal.mul_le_inf
```

One can use ring morphisms to push ideals forward and pull them back using `Ideal.map` and `Ideal.comap`, respectively. As usual, the latter is more convenient to use since it does not involve an existential quantifier. This explains why it is used to state the condition that allows us to build morphisms between quotient rings.

```
example {R S : Type*} [CommRing R] [CommRing S] (I : Ideal R) (J : Ideal S) (f : R␣
↪→+* S)
```

(continues on next page)

(continued from previous page)

```
    (H : I ≤ Ideal.comap f J) : R ⧸ I →+* S ⧸ J :=
  Ideal.quotientMap J f H
```

One subtle point is that the type `R ⧸ I` really depends on `I` (up to definitional equality), so having a proof that two ideals `I` and `J` are equal is not enough to make the corresponding quotients equal. However, the universal properties do provide an isomorphism in this case.

```
example {R : Type*} [CommRing R] {I J : Ideal R} (h : I = J) : R ⧸ I ≃+* R ⧸ J :=
  Ideal.quotEquivOfEq h
```

We can now present the Chinese remainder isomorphism as an example. Pay attention to the difference between the indexed infimum symbol ⨅ and the big product of types symbol Π. Depending on your font, those can be pretty hard to distinguish.

```
example {R : Type*} [CommRing R] {ι : Type*} [Fintype ι] (f : ι → Ideal R)
    (hf : ∀ i j, i ≠ j → IsCoprime (f i) (f j)) : (R ⧸ ⨅ i, f i) ≃+* Π i, R ⧸ f i :=
  Ideal.quotientInfRingEquivPiQuotient f hf
```

The elementary version of the Chinese remainder theorem, a statement about `ZMod`, can be easily deduced from the previous one:

```
open BigOperators PiNotation

example {ι : Type*} [Fintype ι] (a : ι → ℕ) (coprime : ∀ i j, i ≠ j → (a i).Coprime␣
↪(a j)) :
    ZMod (Π i, a i) ≃+* Π i, ZMod (a i) :=
  ZMod.prodEquivPi a coprime
```

As a series of exercises, we will reprove the Chinese remainder theorem in the general case.

We first need to define the map appearing in the theorem, as a ring morphism, using the universal property of quotient rings.

```
variable {ι R : Type*} [CommRing R]
open Ideal Quotient Function

#check Pi.ringHom
#check ker_Pi_Quotient_mk

/-- The homomorphism from ``R ⧸ ⨅ i, I i`` to ``Π i, R ⧸ I i`` featured in the␣
↪Chinese
  Remainder Theorem. -/
def chineseMap (I : ι → Ideal R) : (R ⧸ ⨅ i, I i) →+* Π i, R ⧸ I i :=
  sorry
```

Make sure the following next two lemmas can be proven by `rfl`.

```
lemma chineseMap_mk (I : ι → Ideal R) (x : R) :
    chineseMap I (Quotient.mk _ x) = fun i : ι ↦ Ideal.Quotient.mk (I i) x :=
  sorry

lemma chineseMap_mk' (I : ι → Ideal R) (x : R) (i : ι) :
    chineseMap I (mk _ x) i = mk (I i) x :=
  sorry
```

The next lemma proves the easy half of the Chinese remainder theorem, without any assumption on the family of ideals. The proof is less than one line long.

```
#check injective_lift_iff

lemma chineseMap_inj (I : ι → Ideal R) : Injective (chineseMap I) := by
  sorry
```

We are now ready for the heart of the theorem, which will show the surjectivity of our `chineseMap`. First we need to know the different ways one can express the coprimality (also called co-maximality assumption). Only the first two will be needed below.

```
#check IsCoprime
#check isCoprime_iff_add
#check isCoprime_iff_exists
#check isCoprime_iff_sup_eq
#check isCoprime_iff_codisjoint
```

We take the opportunity to use induction on `Finset`. Relevant lemmas on `Finset` are given below. Remember that the `ring` tactic works for semirings and that the ideals of a ring form a semiring.

```
#check Finset.mem_insert_of_mem
#check Finset.mem_insert_self

theorem isCoprime_Inf {I : Ideal R} {J : ι → Ideal R} {s : Finset ι}
    (hf : ∀ j ∈ s, IsCoprime I (J j)) : IsCoprime I (⨅ j ∈ s, J j) := by
  classical
  simp_rw [isCoprime_iff_add] at *
  induction s using Finset.induction with
  | empty =>
      simp
  | @insert i s _ hs =>
      rw [Finset.iInf_insert, inf_comm, one_eq_top, eq_top_iff, ← one_eq_top]
      set K := ⨅ j ∈ s, J j
      calc
        1 = I + K                 := sorry
        _ = I + K * (I + J i)     := sorry
        _ = (1 + K) * I + K * J i := sorry
        _ ≤ I + K ⊓ J i           := sorry
```

We can now prove surjectivity of the map appearing in the Chinese remainder theorem.

```
lemma chineseMap_surj [Fintype ι] {I : ι → Ideal R}
    (hI : ∀ i j, i ≠ j → IsCoprime (I i) (I j)) : Surjective (chineseMap I) := by
  classical
  intro g
  choose f hf using fun i ↦ Ideal.Quotient.mk_surjective (g i)
  have key : ∀ i, ∃ e : R, mk (I i) e = 1 ∧ ∀ j, j ≠ i → mk (I j) e = 0 := by
    intro i
    have hI' : ∀ j ∈ ({i} : Finset ι)ᶜ, IsCoprime (I i) (I j) := by
      sorry
    sorry
  choose e he using key
  use mk _ (Σ i, f i * e i)
  sorry
```

Now all the pieces come together in the following:

```
noncomputable def chineseIso [Fintype ι] (f : ι → Ideal R)
    (hf : ∀ i j, i ≠ j → IsCoprime (f i) (f j)) : (R ⧸ ⨅ i, f i) ≃+* Π i, R ⧸ f i :=
```

(continues on next page)

(continued from previous page)

```
  { Equiv.ofBijective _ ⟨chineseMap_inj f, chineseMap_surj hf⟩,
    chineseMap f with }
```

9.2.3 Algebras and polynomials

Given a commutative (semi)ring `R`, an *algebra over* `R` is a semiring `A` equipped with a ring morphism whose image commutes with every element of `A`. This is encoded as a type class `Algebra R A`. The morphism from `R` to `A` is called the structure map and is denoted `algebraMap R A : R →+* A` in Lean. Multiplication of `a : A` by `algebraMap R A r` for some `r : R` is called the scalar multiplication of `a` by `r` and denoted by `r • a`. Note that this notion of algebra is sometimes called an *associative unital algebra* to emphasize the existence of more general notions of algebra.

The fact that `algebraMap R A` is ring morphism packages together a lot of properties of scalar multiplication, such as the following:

```
example {R A : Type*} [CommRing R] [Ring A] [Algebra R A] (r r' : R) (a : A) :
    (r + r') • a = r • a + r' • a :=
  add_smul r r' a

example {R A : Type*} [CommRing R] [Ring A] [Algebra R A] (r r' : R) (a : A) :
    (r * r') • a = r • r' • a :=
  mul_smul r r' a
```

The morphisms between two `R`-algebras `A` and `B` are ring morphisms which commute with scalar multiplication by elements of `R`. They are bundled morphisms with type `AlgHom R A B`, which is denoted by `A →ₐ[R] B`.

Important examples of non-commutative algebras include algebras of endomorphisms and algebras of square matrices, both of which will be covered in the chapter on linear algebra. In this chapter we will discuss one of the most important examples of a commutative algebra, namely, polynomial algebras.

The algebra of univariate polynomials with coefficients in `R` is called `Polynomial R`, which can be written as `R[X]` as soon as one opens the `Polynomial` namespace. The algebra structure map from `R` to `R[X]` is denoted by `C`, which stands for "constant" since the corresponding polynomial functions are always constant. The indeterminate is denoted by `X`.

```
open Polynomial

example {R : Type*} [CommRing R] : R[X] := X

example {R : Type*} [CommRing R] (r : R) := X - C r
```

In the first example above, it is crucial that we give Lean the expected type since it cannot be determined from the body of the definition. In the second example, the target polynomial algebra can be inferred from our use of `C r` since the type of `r` is known.

Because `C` is a ring morphism from `R` to `R[X]`, we can use all ring morphisms lemmas such as `map_zero`, `map_one`, `map_mul`, and `map_pow` before computing in the ring `R[X]`. For example:

```
example {R : Type*} [CommRing R] (r : R) : (X + C r) * (X - C r) = X ^ 2 - C (r ^ 2)␣
↪:= by
  rw [C.map_pow]
  ring
```

You can access coefficients using `Polynomial.coeff`

```
example {R : Type*} [CommRing R] (r:R) : (C r).coeff 0 = r := by simp

example {R : Type*} [CommRing R] : (X ^ 2 + 2 * X + C 3 : R[X]).coeff 1 = 2 := by simp
```

Defining the degree of a polynomial is always tricky because of the special case of the zero polynomial. Mathlib has two variants: `Polynomial.natDegree : R[X] → ℕ` assigns degree `0` to the zero polynomial, and `Polynomial.degree : R[X] → WithBot ℕ` assigns $\bot$. In the latter, `WithBot ℕ` can be seen as $\mathbb{N} \cup \{-\infty\}$, except that $-\infty$ is denoted $\bot$, the same symbol as the bottom element in a complete lattice. This special value is used as the degree of the zero polynomial, and it is absorbent for addition. (It is almost absorbent for multiplication, except that `⊥ * 0 = 0`.)

Morally speaking, the `degree` version is the correct one. For instance, it allows us to state the expected formula for the degree of a product (assuming the base ring has no zero divisor).

```
example {R : Type*} [Semiring R] [NoZeroDivisors R] {p q : R[X]} :
    degree (p * q) = degree p + degree q :=
  Polynomial.degree_mul
```

Whereas the version for `natDegree` needs to assume non-zero polynomials.

```
example {R : Type*} [Semiring R] [NoZeroDivisors R] {p q : R[X]} (hp : p ≠ 0) (hq : q␣
↪≠ 0) :
    natDegree (p * q) = natDegree p + natDegree q :=
  Polynomial.natDegree_mul hp hq
```

However, $\mathbb{N}$ is much nicer to use than `WithBot ℕ`, so Mathlib makes both versions available and provides lemmas to convert between them. Also, `natDegree` is the more convenient definition to use when computing the degree of a composition. Composition of polynomial is `Polynomial.comp` and we have:

```
example {R : Type*} [Semiring R] [NoZeroDivisors R] {p q : R[X]} :
    natDegree (comp p q) = natDegree p * natDegree q :=
  Polynomial.natDegree_comp
```

Polynomials give rise to polynomial functions: any polynomial can be evaluated on `R` using `Polynomial.eval`.

```
example {R : Type*} [CommRing R] (P: R[X]) (x : R) := P.eval x

example {R : Type*} [CommRing R] (r : R) : (X - C r).eval r = 0 := by simp
```

In particular, there is a predicate, `IsRoot`, that holds for elements `r` in `R` where a polynomial vanishes.

```
example {R : Type*} [CommRing R] (P : R[X]) (r : R) : IsRoot P r ↔ P.eval r = 0 :=␣
↪Iff.rfl
```

We would like to say that, assuming `R` has no zero divisor, a polynomial has at most as many roots as its degree, where the roots are counted with multiplicities. But once again the case of the zero polynomial is painful. So Mathlib defines `Polynomial.roots` to send a polynomial `P` to a multiset, i.e. the finite set that is defined to be empty if `P` is zero and the roots of `P`, with multiplicities, otherwise. This is defined only when the underlying ring is a domain since otherwise the definition does not have good properties.

```
example {R : Type*} [CommRing R] [IsDomain R] (r : R) : (X - C r).roots = {r} :=
  roots_X_sub_C r

example {R : Type*} [CommRing R] [IsDomain R] (r : R) (n : ℕ):
    ((X - C r) ^ n).roots = n • {r} :=
  by simp
```

Both `Polynomial.eval` and `Polynomial.roots` consider only the coefficients ring. They do not allow us to say that `X ^ 2 - 2 : ℚ[X]` has a root in ℝ or that `X ^ 2 + 1 : ℝ[X]` has a root in ℂ. For this, we need `Polynomial.aeval`, which will evaluate `P : R[X]` in any `R`-algebra. More precisely, given a semiring `A` and an instance of `Algebra R A`, `Polynomial.aeval` sends every element of `a` along the `R`-algebra morphism of evaluation at `a`. Since `AlgHom` has a coercion to functions, one can apply it to a polynomial. But `aeval` does not have a polynomial as an argument, so one cannot use dot notation like in `P.eval` above.

```
example : aeval Complex.I (X ^ 2 + 1 : ℝ[X]) = 0 := by simp
```

The function corresponding to `roots` in this context is `aroots` which takes a polynomial and then an algebra and outputs a multiset (with the same caveat about the zero polynomial as for `roots`).

```
open Complex Polynomial

example : aroots (X ^ 2 + 1 : ℝ[X]) ℂ = {Complex.I, -I} := by
  suffices roots (X ^ 2 + 1 : ℂ[X]) = {I, -I} by simpa [aroots_def]
  have factored : (X ^ 2 + 1 : ℂ[X]) = (X - C I) * (X - C (-I)) := by
    have key : (C I * C I : ℂ[X]) = -1 := by simp [← C_mul]
    rw [C_neg]
    linear_combination key
  have p_ne_zero : (X - C I) * (X - C (-I)) ≠ 0 := by
    intro H
    apply_fun eval 0 at H
    simp [eval] at H
  simp only [factored, roots_mul p_ne_zero, roots_X_sub_C]
  rfl

-- Mathlib knows about D'Alembert-Gauss theorem: ``ℂ`` is algebraically closed.
example : IsAlgClosed ℂ := inferInstance
```

More generally, given an ring morphism `f : R →+* S` one can evaluate `P : R[X]` at a point in `S` using $\texttt{Polynomial.eval}_2$. This one produces an actual function from `R[X]` to `S` since it does not assume the existence of a `Algebra R S` instance, so dot notation works as you would expect.

```
#check (Complex.ofRealHom : ℝ →+* ℂ)

example : (X ^ 2 + 1 : ℝ[X]).eval₂ Complex.ofRealHom Complex.I = 0 := by simp
```

Let us end by mentioning multivariate polynomials briefly. Given a commutative semiring `R`, the `R`-algebra of polynomials with coefficients in `R` and indeterminates indexed by a type σ is `MVPolynomial σ R`. Given `i : σ`, the corresponding polynomial is `MvPolynomial.X i`. (As usual, one can open the `MVPolynomial` namespace to shorten this to `X i`.) For instance, if we want two indeterminates we can use `Fin 2` as σ and write the polynomial defining the unit circle in $\mathbb{R}^{2}$` as:

```
open MvPolynomial

def circleEquation : MvPolynomial (Fin 2) ℝ := X 0 ^ 2 + X 1 ^ 2 - 1
```

Recall that function application has a very high precedence so the expression above is read as `(X 0) ^ 2 + (X 1) ^ 2 - 1`. We can evaluate it to make sure the point with coordinates $(1, 0)$ is on the circle. Recall the `![...]` notation denotes elements of `Fin n → X` for some natural number `n` determined by the number of arguments and some type `X` determined by the type of arguments.

```
example : MvPolynomial.eval ![1, 0] circleEquation = 0 := by simp [circleEquation]
```

CHAPTER

TEN

LINEAR ALGEBRA

10.1 Vector spaces and linear maps

10.1.1 Vector spaces

We will start directly abstract linear algebra, taking place in a vector space over any field. However you can find information about matrices in Section 10.4.1 which does not logically depend on this abstract theory. Mathlib actually deals with a more general version of linear algebra involving the word module, but for now we will pretend this is only an eccentric spelling habit.

The way to say "let K be a field and let V be a vector space over K" (and make them implicit arguments to later results) is:

```
variable {K : Type*} [Field K] {V : Type*} [AddCommGroup V] [Module K V]
```

We explained in Chapter 8 why we need two separate typeclasses `[AddCommGroup V]` `[Module K V]`. The short version is the following. Mathematically we want to say that having a K-vector space structure implies having an additive commutative group structure. We could tell this to Lean. But then whenever Lean would need to find such a group structure on a type V, it would go hunting for vector space structures using a *completely unspecified* field K that cannot be inferred from V. This would be very bad for the type class synthesis system.

The multiplication of a vector v by a scalar a is denoted by $a \cdot v$. We list a couple of algebraic rules about the interaction of this operation with addition in the following examples. Of course *simp* or *apply?* would find those proofs. There is also a *module* tactic that solves goals following from the axioms of vector spaces and fields, in the same way the *ring* tactic is used in commutative rings or the *group* tactic is used in groups. But it is still useful to remember that scalar multiplication is abbreviated *smul* in lemma names.

```
example (a : K) (u v : V) : a • (u + v) = a • u + a • v :=
  smul_add a u v

example (a b : K) (u : V) : (a + b) • u = a • u + b • u :=
  add_smul a b u

example (a b : K) (u : V) : a • b • u = b • a • u :=
  smul_comm a b u
```

As a quick note for more advanced readers, let us point out that, as suggested by terminology, Mathlib's linear algebra also covers modules over (not necessarily commutative) rings. In fact it even covers semi-modules over semi-rings. If you think you do not need this level of generality, you can meditate the following example that nicely captures a lot of algebraic rules about ideals acting on submodules:

```
example {R M : Type*} [CommSemiring R] [AddCommMonoid M] [Module R M] :
    Module (Ideal R) (Submodule R M) :=
  inferInstance
```

10.1.2 Linear maps

Next we need linear maps. Like group morphisms, linear maps in Mathlib are bundled maps, i.e. packages made of a map and proofs of its linearity properties. Those bundled maps are converted to ordinary functions when applied. See Chapter 8 for more information about this design.

The type of linear maps between two `K`-vector spaces `V` and `W` is denoted by `V →ₗ[K] W`. The subscript *l* stands for linear. At first it may feel odd to specify `K` in this notation. But this is crucial when several fields come into play. For instance real-linear maps from $\mathbb{C}$ to $\mathbb{C}$ are every map $z \mapsto az + b\bar{z}$ while only the maps $z \mapsto az$ are complex linear, and this difference is crucial in complex analysis.

```
variable {W : Type*} [AddCommGroup W] [Module K W]

variable (φ : V →ₗ[K] W)

example (a : K) (v : V) : φ (a • v) = a • φ v :=
  map_smul φ a v

example (v w : V) : φ (v + w) = φ v + φ w :=
  map_add φ v w
```

Note that `V →ₗ[K] W` itself carries interesting algebraic structures (this is part of the motivation for bundling those maps). It is a `K`-vector space so we can add linear maps and multiply them by scalars.

```
variable (ψ : V →ₗ[K] W)

#check (2 • φ + ψ : V →ₗ[K] W)
```

One downside of using bundled maps is that we cannot use ordinary function composition. We need to use `LinearMap.comp` or the notation `∘ₗ`.

```
variable (θ : W →ₗ[K] V)

#check (φ.comp θ : W →ₗ[K] W)
#check (φ ∘ₗ θ : W →ₗ[K] W)
```

There are two main ways to construct linear maps. First we can build the structure by providing the function and the linearity proof. As usual, this is facilitated by the structure code action: you can type `example : V →ₗ[K] V := _` and use the code action "Generate a skeleton" attached to the underscore.

```
example : V →ₗ[K] V where
  toFun v := 3 • v
  map_add' _ _ := smul_add ..
  map_smul' _ _ := smul_comm ..
```

You may wonder why the proof fields of `LinearMap` have names ending with a prime. This is because they are defined before the coercion to functions is defined, hence they are phrased in terms of `LinearMap.toFun`. Then they are restated as `LinearMap.map_add` and `LinearMap.map_smul` in terms of the coercion to function. This is not yet the end of the story. One also wants a version of `map_add` that applies to any (bundled) map preserving addition, such as additive group morphisms, linear maps, continuous linear maps, `K`-algebra maps etc… This one is `map_add` (in the root namespace). The intermediate version, `LinearMap.map_add` is a bit redundant but allows to use dot

notation, which can be nice sometimes. A similar story exists for `map_smul`, and the general framework is explained in Chapter 8.

```
#check (φ.map_add' : ∀ x y : V, φ.toFun (x + y) = φ.toFun x + φ.toFun y)
#check (φ.map_add : ∀ x y : V, φ (x + y) = φ x + φ y)
#check (map_add φ : ∀ x y : V, φ (x + y) = φ x + φ y)
```

One can also build linear maps from the ones that are already defined in Mathlib using various combinators. For instance the above example is already known as `LinearMap.lsmul K V 3`. There are several reason why `K` and `V` are explicit arguments here. The most pressing one is that from a bare `LinearMap.lsmul 3` there would be no way for Lean to infer `V` or even `K`. But also `LinearMap.lsmul K V` is an interesting object by itself: it has type `K →ₗ[K] V →ₗ[K] V`, meaning it is a `K`-linear map from `K` —seen as a vector space over itself— to the space of `K`-linear maps from `V` to `V`.

```
#check (LinearMap.lsmul K V 3 : V →ₗ[K] V)
#check (LinearMap.lsmul K V : K →ₗ[K] V →ₗ[K] V)
```

There is also a type `LinearEquiv` of linear isomorphisms denoted by `V ≃ₗ[K] W`. The inverse of `f : V ≃ₗ[K] W` is `f.symm : W ≃ₗ[K] V`, composition of `f` and `g` is `f.trans g` also denoted by `f ≪≫ₗ g`, and the identity isomorphism of `V` is `LinearEquiv.refl K V`. Elements of this type are automatically coerced to morphisms and functions when necessary.

```
example (f : V ≃ₗ[K] W) : f ≪≫ₗ f.symm = LinearEquiv.refl K V :=
  f.self_trans_symm
```

One can use `LinearEquiv.ofBijective` to build an isomorphism from a bijective morphism. Doing so makes the inverse function noncomputable.

```
noncomputable example (f : V →ₗ[K] W) (h : Function.Bijective f) : V ≃ₗ[K] W :=
  .ofBijective f h
```

Note that in the above example, Lean uses the announced type to understand that `.ofBijective` refers to `LinearEquiv.ofBijective` (without needing to open any namespace).

10.1.3 Sums and products of vector spaces

We can build new vector spaces out of old ones using direct sums and direct products. Let us start with two vectors spaces. In this case there is no difference between sum and product, and we can simply use the product type. In the following snippet of code we simply show how to get all the structure maps (inclusions and projections) as linear maps, as well as the universal properties constructing linear maps into products and out of sums (if you are not familiar with the category-theoretic distinction between sums and products, you can simply ignore the universal property vocabulary and focus on the types of the following examples).

```
section binary_product

variable {W : Type*} [AddCommGroup W] [Module K W]
variable {U : Type*} [AddCommGroup U] [Module K U]
variable {T : Type*} [AddCommGroup T] [Module K T]

-- First projection map
example : V × W →ₗ[K] V := LinearMap.fst K V W

-- Second projection map
example : V × W →ₗ[K] W := LinearMap.snd K V W
```

(continues on next page)

(continued from previous page)

```
-- Universal property of the product
example (φ : U →ₗ[K] V) (ψ : U →ₗ[K] W) : U →ₗ[K]  V × W := LinearMap.prod φ ψ

-- The product map does the expected thing, first component
example (φ : U →ₗ[K] V) (ψ : U →ₗ[K] W) : LinearMap.fst K V W ∘ₗ LinearMap.prod φ ψ␣
↪= φ := rfl

-- The product map does the expected thing, second component
example (φ : U →ₗ[K] V) (ψ : U →ₗ[K] W) : LinearMap.snd K V W ∘ₗ LinearMap.prod φ ψ␣
↪= ψ := rfl

-- We can also combine maps in parallel
example (φ : V →ₗ[K] U) (ψ : W →ₗ[K] T) : (V × W) →ₗ[K] (U × T) := φ.prodMap ψ

-- This is simply done by combining the projections with the universal property
example (φ : V →ₗ[K] U) (ψ : W →ₗ[K] T) :
  φ.prodMap ψ = (φ ∘ₗ .fst K V W).prod (ψ ∘ₗ .snd K V W) := rfl

-- First inclusion map
example : V →ₗ[K] V × W := LinearMap.inl K V W

-- Second inclusion map
example : W →ₗ[K] V × W := LinearMap.inr K V W

-- Universal property of the sum (aka coproduct)
example (φ : V →ₗ[K] U) (ψ : W →ₗ[K] U) : V × W →ₗ[K] U := φ.coprod ψ

-- The coproduct map does the expected thing, first component
example (φ : V →ₗ[K] U) (ψ : W →ₗ[K] U) : φ.coprod ψ ∘ₗ LinearMap.inl K V W = φ :=
  LinearMap.coprod_inl φ ψ

-- The coproduct map does the expected thing, second component
example (φ : V →ₗ[K] U) (ψ : W →ₗ[K] U) : φ.coprod ψ ∘ₗ LinearMap.inr K V W = ψ :=
  LinearMap.coprod_inr φ ψ

-- The coproduct map is defined in the expected way
example (φ : V →ₗ[K] U) (ψ : W →ₗ[K] U) (v : V) (w : W) :
    φ.coprod ψ (v, w) = φ v + ψ w :=
  rfl

end binary_product
```

Let us now turn to sums and products of arbitrary families of vector spaces. Here we will simply see how to define a family of vector spaces and access the universal properties of sums and products. Note that the direct sum notation is scoped to the `DirectSum` namespace, and that the universal property of direct sums requires decidable equality on the indexing type (this is somehow an implementation accident).

```
section families
open DirectSum

variable {ι : Type*} [DecidableEq ι]
         (V : ι → Type*) [∀ i, AddCommGroup (V i)] [∀ i, Module K (V i)]

-- The universal property of the direct sum assembles maps from the summands to build
-- a map from the direct sum
example (φ : Π i, (V i →ₗ[K] W)) : (⊕ i, V i) →ₗ[K] W :=
```

(continues on next page)

(continued from previous page)

```
  DirectSum.toModule K ι W φ

-- The universal property of the direct product assembles maps into the factors
-- to build a map into the direct product
example (φ : Π i, (W →ₗ[K] V i)) : W →ₗ[K] (Π i, V i) :=
  LinearMap.pi φ

-- The projection maps from the product
example (i : ι) : (Π j, V j) →ₗ[K] V i := LinearMap.proj i

-- The inclusion maps into the sum
example (i : ι) : V i →ₗ[K] (⊕ i, V i) := DirectSum.lof K ι V i

-- The inclusion maps into the product
example (i : ι) : V i →ₗ[K] (Π i, V i) := LinearMap.single K V i

-- In case `ι` is a finite type, there is an isomorphism between the sum and product.
example [Fintype ι] : (⊕ i, V i) ≃ₗ[K] (Π i, V i) :=
  linearEquivFunOnFintype K ι V

end families
```

10.2 Subspaces and quotients

10.2.1 Subspaces

Just as linear maps are bundled, a linear subspace of `V` is also a bundled structure consisting of a set in `V`, called the carrier of the subspace, with the relevant closure properties. Again the word module appears instead of vector space because of the more general context that Mathlib actually uses for linear algebra.

```
variable {K : Type*} [Field K] {V : Type*} [AddCommGroup V] [Module K V]

example (U : Submodule K V) {x y : V} (hx : x ∈ U) (hy : y ∈ U) :
    x + y ∈ U :=
  U.add_mem hx hy

example (U : Submodule K V) {x : V} (hx : x ∈ U) (a : K) :
    a • x ∈ U :=
  U.smul_mem a hx
```

In the example above, it is important to understand that `Submodule K V` is the type of `K`-linear subspaces of `V`, rather than a predicate `IsSubmodule U` where `U` is an element of `Set V`. `Submodule K V` is endowed with a coercion to `Set V` and a membership predicate on `V`. See Section 8.3 for an explanation of how and why this is done.

Of course, two subspaces are the same if and only if they have the same elements. This fact is registered for use with the `ext` tactic, which can be used to prove two subspaces are equal in the same way it is used to prove that two sets are equal.

To state and prove, for example, that $\mathbb{R}$ is a $\mathbb{R}$-linear subspace of $\mathbb{C}$, what we really want is to construct a term of type `Submodule ℝ ℂ` whose projection to `Set ℂ` is $\mathbb{R}$, or, more precisely, the image of $\mathbb{R}$ in $\mathbb{C}$.

```
noncomputable example : Submodule ℝ ℂ where
  carrier := Set.range ((↑) : ℝ → ℂ)
```

(continues on next page)

(continued from previous page)

```
  add_mem' := by
    rintro _ _ ⟨n, rfl⟩ ⟨m, rfl⟩
    use n + m
    simp
  zero_mem' := by
    use 0
    simp
  smul_mem' := by
    rintro c - ⟨a, rfl⟩
    use c*a
    simp
```

The prime at the end of proof fields in `Submodule` are analogous to the one in `LinearMap`. Those fields are stated in terms of the `carrier` field because they are defined before the `MemberShip` instance. They are then superseded by `Submodule.add_mem`, `Submodule.zero_mem` and `Submodule.smul_mem` that we saw above.

As an exercise in manipulating subspaces and linear maps, you will define the pre-image of a subspace by a linear map (of course we will see below that Mathlib already knows about this). Remember that `Set.mem_preimage` can be used to rewrite a statement involving membership and preimage. This is the only lemma you will need in addition to the lemmas discussed above about `LinearMap` and `Submodule`.

```
def preimage {W : Type*} [AddCommGroup W] [Module K W] (φ : V →ₗ[K] W) (H : Submodule␣
↪K W) :
    Submodule K V where
  carrier := φ ⁻¹' H
  zero_mem' := by
    sorry
  add_mem' := by
    sorry
  smul_mem' := by
    sorry
```

Using type classes, Mathlib knows that a subspace of a vector space inherits a vector space structure.

```
example (U : Submodule K V) : Module K U := inferInstance
```

This example is subtle. The object `U` is not a type, but Lean automatically coerces it to a type by interpreting it as a subtype of `V`. So the above example can be restated more explicitly as:

```
example (U : Submodule K V) : Module K {x : V // x ∈ U} := inferInstance
```

10.2.2 Complete lattice structure and internal direct sums

An important benefit of having a type `Submodule K V` instead of a predicate `IsSubmodule : Set V → Prop` is that one can easily endow `Submodule K V` with additional structure. Importantly, it has the structure of a complete lattice structure with respect to inclusion. For instance, instead of having a lemma stating that an intersection of two subspaces of `V` is again a subspace, we use the lattice operation `⊓` to construct the intersection. We can then apply arbitrary lemmas about lattices to the construction.

Let us check that the set underlying the infimum of two subspaces is indeed, by definition, their intersection.

```
example (H H' : Submodule K V) :
    ((H ⊓ H' : Submodule K V) : Set V) = (H : Set V) ∩ (H' : Set V) := rfl
```

It may look strange to have a different notation for what amounts to the intersection of the underlying sets, but the correspondence does not carry over to the supremum operation and set union, since a union of subspaces is not, in general, a subspace. Instead one needs to use the subspace generated by the union, which is done using `Submodule.span`.

```
example (H H' : Submodule K V) :
    ((H ⊔ H' : Submodule K V) : Set V) = Submodule.span K ((H : Set V) ∪ (H' : Set␣
↪V)) := by
  simp [Submodule.span_union]
```

Another subtlety is that `V` itself does not have type `Submodule K V`, so we need a way to talk about `V` seen as a subspace of `V`. This is also provided by the lattice structure: the full subspace is the top element of this lattice.

```
example (x : V) : x ∈ (⊤ : Submodule K V) := trivial
```

Similarly the bottom element of this lattice is the subspace whose only element is the zero element.

```
example (x : V) : x ∈ (⊥ : Submodule K V) ↔ x = 0 := Submodule.mem_bot K
```

In particular we can discuss the case of subspaces that are in (internal) direct sum. In the case of two subspaces, we use the general purpose predicate `IsCompl` which makes sense for any bounded partially ordered type. In the case of general families of subspaces we use `DirectSum.IsInternal`.

```
-- If two subspaces are in direct sum then they span the whole space.
example (U V : Submodule K V) (h : IsCompl U V) :
  U ⊔ V = ⊤ := h.sup_eq_top

-- If two subspaces are in direct sum then they intersect only at zero.
example (U V : Submodule K V) (h : IsCompl U V) :
  U ⊓ V = ⊥ := h.inf_eq_bot

section
open DirectSum
variable {ι : Type*} [DecidableEq ι]

-- If subspaces are in direct sum then they span the whole space.
example (U : ι → Submodule K V) (h : DirectSum.IsInternal U) :
  ⨆ i, U i = ⊤ := h.submodule_iSup_eq_top

-- If subspaces are in direct sum then they pairwise intersect only at zero.
example {ι : Type*} [DecidableEq ι] (U : ι → Submodule K V) (h : DirectSum.IsInternal␣
↪U)
    {i j : ι} (hij : i ≠ j) : U i ⊓ U j = ⊥ :=
  (h.submodule_iSupIndep.pairwiseDisjoint hij).eq_bot

-- Those conditions characterize direct sums.
#check DirectSum.isInternal_submodule_iff_independent_and_iSup_eq_top

-- The relation with external direct sums: if a family of subspaces is
-- in internal direct sum then the map from their external direct sum into `V`
-- is a linear isomorphism.
noncomputable example {ι : Type*} [DecidableEq ι] (U : ι → Submodule K V)
    (h : DirectSum.IsInternal U) : (⨁ i, U i) ≃ₗ[K] V :=
  LinearEquiv.ofBijective (coeLinearMap U) h
end
```

10.2.3 Subspace spanned by a set

In addition to building subspaces out of existing subspaces, we can build them out of any set `s` using `Submodule.span K s` which builds the smallest subspace containing `s`. On paper it is common to use that this space is made of all linear combinations of elements of `s`. But it is often more efficient to use its universal property expressed by `Submodule.span_le`, and the whole theory of Galois connections.

```
example {s : Set V} (E : Submodule K V) : Submodule.span K s ≤ E ↔ s ⊆ E :=
  Submodule.span_le

example : GaloisInsertion (Submodule.span K) ((↑) : Submodule K V → Set V) :=
  Submodule.gi K V
```

When those are not enough, one can use the relevant induction principle `Submodule.span_induction` which ensures a property holds for every element of the span of `s` as long as it holds on `zero` and elements of `s` and is stable under sum and scalar multiplication.

As an exercise, let us reprove one implication of `Submodule.mem_sup`. Remember that you can use the *module* tactic to close goals that follow from the axioms relating the various algebraic operations on `V`.

```
example {S T : Submodule K V} {x : V} (h : x ∈ S ⊔ T) :
    ∃ s ∈ S, ∃ t ∈ T, x = s + t  := by
  rw [← S.span_eq, ← T.span_eq, ← Submodule.span_union] at h
  induction h using Submodule.span_induction with
  | mem y h =>
      sorry
  | zero =>
      sorry
  | add x y hx hy hx' hy' =>
      sorry
  | smul a x hx hx' =>
      sorry
```

10.2.4 Pushing and pulling subspaces

As promised earlier, we now describe how to push and pull subspaces by linear maps. As usual in Mathlib, the first operation is called `map` and the second one is called `comap`.

```
section

variable {W : Type*} [AddCommGroup W] [Module K W] (φ : V →ₗ[K] W)

variable (E : Submodule K V) in
#check (Submodule.map φ E : Submodule K W)

variable (F : Submodule K W) in
#check (Submodule.comap φ F : Submodule K V)
```

Note those live in the `Submodule` namespace so one can use dot notation and write `E.map φ` instead of `Submodule.map φ E`, but this is pretty awkward to read (although some Mathlib contributors use this spelling).

In particular the range and kernel of a linear map are subspaces. Those special cases are important enough to get declarations.

```
example : LinearMap.range φ = .map φ ⊤ := LinearMap.range_eq_map φ

example : LinearMap.ker φ = .comap φ ⊥ := Submodule.comap_bot φ -- or `rfl`
```

Note that we cannot write `φ.ker` instead of `LinearMap.ker φ` because `LinearMap.ker` also applies to classes of maps preserving more structure, hence it does not expect an argument whose type starts with `LinearMap`, hence dot notation doesn't work here. However we were able to use the other flavor of dot notation in the right-hand side. Because Lean expects a term with type `Submodule K V` after elaborating the left-hand side, it interprets `.comap` as `Submodule.comap`.

The following lemmas give the key relations between those submodule and the properties of φ.

```
open Function LinearMap

example : Injective φ ↔ ker φ = ⊥ := ker_eq_bot.symm

example : Surjective φ ↔ range φ = ⊤ := range_eq_top.symm
```

As an exercise, let us prove the Galois connection property for `map` and `comap`. One can use the following lemmas but this is not required since they are true by definition.

```
#check Submodule.mem_map_of_mem
#check Submodule.mem_map
#check Submodule.mem_comap

example (E : Submodule K V) (F : Submodule K W) :
    Submodule.map φ E ≤ F ↔ E ≤ Submodule.comap φ F := by
  sorry
```

10.2.5 Quotient spaces

Quotient vector spaces use the general quotient notation (typed with `\quot`, not the ordinary `/`). The projection onto a quotient space is `Submodule.mkQ` and the universal property is `Submodule.liftQ`.

```
variable (E : Submodule K V)

example : Module K (V ⧸ E) := inferInstance

example : V →ₗ[K] V ⧸ E := E.mkQ

example : ker E.mkQ = E := E.ker_mkQ

example : range E.mkQ = ⊤ := E.range_mkQ

example (hφ : E ≤ ker φ) : V ⧸ E →ₗ[K] W := E.liftQ φ hφ

example (F : Submodule K W) (hφ : E ≤ .comap φ F) : V ⧸ E →ₗ[K] W ⧸ F := E.mapQ F φ␣
↪hφ

noncomputable example : (V ⧸ LinearMap.ker φ) ≃ₗ[K] range φ := φ.quotKerEquivRange
```

As an exercise, let us prove the correspondence theorem for subspaces of quotient spaces. Mathlib knows a slightly more precise version as `Submodule.comapMkQRelIso`.

```
open Submodule

#check Submodule.map_comap_eq
#check Submodule.comap_map_eq

example : Submodule K (V ⧸ E) ≃ { F : Submodule K V // E ≤ F } where
  toFun := sorry
  invFun := sorry
  left_inv := sorry
  right_inv := sorry
```

10.3 Endomorphisms

An important special case of linear maps are endomorphisms: linear maps from a vector space to itself. They are interesting because they form a `K`-algebra. In particular we can evaluate polynomials with coefficients in `K` on them, and they can have eigenvalues and eigenvectors.

Mathlib uses the abbreviation `Module.End K V := V →ₗ[K] V` which is convenient when using a lot of these (especially after opening the `Module` namespace).

```
variable {K : Type*} [Field K] {V : Type*} [AddCommGroup V] [Module K V]

variable {W : Type*} [AddCommGroup W] [Module K W]


open Polynomial Module LinearMap End

example (φ ψ : End K V) : φ * ψ = φ ∘ₗ ψ :=
  End.mul_eq_comp φ ψ -- `rfl` would also work

-- evaluating `P` on `φ`
example (P : K[X]) (φ : End K V) : V →ₗ[K] V :=
  aeval φ P

-- evaluating `X` on `φ` gives back `φ`
example (φ : End K V) : aeval φ (X : K[X]) = φ :=
  aeval_X φ
```

As an exercise manipulating endomorphisms, subspaces and polynomials, let us prove the (binary) kernels lemma: for any endomorphism φ and any two relatively prime polynomials P and Q, we have $\ker P(\varphi) \oplus \ker Q(\varphi) = \ker \big(PQ(\varphi)\big)$.

Note that `IsCoprime x y` is defined as `∃ a b, a * x + b * y = 1`.

```
#check Submodule.eq_bot_iff
#check Submodule.mem_inf
#check LinearMap.mem_ker

example (P Q : K[X]) (h : IsCoprime P Q) (φ : End K V) : ker (aeval φ P) ⊓ ker (aeval␣
↪φ Q) = ⊥ := by
  sorry

#check Submodule.add_mem_sup
#check map_mul
#check End.mul_apply
#check LinearMap.ker_le_ker_comp
```

(continues on next page)

(continued from previous page)

```
example (P Q : K[X]) (h : IsCoprime P Q) (φ : End K V) :
    ker (aeval φ P) ⊓ ker (aeval φ Q) = ker (aeval φ (P*Q)) := by
  sorry
```

We now move to the discussions of eigenspaces and eigenvalues. The eigenspace associated to an endomorphism φ and a scalar a is the kernel of $\varphi - aId$. Eigenspaces are defined for all values of `a`, although they are interesting only when they are non-zero. However an eigenvector is, by definition, a non-zero element of an eigenspace. The corresponding predicate is `End.HasEigenvector`.

```
example (φ : End K V) (a : K) : φ.eigenspace a = LinearMap.ker (φ - a • 1) :=
  End.eigenspace_def
```

Then there is a predicate `End.HasEigenvalue` and the corresponding subtype `End.Eigenvalues`.

```
example (φ : End K V) (a : K) : φ.HasEigenvalue a ↔ φ.eigenspace a ≠ ⊥ :=
  Iff.rfl

example (φ : End K V) (a : K) : φ.HasEigenvalue a ↔ ∃ v, φ.HasEigenvector a v  :=
  ⟨End.HasEigenvalue.exists_hasEigenvector, fun ⟨_, hv⟩ ↦ φ.hasEigenvalue_of_
↪hasEigenvector hv⟩

example (φ : End K V) : φ.Eigenvalues = {a // φ.HasEigenvalue a} :=
  rfl

-- Eigenvalue are roots of the minimal polynomial
example (φ : End K V) (a : K) : φ.HasEigenvalue a → (minpoly K φ).IsRoot a :=
  φ.isRoot_of_hasEigenvalue

-- In finite dimension, the converse is also true (we will discuss dimension below)
example [FiniteDimensional K V] (φ : End K V) (a : K) :
    φ.HasEigenvalue a ↔ (minpoly K φ).IsRoot a :=
  φ.hasEigenvalue_iff_isRoot

-- Cayley-Hamilton
example [FiniteDimensional K V] (φ : End K V) : aeval φ φ.charpoly = 0 :=
  φ.aeval_self_charpoly
```

10.4 Matrices, bases and dimension

10.4.1 Matrices

Before introducing bases for abstract vector spaces, we go back to the much more elementary setup of linear algebra in K^n for some field K. Here the main objects are vectors and matrices. For concrete vectors, one can use the `![...]` notation, where components are separated by commas. For concrete matrices we can use the `!![...]` notation, lines are separated by semi-colons and components of lines are separated by colons. When entries have a computable type such as $\mathbb{N}$ or $\mathbb{Q}$, we can use the `eval` command to play with basic operations.

```
section matrices

-- Adding vectors
#eval ![1, 2] + ![3, 4]  -- ![4, 6]
```

(continues on next page)

(continued from previous page)

```
-- Adding matrices
#eval !![1, 2; 3, 4] + !![3, 4; 5, 6]  -- !![4, 6; 8, 10]

-- Multiplying matrices
#eval !![1, 2; 3, 4] * !![3, 4; 5, 6]  -- !![13, 16; 29, 36]
```

It is important to understand that this use of `#eval` is interesting only for exploration, it is not meant to replace a computer algebra system such as Sage. The data representation used here for matrices is *not* computationally efficient in any way. It uses functions instead of arrays and is optimized for proving, not computing. The virtual machine used by `#eval` is also not optimized for this use.

Beware the matrix notation list rows but the vector notation is neither a row vector nor a column vector. Multiplication of a matrix with a vector from the left (resp. right) interprets the vector as a row (resp. column) vector. This corresponds to operations `Matrix.vecMul`, with notation $_v*$ and `Matrix.mulVec`, with notation `` *$_v$` ``. Those notations are scoped in the `Matrix` namespace that we therefore need to open.

```
open Matrix

-- matrices acting on vectors on the left
#eval !![1, 2; 3, 4] *ᵥ ![1, 1] -- ![3, 7]

-- matrices acting on vectors on the left, resulting in a size one matrix
#eval !![1, 2] *ᵥ ![1, 1]  -- ![3]

-- matrices acting on vectors on the right
#eval  ![1, 1, 1] ᵥ* !![1, 2; 3, 4; 5, 6] -- ![9, 12]
```

In order to generate matrices with identical rows or columns specified by a vector, we use `Matrix.replicateRow` and `Matrix.replicateCol`, with arguments the type indexing the rows or columns and the vector. For instance one can get single row or single column matrixes (more precisely matrices whose rows or columns are indexed by `Fin 1`).

```
#eval replicateRow (Fin 1) ![1, 2] -- !![1, 2]

#eval replicateCol (Fin 1) ![1, 2] -- !![1; 2]
```

Other familiar operations include the vector dot product, matrix transpose, and, for square matrices, determinant and trace.

```
-- vector dot product
#eval ![1, 2] ⬝ᵥ ![3, 4] -- `11`

-- matrix transpose
#eval !![1, 2; 3, 4]ᵀ -- `!![1, 3; 2, 4]`

-- determinant
#eval !![(1 : ℤ), 2; 3, 4].det -- `-2`

-- trace
#eval !![(1 : ℤ), 2; 3, 4].trace -- `5`
```

When entries do not have a computable type, for instance if they are real numbers, we cannot hope that `#eval` can help. Also this kind of evaluation cannot be used in proofs without considerably expanding the trusted code base (i.e. the part of Lean that you need to trust when checking proofs).

So it is good to also use the `simp` and `norm_num` tactics in proofs, or their command counter-part for quick exploration.

```
#simp !![(1 : ℝ), 2; 3, 4].det -- `4 - 2*3`

#norm_num !![(1 : ℝ), 2; 3, 4].det -- `-2`

#norm_num !![(1 : ℝ), 2; 3, 4].trace -- `5`

variable (a b c d : ℝ) in
#simp !![a, b; c, d].det -- `a * d - b * c`
```

The next important operation on square matrices is inversion. In the same way as division of numbers is always defined and returns the artificial value zero for division by zero, the inversion operation is defined on all matrices and returns the zero matrix for non-invertible matrices.

More precisely, there is general function `Ring.inverse` that does this in any ring, and, for any matrix `A`, A^{-1} is defined as `Ring.inverse A.det • A.adjugate`. According to Cramer's rule, this is indeed the inverse of `A` when the determinant of `A` is not zero.

```
#norm_num [Matrix.inv_def] !![(1 : ℝ), 2; 3, 4]⁻¹ -- !![-2, 1; 3 / 2, -(1 / 2)]
```

Of course this definition is really useful only for invertible matrices. There is a general type class `Invertible` that helps recording this. For instance, the `simp` call in the next example will use the `inv_mul_of_invertible` lemma which has an `Invertible` type-class assumption, so it will trigger only if this can be found by the type-class synthesis system. Here we make this fact available using a `have` statement.

```
example : !![(1 : ℝ), 2; 3, 4]⁻¹ * !![(1 : ℝ), 2; 3, 4] = 1 := by
  have : Invertible !![(1 : ℝ), 2; 3, 4] := by
    apply Matrix.invertibleOfIsUnitDet
    norm_num
  simp
```

In this fully concrete case, we could also use the `norm_num` machinery, and `apply?` to find the final line:

```
example : !![(1 : ℝ), 2; 3, 4]⁻¹ * !![(1 : ℝ), 2; 3, 4] = 1 := by
  norm_num [Matrix.inv_def]
  exact one_fin_two.symm
```

All the concrete matrices above have their rows and columns indexed by `Fin n` for some `n` (not necessarily the same for rows and columns). But sometimes it is more convenient to index matrices using arbitrary finite types. For instance the adjacency matrix of a finite graph has rows and columns naturally indexed by the vertices of the graph.

In fact when simply wants to define matrices without defining any operation on them, finiteness of the indexing types are not even needed, and coefficients can have any type, without any algebraic structure. So Mathlib simply defines `Matrix m n α` to be `m → n → α` for any types `m`, `n` and α, and the matrices we have been using so far had types such as `Matrix (Fin 2) (Fin 2) ℝ`. Of course algebraic operations require more assumptions on `m`, `n` and α.

Note the main reason why we do not use `m → n → α` directly is to allow the type class system to understand what we want. For instance, for a ring `R`, the type `n → R` is endowed with the point-wise multiplication operation, and similarly `m → n → R` has this operation which is *not* the multiplication we want on matrices.

In the first example below, we force Lean to see through the definition of `Matrix` and accept the statement as meaningful, and then prove it by checking all entries.

But then the next two examples reveal that Lean uses the point-wise multiplication on `Fin 2 → Fin 2 → ℤ` but the matrix multiplication on `Matrix (Fin 2) (Fin 2) ℤ`.

```
section

example : (fun _ ↦ 1 : Fin 2 → Fin 2 → ℤ) = !![1, 1; 1, 1] := by
```

(continues on next page)

(continued from previous page)

```
  ext i j
  fin_cases i <;> fin_cases j <;> rfl

example : (fun _ ↦ 1 : Fin 2 → Fin 2 → ℤ) * (fun _ ↦ 1 : Fin 2 → Fin 2 → ℤ) = !!
↪[1, 1; 1, 1] := by
  ext i j
  fin_cases i <;> fin_cases j <;> rfl

example : !![1, 1; 1, 1] * !![1, 1; 1, 1] = !![2, 2; 2, 2] := by
  norm_num
```

In order to define matrices as functions without losing the benefits of `Matrix` for type class synthesis, we can use the equivalence `Matrix.of` between functions and matrices. This equivalence is secretly defined using `Equiv.refl`.

For instance we can define Vandermonde matrices corresponding to a vector `v`.

```
example {n : ℕ} (v : Fin n → ℝ) :
    Matrix.vandermonde v = Matrix.of (fun i j : Fin n ↦ v i ^ (j : ℕ)) :=
  rfl
end
end matrices
```

10.4.2 Bases

We now want to discuss bases of vector spaces. Informally there are many ways to define this notion. One can use a universal property. One can say a basis is a family of vectors that is linearly independent and spanning. Or one can combine those properties and directly say that a basis is a family of vectors such that every vectors can be written uniquely as a linear combination of bases vectors. Yet another way to say it is that a basis provides a linear isomorphism with a power of the base field `K`, seen as a vector space over `K`.

This isomorphism version is actually the one that Mathlib uses as a definition under the hood, and other characterizations are proven from it. One must be slightly careful with the "power of `K`" idea in the case of infinite bases. Indeed only finite linear combinations make sense in this algebraic context. So what we need as a reference vector space is not a direct product of copies of `K` but a direct sum. We could use `⨁ i : ι, K` for some type ι indexing the basis But we rather use the more specialized spelling $\iota \to_0$ `K` which means "functions from ι to `K` with finite support", i.e. functions which vanish outside a finite set in ι (this finite set is not fixed, it depends on the function). Evaluating such a function coming from a basis `B` at a vector `v` and `i : ι` returns the component (or coordinate) of `v` on the `i`-th basis vector.

The type of bases indexed by a type ι of `V` as a `K` vector space is `Basis ι K V`. The isomorphism is called `Basis.repr`.

```
variable {K : Type*} [Field K] {V : Type*} [AddCommGroup V] [Module K V]

section

variable {ι : Type*} (B : Basis ι K V) (v : V) (i : ι)

-- The basis vector with index ``i``
#check (B i : V)

-- the linear isomorphism with the model space given by ``B``
#check (B.repr : V ≃ₗ[K] ι →₀ K)

-- the component function of ``v``
#check (B.repr v : ι →₀ K)
```

(continues on next page)

(continued from previous page)

```
-- the component of ``v`` with index ``i``
#check (B.repr v i : K)
```

Instead of starting with such an isomorphism, one can start with a family `b` of vectors that is linearly independent and spanning, this is `Basis.mk`.

The assumption that the family is spanning is spelled out as `⊤ ≤ Submodule.span K (Set.range b)`. Here `⊤` is the top submodule of `V`, i.e. `V` seen as submodule of itself. This spelling looks a bit tortuous, but we will see below that it is almost equivalent by definition to the more readable `∀ v, v ∈ Submodule.span K (Set.range b)` (the underscores in the snippet below refers to the useless information `v ∈ ⊤`).

```
noncomputable example (b : ι → V) (b_indep : LinearIndependent K b)
    (b_spans : ∀ v, v ∈ Submodule.span K (Set.range b)) : Basis ι K V :=
  Basis.mk b_indep (fun v _ ↦ b_spans v)

-- The family of vectors underlying the above basis is indeed ``b``.
example (b : ι → V) (b_indep : LinearIndependent K b)
    (b_spans : ∀ v, v ∈ Submodule.span K (Set.range b)) (i : ι) :
    Basis.mk b_indep (fun v _ ↦ b_spans v) i = b i :=
  Basis.mk_apply b_indep (fun v _ ↦ b_spans v) i
```

In particular the model vector space `ι →₀ K` has a so-called canonical basis whose `repr` function evaluated on any vector is the identity isomorphism. It is called `Finsupp.basisSingleOne` where `Finsupp` means function with finite support and `basisSingleOne` refers to the fact that basis vectors are functions which vanish expect for a single input value. More precisely the basis vector indexed by `i : ι` is `Finsupp.single i 1` which is the finitely supported function taking value `1` at `i` and `0` everywhere else.

```
variable [DecidableEq ι]
```

```
example : Finsupp.basisSingleOne.repr = LinearEquiv.refl K (ι →₀ K) :=
  rfl

example (i : ι) : Finsupp.basisSingleOne i = Finsupp.single i 1 :=
  rfl
```

The story of finitely supported functions is unneeded when the indexing type is finite. In this case we can use the simpler `Pi.basisFun` which gives a basis of the whole `ι → K`.

```
example [Finite ι] (x : ι → K) (i : ι) : (Pi.basisFun K ι).repr x i = x i := by
  simp
```

Going back to the general case of bases of abstract vector spaces, we can express any vector as a linear combination of basis vectors. Let us first see the easy case of finite bases.

```
example [Fintype ι] : Σ i : ι, B.repr v i • (B i) = v :=
  B.sum_repr v
```

When ι is not finite, the above statement makes no sense a priori: we cannot take a sum over ι. However the support of the function being summed is finite (it is the support of `B.repr v`). But we need to apply a construction that takes this into account. Here Mathlib uses a special purpose function that requires some time to get used to: `Finsupp.linearCombination` (which is built on top of the more general `Finsupp.sum`). Given a finitely supported function `c` from a type ι to the base field `K` and any function `f` from ι to `V`, `Finsupp.linearCombination K f c` is the sum over the support of `c` of the scalar multiplication `c • f`. In particular, we can replace it by a sum over any finite set containing the support of `c`.

```
example (c : ι →₀ K) (f : ι → V) (s : Finset ι) (h : c.support ⊆ s) :
    Finsupp.linearCombination K f c = Σ i ∈ s, c i • f i :=
  Finsupp.linearCombination_apply_of_mem_supported K h
```

One could also assume that `f` is finitely supported and still get a well defined sum. But the choice made by `Finsupp.linearCombination` is the one relevant to our basis discussion since it allows to state the generalization of `Basis.sum_repr`.

```
example : Finsupp.linearCombination K B (B.repr v) = v :=
  B.linearCombination_repr v
```

One could wonder why `K` is an explicit argument here, despite the fact it can be inferred from the type of `c`. The point is that the partially applied `Finsupp.linearCombination K f` is interesting in itself. It is not a bare function from `ι →₀ K` to `V` but a `K`-linear map.

```
variable (f : ι → V) in
#check (Finsupp.linearCombination K f : (ι →₀ K) →ₗ[K] V)
```

Returning to the mathematical discussion, it is important to understand that the representation of vectors in a basis is less useful in formalized mathematics than you may think. Indeed it is very often more efficient to directly use more abstract properties of bases. In particular the universal property of bases connecting them to other free objects in algebra allows to construct linear maps by specifying the images of basis vectors. This is `Basis.constr`. For any `K`-vector space `W`, our basis `B` gives a linear isomorphism `Basis.constr B K` from `ι → W` to `V →ₗ[K] W`. This isomorphism is characterized by the fact that it sends any function `u : ι → W` to a linear map sending the basis vector `B i` to `u i`, for every `i : ι`.

```
section

variable {W : Type*} [AddCommGroup W] [Module K W]
         (φ : V →ₗ[K] W) (u : ι → W)

#check (B.constr K : (ι → W) ≃ₗ[K] (V →ₗ[K] W))

#check (B.constr K u : V →ₗ[K] W)

example (i : ι) : B.constr K u (B i) = u i :=
  B.constr_basis K u i
```

This property is indeed characteristic because linear maps are determined by their values on bases:

```
example (φ ψ : V →ₗ[K] W) (h : ∀ i, φ (B i) = ψ (B i)) : φ = ψ :=
  B.ext h
```

If we also have a basis `B'` on the target space then we can identify linear maps with matrices. This identification is a `K`-linear isomorphism.

```
variable {ι' : Type*} (B' : Basis ι' K W) [Fintype ι] [DecidableEq ι] [Fintype ι']␣
↪[DecidableEq ι']

open LinearMap

#check (toMatrix B B' : (V →ₗ[K] W) ≃ₗ[K] Matrix ι' ι K)

open Matrix -- get access to the ``*ᵥ`` notation for multiplication between matrices␣
↪and vectors.
```

(continues on next page)

(continued from previous page)

```
example (φ : V →ₗ[K] W) (v : V) : (toMatrix B B' φ) *ᵥ (B.repr v) = B'.repr (φ v) :=
  toMatrix_mulVec_repr B B' φ v


variable {ι'' : Type*} (B'' : Basis ι'' K W) [Fintype ι''] [DecidableEq ι'']

example (φ : V →ₗ[K] W) : (toMatrix B B'' φ) = (toMatrix B' B'' .id) * (toMatrix B B'
↪φ) := by
  simp

end
```

As an exercise on this topic, we will prove part of the theorem which guarantees that endomorphisms have a well-defined determinant. Namely we want to prove that when two bases are indexed by the same type, the matrices they attach to any endomorphism have the same determinant. This would then need to be complemented using that bases all have isomorphic indexing types to get the full result.

Of course Mathlib already knows this, and `simp` can close the goal immediately, so you shouldn't use it too soon, but rather use the provided lemmas.

```
open Module LinearMap Matrix

-- Some lemmas coming from the fact that `LinearMap.toMatrix` is an algebra morphism.
#check toMatrix_comp
#check id_comp
#check comp_id
#check toMatrix_id

-- Some lemmas coming from the fact that ``Matrix.det`` is a multiplicative monoid
↪morphism.
#check Matrix.det_mul
#check Matrix.det_one

example [Fintype ι] (B' : Basis ι K V) (φ : End K V) :
    (toMatrix B B φ).det = (toMatrix B' B' φ).det := by
  set M := toMatrix B B φ
  set M' := toMatrix B' B' φ
  set P := (toMatrix B B') LinearMap.id
  set P' := (toMatrix B' B) LinearMap.id
  sorry
end
```

10.4.3 Dimension

Returning to the case of a single vector space, bases are also useful to define the concept of dimension. Here again, there is the elementary case of finite-dimensional vector spaces. For such spaces we expect a dimension which is a natural number. This is `Module.finrank`. It takes the base field as an explicit argument since a given abelian group can be a vector space over different fields.

```
section
#check (Module.finrank K V : ℕ)

-- `Fin n → K` is the archetypical space with dimension `n` over `K`.
example (n : ℕ) : Module.finrank K (Fin n → K) = n :=
  Module.finrank_fin_fun K
```

(continues on next page)

(continued from previous page)

```
-- Seen as a vector space over itself, `ℂ` has dimension one.
example : Module.finrank ℂ ℂ = 1 :=
  Module.finrank_self ℂ

-- But as a real vector space it has dimension two.
example : Module.finrank ℝ ℂ = 2 :=
  Complex.finrank_real_complex
```

Note that `Module.finrank` is defined for any vector space. It returns zero for infinite dimensional vector spaces, just as division by zero returns zero.

Of course many lemmas require a finite dimension assumption. This is the role of the `FiniteDimensional` typeclass. For instance, think about how the next example fails without this assumption.

```
example [FiniteDimensional K V] : 0 < Module.finrank K V ↔ Nontrivial V  :=
  Module.finrank_pos_iff
```

In the above statement, `Nontrivial V` means `V` has at least two different elements. Note that `Module.finrank_pos_iff` has no explicit argument. This is fine when using it from left to right, but not when using it from right to left because Lean has no way to guess `K` from the statement `Nontrivial V`. In that case it is useful to use the name argument syntax, after checking that the lemma is stated over a ring named `R`. So we can write:

```
example [FiniteDimensional K V] (h : 0 < Module.finrank K V) : Nontrivial V := by
  apply (Module.finrank_pos_iff (R := K)).1
  exact h
```

The above spelling is strange because we already have `h` as an assumption, so we could just as well give the full proof `Module.finrank_pos_iff.1 h` but it is good to know for more complicated cases.

By definition, `FiniteDimensional K V` can be read from any basis.

```
variable {ι : Type*} (B : Basis ι K V)

example [Finite ι] : FiniteDimensional K V := FiniteDimensional.of_fintype_basis B

example [FiniteDimensional K V] : Finite ι :=
  (FiniteDimensional.fintypeBasisIndex B).finite
end
```

Using that the subtype corresponding to a linear subspace has a vector space structure, we can talk about the dimension of a subspace.

```
section
variable (E F : Submodule K V) [FiniteDimensional K V]

open Module

example : finrank K (E ⊔ F : Submodule K V) + finrank K (E ⊓ F : Submodule K V) =
    finrank K E + finrank K F :=
  Submodule.finrank_sup_add_finrank_inf_eq E F

example : finrank K E ≤ finrank K V := Submodule.finrank_le E
```

In the first statement above, the purpose of the type ascriptions is to make sure that coercion to `Type*` does not trigger too early.

We are now ready for an exercise about `finrank` and subspaces.

```
example (h : finrank K V < finrank K E + finrank K F) :
    Nontrivial (E ⊓ F : Submodule K V) := by
  sorry
end
```

Let us now move to the general case of dimension theory. In this case `finrank` is useless, but we still have that, for any two bases of the same vector space, there is a bijection between the types indexing those bases. So we can still hope to define the rank as a cardinal, i.e. an element of the "quotient of the collection of types under the existence of a bijection equivalence relation".

When discussing cardinal, it gets harder to ignore foundational issues around Russel's paradox like we do everywhere else in this book. There is no type of all types because it would lead to logical inconsistencies. This issue is solved by the hierarchy of universes that we usually try to ignore.

Each type has a universe level, and those levels behave similarly to natural numbers. In particular there is zeroth level, and the corresponding universe `Type 0` is simply denoted by `Type`. This universe is enough to hold almost all of classical mathematics. For instance $\mathbb{N}$ and $\mathbb{R}$ have type `Type`. Each level `u` has a successor denoted by `u + 1`, and `Type u` has type `Type (u+1)`.

But universe levels are not natural numbers, they have a really different nature and don't have a type. In particular you cannot state in Lean something like `u ≠ u + 1`. There is simply no type where this would take place. Even stating `Type u ≠ Type (u+1)` does not make any sense since `Type u` and `Type (u+1)` have different types.

Whenever we write `Type*`, Lean inserts a universe level variable named `u_n` where `n` is a number. This allows definitions and statements to live in all universes.

Given a universe level `u`, we can define an equivalence relation on `Type u` saying two types α and β are equivalent if there is a bijection between them. The quotient type `Cardinal.{u}` lives in `Type (u+1)`. The curly braces denote a universe variable. The image of α `: Type u` in this quotient is `Cardinal.mk` α `: Cardinal.{u}`.

But we cannot directly compare cardinals in different universes. So technically we cannot define the rank of a vector space `V` as the cardinal of all types indexing a basis of `V`. So instead it is defined as the supremum `Module.rank K V` of cardinals of all linearly independent sets in `V`. If `V` has universe level `u` then its rank has type `Cardinal.{u}`.

```
#check V -- Type u_2
#check Module.rank K V -- Cardinal.{u_2}
```

One can still relate this definition to bases. Indeed there is also a commutative `max` operation on universe levels, and given two universe levels `u` and `v` there is an operation `Cardinal.lift.{u, v} : Cardinal.{v} → Cardinal.{max v u}` that allows to put cardinals in a common universe and state the dimension theorem.

```
universe u v -- `u` and `v` will denote universe levels

variable {ι : Type u} (B : Basis ι K V)
         {ι' : Type v} (B' : Basis ι' K V)

example : Cardinal.lift.{v, u} (.mk ι) = Cardinal.lift.{u, v} (.mk ι') :=
  mk_eq_mk_of_basis B B'
```

We can relate the finite dimensional case to this discussion using the coercion from natural numbers to finite cardinals (or more precisely the finite cardinals which live in `Cardinal.{v}` where `v` is the universe level of `V`).

```
example [FiniteDimensional K V] :
    (Module.finrank K V : Cardinal) = Module.rank K V :=
  Module.finrank_eq_rank K V
```

CHAPTER

ELEVEN

TOPOLOGY

Calculus is based on the concept of a function, which is used to model quantities that depend on one another. For example, it is common to study quantities that change over time. The notion of a *limit* is also fundamental. We may say that the limit of a function $f(x)$ is a value b as x approaches a value a, or that $f(x)$ *converges to* b as x approaches a. Equivalently, we may say that $f(x)$ approaches b as x approaches a value a, or that it *tends to* b as x tends to a. We have already begun to consider such notions in Section 3.6.

Topology is the abstract study of limits and continuity. Having covered the essentials of formalization in Chapters 2 to 7, in this chapter, we will explain how topological notions are formalized in Mathlib. Not only do topological abstractions apply in much greater generality, but they also, somewhat paradoxically, make it easier to reason about limits and continuity in concrete instances.

Topological notions build on quite a few layers of mathematical structure. The first layer is naive set theory, as described in Chapter 4. The next layer is the theory of *filters*, which we will describe in Section 11.1. On top of that, we layer the theories of *topological spaces*, *metric spaces*, and a slightly more exotic intermediate notion called a *uniform space*.

Whereas previous chapters relied on mathematical notions that were likely familiar to you, the notion of a filter is less well known, even to many working mathematicians. The notion is essential, however, for formalizing mathematics effectively. Let us explain why. Let `f : ℝ → ℝ` be any function. We can consider the limit of `f x` as `x` approaches some value x_0, but we can also consider the limit of `f x` as `x` approaches infinity or negative infinity. We can moreover consider the limit of `f x` as `x` approaches x_0 from the right, conventionally written $x_0{}^+$, or from the left, written $x_0{}^-$. There are variations where `x` approaches x_0 or $x_0{}^+$ or $x_0{}^-$ but is not allowed to take on the value x_0 itself. This results in at least eight ways that `x` can approach something. We can also restrict to rational values of `x` or place other constraints on the domain, but let's stick to those 8 cases.

We have a similar variety of options on the codomain: we can specify that `f x` approaches a value from the left or right, or that it approaches positive or negative infinity, and so on. For example, we may wish to say that `f x` tends to $+\infty$ when `x` tends to x_0 from the right without being equal to x_0. This results in 64 different kinds of limit statements, and we haven't even begun to deal with limits of sequences, as we did in Section 3.6.

The problem is compounded even further when it comes to the supporting lemmas. For instance, limits compose: if `f x` tends to y_0 when `x` tends to x_0 and `g y` tends to z_0 when `y` tends to y_0 then `g ∘ f x` tends to z_0 when `x` tends to x_0. There are three notions of "tends to" at play here, each of which can be instantiated in any of the eight ways described in the previous paragraph. This results in 512 lemmas, a lot to have to add to a library! Informally, mathematicians generally prove two or three of these and simply note that the rest can be proved "in the same way." Formalizing mathematics requires making the relevant notion of "sameness" fully explicit, and that is exactly what Bourbaki's theory of filters manages to do.

11.1 Filters

A *filter* on a type `X` is a collection of sets of `X` that satisfies three conditions that we will spell out below. The notion supports two related ideas:

- *limits*, including all the kinds of limits discussed above: finite and infinite limits of sequences, finite and infinite limits of functions at a point or at infinity, and so on.
- *things happening eventually*, including things happening for large enough `n : ℕ`, or sufficiently near a point `x`, or for sufficiently close pairs of points, or almost everywhere in the sense of measure theory. Dually, filters can also express the idea of *things happening often*: for arbitrarily large `n`, at a point in any neighborhood of a given point, etc.

The filters that correspond to these descriptions will be defined later in this section, but we can already name them:

- `(atTop : Filter ℕ)`, made of sets of $\mathbb{N}$ containing `{n | n ≥ N}` for some `N`
- $\mathcal{N}$ `x`, made of neighborhoods of `x` in a topological space
- $\mathcal{U}$ `X`, made of entourages of a uniform space (uniform spaces generalize metric spaces and topological groups)
- μ`.ae`, made of sets whose complement has zero measure with respect to a measure μ.

The general definition is as follows: a filter `F : Filter X` is a collection of sets `F.sets : Set (Set X)` satisfying the following:

- `F.univ_sets : univ ∈ F.sets`
- `F.sets_of_superset : ∀ {U V}, U ∈ F.sets → U ⊆ V → V ∈ F.sets`
- `F.inter_sets : ∀ {U V}, U ∈ F.sets → V ∈ F.sets → U ∩ V ∈ F.sets`.

The first condition says that the set of all elements of `X` belongs to `F.sets`. The second condition says that if `U` belongs to `F.sets` then anything containing `U` also belongs to `F.sets`. The third condition says that `F.sets` is closed under finite intersections. In Mathlib, a filter `F` is defined to be a structure bundling `F.sets` and its three properties, but the properties carry no additional data, and it is convenient to blur the distinction between `F` and `F.sets`. We therefore define `U ∈ F` to mean `U ∈ F.sets`. This explains why the word `sets` appears in the names of some lemmas that that mention `U ∈ F`.

It may help to think of a filter as defining a notion of a "sufficiently large" set. The first condition then says that `univ` is sufficiently large, the second one says that a set containing a sufficiently large set is sufficiently large and the third one says that the intersection of two sufficiently large sets is sufficiently large.

It may be even more useful to think of a filter on a type `X` as a generalized element of `Set X`. For instance, `atTop` is the "set of very large numbers" and $\mathcal{N}$ `x₀` is the "set of points very close to `x₀`." One manifestation of this view is that we can associate to any `s : Set X` the so-called *principal filter* consisting of all sets that contain `s`. This definition is already in Mathlib and has a notation $\mathcal{P}$ (localized in the `Filter` namespace). For the purpose of demonstration, we ask you to take this opportunity to work out the definition here.

```
def principal {α : Type*} (s : Set α) : Filter α
    where
  sets := { t | s ⊆ t }
  univ_sets := sorry
  sets_of_superset := sorry
  inter_sets := sorry
```

For our second example, we ask you to define the filter `atTop : Filter ℕ`. (We could use any type with a preorder instead of $\mathbb{N}$.)

```
example : Filter ℕ :=
  { sets := { s | ∃ a, ∀ b, a ≤ b → b ∈ s }
    univ_sets := sorry
    sets_of_superset := sorry
    inter_sets := sorry }
```

We can also directly define the filter `𝓝 x` of neighborhoods of any `x : ℝ`. In the real numbers, a neighborhood of `x` is a set containing an open interval $(x_0 - \varepsilon, x_0 + \varepsilon)$, defined in Mathlib as `Ioo (x₀ - ε) (x₀ + ε)`. (This notion of a neighborhood is only a special case of a more general construction in Mathlib.)

With these examples, we can already define what it means for a function `f : X → Y` to converge to some `G : Filter Y` along some `F : Filter X`, as follows:

```
def Tendsto₁ {X Y : Type*} (f : X → Y) (F : Filter X) (G : Filter Y) :=
  ∀ V ∈ G, f ⁻¹' V ∈ F
```

When `X` is $\mathbb{N}$ and `Y` is $\mathbb{R}$, `Tendsto₁ u atTop (𝓝 x)` is equivalent to saying that the sequence `u : ℕ → ℝ` converges to the real number `x`. When both `X` and `Y` are $\mathbb{R}$, `Tendsto f (𝓝 x₀) (𝓝 y₀)` is equivalent to the familiar notion $\lim_{x \to x_0} f(x) = y_0$. All of the other kinds of limits mentioned in the introduction are also equivalent to instances of `Tendsto₁` for suitable choices of filters on the source and target.

The notion `Tendsto₁` above is definitionally equivalent to the notion `Tendsto` that is defined in Mathlib, but the latter is defined more abstractly. The problem with the definition of `Tendsto₁` is that it exposes a quantifier and elements of `G`, and it hides the intuition that we get by viewing filters as generalized sets. We can hide the quantifier `∀ V` and make the intuition more salient by using more algebraic and set-theoretic machinery. The first ingredient is the *pushforward* operation f_* associated to any map `f : X → Y`, denoted `Filter.map f` in Mathlib. Given a filter `F` on `X`, `Filter.map f F : Filter Y` is defined so that `V ∈ Filter.map f F ↔ f ⁻¹' V ∈ F` holds definitionally. In the example file we've opened the `Filter` namespace so that `Filter.map` can be written as `map`. This means that we can rewrite the definition of `Tendsto` using the order relation on `Filter Y`, which is reversed inclusion of the set of members. In other words, given `G H : Filter Y`, we have `G ≤ H ↔ ∀ V : Set Y, V ∈ H → V ∈ G`.

```
def Tendsto₂ {X Y : Type*} (f : X → Y) (F : Filter X) (G : Filter Y) :=
  map f F ≤ G

example {X Y : Type*} (f : X → Y) (F : Filter X) (G : Filter Y) :
    Tendsto₂ f F G ↔ Tendsto₁ f F G :=
  Iff.rfl
```

It may seem that the order relation on filters is backward. But recall that we can view filters on `X` as generalized elements of `Set X`, via the inclusion of `𝒫 : Set X → Filter X` which maps any set `s` to the corresponding principal filter. This inclusion is order preserving, so the order relation on `Filter` can indeed be seen as the natural inclusion relation between generalized sets. In this analogy, pushforward is analogous to the direct image. And, indeed, `map f (𝒫 s) = 𝒫 (f '' s)`.

We can now understand intuitively why a sequence `u : ℕ → ℝ` converges to a point x_0 if and only if we have `map u atTop ≤ 𝓝 x₀`. The inequality means the "direct image under `u`" of "the set of very big natural numbers" is "included" in "the set of points very close to x_0."

As promised, the definition of `Tendsto₂` does not exhibit any quantifiers or sets. It also leverages the algebraic properties of the pushforward operation. First, each `Filter.map f` is monotone. And, second, `Filter.map` is compatible with composition.

```
#check (@Filter.map_mono : ∀ {α β} {m : α → β}, Monotone (map m))

#check
  (@Filter.map_map :
```

(continues on next page)

(continued from previous page)

```
    ∀ {α β γ} {f : Filter α} {m : α → β} {m' : β → γ}, map m' (map m f) = map (m' ∘
↪m) f)
```

Together these two properties allow us to prove that limits compose, yielding in one shot all 512 variants of the composition lemma described in the introduction, and lots more. You can practice proving the following statement using either the definition of `Tendsto₁` in terms of the universal quantifier or the algebraic definition, together with the two lemmas above.

```
example {X Y Z : Type*} {F : Filter X} {G : Filter Y} {H : Filter Z} {f : X → Y} {g :
↪Y → Z}
    (hf : Tendsto₁ f F G) (hg : Tendsto₁ g G H) : Tendsto₁ (g ∘ f) F H :=
  sorry
```

The pushforward construction uses a map to push filters from the map source to the map target. There also a *pullback* operation, `Filter.comap`, going in the other direction. This generalizes the preimage operation on sets. For any map `f`, `Filter.map f` and `Filter.comap f` form what is known as a *Galois connection*, which is to say, they satisfy

```
Filter.map_le_iff_le_comap : Filter.map f F ≤ G ↔ F ≤ Filter.comap f
G
```

for every `F` and `G`. This operation could be used to provided another formulation of `Tendsto` that would be provably (but not definitionally) equivalent to the one in Mathlib.

The `comap` operation can be used to restrict filters to a subtype. For instance, suppose we have `f : ℝ → ℝ`, `x₀ : ℝ` and `y₀ : ℝ`, and suppose we want to state that `f x` approaches `y₀` when `x` approaches `x₀` within the rational numbers. We can pull the filter `𝓝 x₀` back to `ℚ` using the coercion map `(↑) : ℚ → ℝ` and state `Tendsto (f ∘ (↑) : ℚ → ℝ) (comap (↑) (𝓝 x₀)) (𝓝 y₀)`.

```
variable (f : ℝ → ℝ) (x₀ y₀ : ℝ)

#check comap ((↑) : ℚ → ℝ) (𝓝 x₀)

#check Tendsto (f ∘ (↑)) (comap ((↑) : ℚ → ℝ) (𝓝 x₀)) (𝓝 y₀)
```

The pullback operation is also compatible with composition, but it is *contravariant*, which is to say, it reverses the order of the arguments.

```
section
variable {α β γ : Type*} (F : Filter α) {m : γ → β} {n : β → α}

#check (comap_comap : comap m (comap n F) = comap (n ∘ m) F)

end
```

Let's now shift attention to the plane `ℝ × ℝ` and try to understand how the neighborhoods of a point `(x₀, y₀)` are related to `𝓝 x₀` and `𝓝 y₀`. There is a product operation `Filter.prod : Filter X → Filter Y → Filter (X × Y)`, denoted by `×ˢ`, which answers this question:

```
example : 𝓝 (x₀, y₀) = 𝓝 x₀ ×ˢ 𝓝 y₀ :=
  nhds_prod_eq
```

The product operation is defined in terms of the pullback operation and the `inf` operation:

```
F ×ˢ G = (comap Prod.fst F) ⊓ (comap Prod.snd G).
```

Here the `inf` operation refers to the lattice structure on `Filter X` for any type `X`, whereby `F ⊓ G` is the greatest filter that is smaller than both `F` and `G`. Thus the `inf` operation generalizes the notion of the intersection of sets.

A lot of proofs in Mathlib use all of the aforementioned structure (`map`, `comap`, `inf`, `sup`, and `prod`) to give algebraic proofs about convergence without ever referring to members of filters. You can practice doing this in a proof of the following lemma, unfolding the definition of `Tendsto` and `Filter.prod` if needed.

```
#check le_inf_iff

example (f : ℕ → ℝ × ℝ) (x₀ y₀ : ℝ) :
    Tendsto f atTop (𝓝 (x₀, y₀)) ↔
      Tendsto (Prod.fst ∘ f) atTop (𝓝 x₀) ∧ Tendsto (Prod.snd ∘ f) atTop (𝓝 y₀) :=
  sorry
```

The ordered type `Filter X` is actually a *complete* lattice, which is to say, there is a bottom element, there is a top element, and every set of filters on `X` has an `Inf` and a `Sup`.

Note that given the second property in the definition of a filter (if `U` belongs to `F` then anything larger than `U` also belongs to `F`), the first property (the set of all inhabitants of `X` belongs to `F`) is equivalent to the property that `F` is not the empty collection of sets. This shouldn't be confused with the more subtle question as to whether the empty set is an *element* of `F`. The definition of a filter does not prohibit `∅ ∈ F`, but if the empty set is in `F` then every set is in `F`, which is to say, `∀ U : Set X, U ∈ F`. In this case, `F` is a rather trivial filter, which is precisely the bottom element of the complete lattice `Filter X`. This contrasts with the definition of filters in Bourbaki, which doesn't allow filters containing the empty set.

Because we include the trivial filter in our definition, we sometimes need to explicitly assume nontriviality in some lemmas. In return, however, the theory has nicer global properties. We have already seen that including the trivial filter gives us a bottom element. It also allows us to define `principal : Set X → Filter X`, which maps `∅` to `⊥`, without adding a precondition to rule out the empty set. And it allows us to define the pullback operation without a precondition as well. Indeed, it can happen that `comap f F = ⊥` although `F ≠ ⊥`. For instance, given `x₀ : ℝ` and `s : Set ℝ`, the pullback of `𝓝 x₀` under the coercion from the subtype corresponding to `s` is nontrivial if and only if `x₀` belongs to the closure of `s`.

In order to manage lemmas that do need to assume some filter is nontrivial, Mathlib has a type class `Filter.NeBot`, and the library has lemmas that assume `(F : Filter X) [F.NeBot]`. The instance database knows, for example, that `(atTop : Filter ℕ).NeBot`, and it knows that pushing forward a nontrivial filter gives a nontrivial filter. As a result, a lemma assuming `[F.NeBot]` will automatically apply to `map u atTop` for any sequence `u`.

Our tour of the algebraic properties of filters and their relation to limits is essentially done, but we have not yet justified our claim to have recaptured the usual limit notions. Superficially, it may seem that `Tendsto u atTop (𝓝 x₀)` is stronger than the notion of convergence defined in Section 3.6 because we ask that *every* neighborhood of `x₀` has a preimage belonging to `atTop`, whereas the usual definition only requires this for the standard neighborhoods `Ioo (x₀ - ε) (x₀ + ε)`. The key is that, by definition, every neighborhood contains such a standard one. This observation leads to the notion of a *filter basis*.

Given `F : Filter X`, a family of sets `s : ι → Set X` is a basis for `F` if for every set `U`, we have `U ∈ F` if and only if it contains some `s i`. In other words, formally speaking, `s` is a basis if it satisfies `∀ U : Set X, U ∈ F ↔ ∃ i, s i ⊆ U`. It is even more flexible to consider a predicate on `ι` that selects only some of the values `i` in the indexing type. In the case of `𝓝 x₀`, we want `ι` to be `ℝ`, we write `ε` for `i`, and the predicate should select the positive values of `ε`. So the fact that the sets `Ioo (x₀ - ε) (x₀ + ε)` form a basis for the neighborhood topology on `ℝ` is stated as follows:

```
example (x₀ : ℝ) : HasBasis (𝓝 x₀) (fun ε : ℝ ↦ 0 < ε) fun ε ↦ Ioo (x₀ - ε) (x₀ + ε)␣
↪:=
  nhds_basis_Ioo_pos x₀
```

There is also a nice basis for the filter `atTop`. The lemma `Filter.HasBasis.tendsto_iff` allows us to reformulate a statement of the form `Tendsto f F G` given bases for `F` and `G`. Putting these pieces together gives us essentially the notion of convergence that we used in Section 3.6.

```
example (u : ℕ → ℝ) (x₀ : ℝ) :
    Tendsto u atTop (𝓝 x₀) ↔ ∀ ε > 0, ∃ N, ∀ n ≥ N, u n ∈ Ioo (x₀ - ε) (x₀ + ε) := by
  have : atTop.HasBasis (fun _ : ℕ ↦ True) Ici := atTop_basis
  rw [this.tendsto_iff (nhds_basis_Ioo_pos x₀)]
  simp
```

We now show how filters facilitate working with properties that hold for sufficiently large numbers or for points that are sufficiently close to a given point. In Section 3.6, we were often faced with the situation where we knew that some property `P n` holds for sufficiently large `n` and that some other property `Q n` holds for sufficiently large `n`. Using `cases` twice gave us `N_P` and `N_Q` satisfying `∀ n ≥ N_P, P n` and `∀ n ≥ N_Q, Q n`. Using `set N := max N_P N_Q`, we could eventually prove `∀ n ≥ N, P n ∧ Q n`. Doing this repeatedly becomes tiresome.

We can do better by noting that the statement "`P n` and `Q n` hold for large enough `n`" means that we have `{n | P n} ∈ atTop` and `{n | Q n} ∈ atTop`. The fact that `atTop` is a filter implies that the intersection of two elements of `atTop` is again in `atTop`, so we have `{n | P n ∧ Q n} ∈ atTop`. Writing `{n | P n} ∈ atTop` is unpleasant, but we can use the more suggestive notation `∀ᶠ n in atTop, P n`. Here the superscripted `f` stands for "Filter." You can think of the notation as saying that for all `n` in the "set of very large numbers," `P n` holds. The `∀ᶠ` notation stands for `Filter.Eventually`, and the lemma `Filter.Eventually.and` uses the intersection property of filters to do what we just described:

```
example (P Q : ℕ → Prop) (hP : ∀ᶠ n in atTop, P n) (hQ : ∀ᶠ n in atTop, Q n) :
    ∀ᶠ n in atTop, P n ∧ Q n :=
  hP.and hQ
```

This notation is so convenient and intuitive that we also have specializations when `P` is an equality or inequality statement. For example, let `u` and `v` be two sequences of real numbers, and let us show that if `u n` and `v n` coincide for sufficiently large `n` then `u` tends to `x₀` if and only if `v` tends to `x₀`. First we'll use the generic `Eventually` and then the one specialized for the equality predicate, `EventuallyEq`. The two statements are definitionally equivalent so the same proof work in both cases.

```
example (u v : ℕ → ℝ) (h : ∀ᶠ n in atTop, u n = v n) (x₀ : ℝ) :
    Tendsto u atTop (𝓝 x₀) ↔ Tendsto v atTop (𝓝 x₀) :=
  tendsto_congr' h

example (u v : ℕ → ℝ) (h : u =ᶠ[atTop] v) (x₀ : ℝ) :
    Tendsto u atTop (𝓝 x₀) ↔ Tendsto v atTop (𝓝 x₀) :=
  tendsto_congr' h
```

It is instructive to review the definition of filters in terms of `Eventually`. Given `F : Filter X`, for any predicates `P` and `Q` on `X`,

- the condition `univ ∈ F` ensures `(∀ x, P x) → ∀ᶠ x in F, P x`,
- the condition `U ∈ F → U ⊆ V → V ∈ F` ensures `(∀ᶠ x in F, P x) → (∀ x, P x → Q x) → ∀ᶠ x in F, Q x`, and
- the condition `U ∈ F → V ∈ F → U ∩ V ∈ F` ensures `(∀ᶠ x in F, P x) → (∀ᶠ x in F, Q x) → ∀ᶠ x in F, P x ∧ Q x`.

```
#check Eventually.of_forall
#check Eventually.mono
#check Eventually.and
```

The second item, corresponding to `Eventually.mono`, supports nice ways of using filters, especially when combined with `Eventually.and`. The `filter_upwards` tactic allows us to combine them. Compare:

```
example (P Q R : ℕ → Prop) (hP : ∀ᶠ n in atTop, P n) (hQ : ∀ᶠ n in atTop, Q n)
    (hR : ∀ᶠ n in atTop, P n ∧ Q n → R n) : ∀ᶠ n in atTop, R n := by
  apply (hP.and (hQ.and hR)).mono
  rintro n ⟨h, h', h''⟩
  exact h'' ⟨h, h'⟩

example (P Q R : ℕ → Prop) (hP : ∀ᶠ n in atTop, P n) (hQ : ∀ᶠ n in atTop, Q n)
    (hR : ∀ᶠ n in atTop, P n ∧ Q n → R n) : ∀ᶠ n in atTop, R n := by
  filter_upwards [hP, hQ, hR] with n h h' h''
  exact h'' ⟨h, h'⟩
```

Readers who know about measure theory will note that the filter `μ.ae` of sets whose complement has measure zero (aka "the set consisting of almost every point") is not very useful as the source or target of `Tendsto`, but it can be conveniently used with `Eventually` to say that a property holds for almost every point.

There is a dual version of `∀ᶠ x in F, P x`, which is occasionally useful: `∃ᶠ x in F, P x` means `{x | ¬P x} ∉ F`. For example, `∃ᶠ n in atTop, P n` means there are arbitrarily large `n` such that `P n` holds. The `∃ᶠ` notation stands for `Filter.Frequently`.

For a more sophisticated example, consider the following statement about a sequence `u`, a set `M`, and a value `x`:

> If `u` converges to `x` and `u n` belongs to `M` for sufficiently large `n` then `x` is in the closure of `M`.

This can be formalized as follows:

```
Tendsto u atTop (𝓝 x) → (∀ᶠ n in atTop, u n ∈ M) → x ∈ closure M.
```

This is a special case of the theorem `mem_closure_of_tendsto` from the topology library. See if you can prove it using the quoted lemmas, using the fact that `ClusterPt x F` means `(𝓝 x ⊓ F).NeBot` and that, by definition, the assumption `∀ᶠ n in atTop, u n ∈ M` means `M ∈ map u atTop`.

```
#check mem_closure_iff_clusterPt
#check le_principal_iff
#check neBot_of_le

example (u : ℕ → ℝ) (M : Set ℝ) (x : ℝ) (hux : Tendsto u atTop (𝓝 x))
    (huM : ∀ᶠ n in atTop, u n ∈ M) : x ∈ closure M :=
  sorry
```

11.2 Metric spaces

Examples in the previous section focus on sequences of real numbers. In this section we will go up a bit in generality and focus on metric spaces. A metric space is a type `X` equipped with a distance function `dist : X → X → ℝ` which is a generalization of the function `fun x y ↦ |x - y|` from the case where `X = ℝ`.

Introducing such a space is easy and we will check all properties required from the distance function.

```
variable {X : Type*} [MetricSpace X] (a b c : X)

#check (dist a b : ℝ)
#check (dist_nonneg : 0 ≤ dist a b)
#check (dist_eq_zero : dist a b = 0 ↔ a = b)
#check (dist_comm a b : dist a b = dist b a)
#check (dist_triangle a b c : dist a c ≤ dist a b + dist b c)
```

Note we also have variants where the distance can be infinite or where `dist a b` can be zero without having `a = b` or both. They are called `EMetricSpace`, `PseudoMetricSpace` and `PseudoEMetricSpace` respectively (here "e" stands for "extended").

Note that our journey from $\mathbb{R}$ to metric spaces jumped over the special case of normed spaces that also require linear algebra and will be explained as part of the calculus chapter.

11.2.1 Convergence and continuity

Using distance functions, we can already define convergent sequences and continuous functions between metric spaces. They are actually defined in a more general setting covered in the next section, but we have lemmas recasting the definition in terms of distances.

```
example {u : ℕ → X} {a : X} :
    Tendsto u atTop (𝓝 a) ↔ ∀ ε > 0, ∃ N, ∀ n ≥ N, dist (u n) a < ε :=
  Metric.tendsto_atTop

example {X Y : Type*} [MetricSpace X] [MetricSpace Y] {f : X → Y} :
    Continuous f ↔
      ∀ x : X, ∀ ε > 0, ∃ δ > 0, ∀ x', dist x' x < δ → dist (f x') (f x) < ε :=
  Metric.continuous_iff
```

A *lot* of lemmas have some continuity assumptions, so we end up proving a lot of continuity results and there is a `continuity` tactic devoted to this task. Let's prove a continuity statement that will be needed in an exercise below. Notice that Lean knows how to treat a product of two metric spaces as a metric space, so it makes sense to consider continuous functions from `X × X` to $\mathbb{R}$. In particular the (uncurried version of the) distance function is such a function.

```
example {X Y : Type*} [MetricSpace X] [MetricSpace Y] {f : X → Y} (hf : Continuous f)␣
↪:
    Continuous fun p : X × X ↦ dist (f p.1) (f p.2) := by continuity
```

This tactic is a bit slow, so it is also useful to know how to do it by hand. We first need to use that `fun p : X × X ↦ f p.1` is continuous because it is the composition of `f`, which is continuous by assumption `hf`, and the projection `prod.fst` whose continuity is the content of the lemma `continuous_fst`. The composition property is `Continuous.comp` which is in the `Continuous` namespace so we can use dot notation to compress `Continuous.comp hf continuous_fst` into `hf.comp continuous_fst` which is actually more readable since it really reads as composing our assumption and our lemma. We can do the same for the second component to get continuity of `fun p : X × X ↦ f p.2`. We then assemble those two continuities using `Continuous.prod_mk` to get `(hf.comp continuous_fst).prod_mk (hf.comp continuous_snd) : Continuous (fun p : X × X ↦ (f p.1, f p.2))` and compose once more to get our full proof.

```
example {X Y : Type*} [MetricSpace X] [MetricSpace Y] {f : X → Y} (hf : Continuous f)␣
↪:
    Continuous fun p : X × X ↦ dist (f p.1) (f p.2) :=
  continuous_dist.comp ((hf.comp continuous_fst).prodMk (hf.comp continuous_snd))
```

The combination of `Continuous.prod_mk` and `continuous_dist` via `Continuous.comp` feels clunky, even when heavily using dot notation as above. A more serious issue is that this nice proof requires a lot of planning. Lean accepts the above proof term because it is a full term proving a statement which is definitionally equivalent to our goal, the crucial definition to unfold being that of a composition of functions. Indeed our target function `fun p : X × X ↦ dist (f p.1) (f p.2)` is not presented as a composition. The proof term we provided proves continuity of `dist ∘ (fun p : X × X ↦ (f p.1, f p.2))` which happens to be definitionally equal to our target function. But if we try to build this proof gradually using tactics starting with `apply continuous_dist.comp` then Lean's elaborator will fail to recognize a composition and refuse to apply this lemma. It is especially bad at this when products of types are involved.

A better lemma to apply here is `Continuous.dist {f g : X → Y} : Continuous f → Continuous g → Continuous (fun x ↦ dist (f x) (g x))` which is nicer to Lean's elaborator and also provides a shorter proof when directly providing a full proof term, as can be seen from the following two new proofs of the above statement:

```
example {X Y : Type*} [MetricSpace X] [MetricSpace Y] {f : X → Y} (hf : Continuous f)␣
↪:
    Continuous fun p : X × X ↦ dist (f p.1) (f p.2) := by
  apply Continuous.dist
  exact hf.comp continuous_fst
  exact hf.comp continuous_snd

example {X Y : Type*} [MetricSpace X] [MetricSpace Y] {f : X → Y} (hf : Continuous f)␣
↪:
    Continuous fun p : X × X ↦ dist (f p.1) (f p.2) :=
  (hf.comp continuous_fst).dist (hf.comp continuous_snd)
```

Note that, without the elaboration issue coming from composition, another way to compress our proof would be to use `Continuous.prod_map` which is sometimes useful and gives as an alternate proof term `continuous_dist.comp (hf.prod_map hf)` which even shorter to type.

Since it is sad to decide between a version which is better for elaboration and a version which is shorter to type, let us wrap this discussion with a last bit of compression offered by `Continuous.fst'` which allows to compress `hf.comp continuous_fst` to `hf.fst'` (and the same with `snd`) and get our final proof, now bordering obfuscation.

```
example {X Y : Type*} [MetricSpace X] [MetricSpace Y] {f : X → Y} (hf : Continuous f)␣
↪:
    Continuous fun p : X × X ↦ dist (f p.1) (f p.2) :=
  hf.fst'.dist hf.snd'
```

It's your turn now to prove some continuity lemma. After trying the continuity tactic, you will need `Continuous.add`, `continuous_pow` and `continuous_id` to do it by hand.

```
example {f : ℝ → X} (hf : Continuous f) : Continuous fun x : ℝ ↦ f (x ^ 2 + x) :=
  sorry
```

So far we saw continuity as a global notion, but one can also define continuity at a point.

```
example {X Y : Type*} [MetricSpace X] [MetricSpace Y] (f : X → Y) (a : X) :
    ContinuousAt f a ↔ ∀ ε > 0, ∃ δ > 0, ∀ {x}, dist x a < δ → dist (f x) (f a) < ε :=
  Metric.continuousAt_iff
```

11.2.2 Balls, open sets and closed sets

Once we have a distance function, the most important geometric definitions are (open) balls and closed balls.

```
variable (r : ℝ)

example : Metric.ball a r = { b | dist b a < r } :=
  rfl

example : Metric.closedBall a r = { b | dist b a ≤ r } :=
  rfl
```

Note that r is any real number here, there is no sign restriction. Of course some statements do require a radius condition.

```
example (hr : 0 < r) : a ∈ Metric.ball a r :=
  Metric.mem_ball_self hr

example (hr : 0 ≤ r) : a ∈ Metric.closedBall a r :=
  Metric.mem_closedBall_self hr
```

Once we have balls, we can define open sets. They are actually defined in a more general setting covered in the next section, but we have lemmas recasting the definition in terms of balls.

```
example (s : Set X) : IsOpen s ↔ ∀ x ∈ s, ∃ ε > 0, Metric.ball x ε ⊆ s :=
  Metric.isOpen_iff
```

Then closed sets are sets whose complement is open. Their important property is they are closed under limits. The closure of a set is the smallest closed set containing it.

```
example {s : Set X} : IsClosed s ↔ IsOpen (sᶜ) :=
  isOpen_compl_iff.symm

example {s : Set X} (hs : IsClosed s) {u : ℕ → X} (hu : Tendsto u atTop (𝓝 a))
    (hus : ∀ n, u n ∈ s) : a ∈ s :=
  hs.mem_of_tendsto hu (Eventually.of_forall hus)

example {s : Set X} : a ∈ closure s ↔ ∀ ε > 0, ∃ b ∈ s, a ∈ Metric.ball b ε :=
  Metric.mem_closure_iff
```

Do the next exercise without using *mem_closure_iff_seq_limit*

```
example {u : ℕ → X} (hu : Tendsto u atTop (𝓝 a)) {s : Set X} (hs : ∀ n, u n ∈ s) :
    a ∈ closure s := by
  sorry
```

Remember from the filters sections that neighborhood filters play a big role in Mathlib. In the metric space context, the crucial point is that balls provide bases for those filters. The main lemmas here are `Metric.nhds_basis_ball` and `Metric.nhds_basis_closedBall` that claim this for open and closed balls with positive radius. The center point is an implicit argument so we can invoke `Filter.HasBasis.mem_iff` as in the following example.

```
example {x : X} {s : Set X} : s ∈ 𝓝 x ↔ ∃ ε > 0, Metric.ball x ε ⊆ s :=
  Metric.nhds_basis_ball.mem_iff

example {x : X} {s : Set X} : s ∈ 𝓝 x ↔ ∃ ε > 0, Metric.closedBall x ε ⊆ s :=
  Metric.nhds_basis_closedBall.mem_iff
```

11.2.3 Compactness

Compactness is an important topological notion. It distinguishes subsets of a metric space that enjoy the same kind of properties as segments in the reals compared to other intervals:

- Any sequence with values in a compact set has a subsequence that converges in this set.
- Any continuous function on a nonempty compact set with values in real numbers is bounded and attains its bounds somewhere (this is called the extreme value theorem).
- Compact sets are closed sets.

Let us first check that the unit interval in the reals is indeed a compact set, and then check the above claims for compact sets in general metric spaces. In the second statement we only need continuity on the given set so we will use `ContinuousOn`

instead of `Continuous`, and we will give separate statements for the minimum and the maximum. Of course all these results are deduced from more general versions, some of which will be discussed in later sections.

```
example : IsCompact (Set.Icc 0 1 : Set ℝ) :=
  isCompact_Icc

example {s : Set X} (hs : IsCompact s) {u : ℕ → X} (hu : ∀ n, u n ∈ s) :
    ∃ a ∈ s, ∃ φ : ℕ → ℕ, StrictMono φ ∧ Tendsto (u ∘ φ) atTop (𝓝 a) :=
  hs.tendsto_subseq hu

example {s : Set X} (hs : IsCompact s) (hs' : s.Nonempty) {f : X → ℝ}
      (hfs : ContinuousOn f s) :
    ∃ x ∈ s, ∀ y ∈ s, f x ≤ f y :=
  hs.exists_isMinOn hs' hfs

example {s : Set X} (hs : IsCompact s) (hs' : s.Nonempty) {f : X → ℝ}
      (hfs : ContinuousOn f s) :
    ∃ x ∈ s, ∀ y ∈ s, f y ≤ f x :=
  hs.exists_isMaxOn hs' hfs

example {s : Set X} (hs : IsCompact s) : IsClosed s :=
  hs.isClosed
```

We can also specify that a metric spaces is globally compact, using an extra `Prop`-valued type class:

```
example {X : Type*} [MetricSpace X] [CompactSpace X] : IsCompact (univ : Set X) :=
  isCompact_univ
```

In a compact metric space any closed set is compact, this is `IsClosed.isCompact`.

11.2.4 Uniformly continuous functions

We now turn to uniformity notions on metric spaces : uniformly continuous functions, Cauchy sequences and completeness. Again those are defined in a more general context but we have lemmas in the metric name space to access their elementary definitions. We start with uniform continuity.

```
example {X : Type*} [MetricSpace X] {Y : Type*} [MetricSpace Y] {f : X → Y} :
    UniformContinuous f ↔
      ∀ ε > 0, ∃ δ > 0, ∀ {a b : X}, dist a b < δ → dist (f a) (f b) < ε :=
  Metric.uniformContinuous_iff
```

In order to practice manipulating all those definitions, we will prove that continuous functions from a compact metric space to a metric space are uniformly continuous (we will see a more general version in a later section).

We will first give an informal sketch. Let `f : X → Y` be a continuous function from a compact metric space to a metric space. We fix ε `> 0` and start looking for some δ.

Let φ `: X × X → ℝ := fun p ↦ dist (f p.1) (f p.2)` and let `K := { p : X × X | ε ≤ φ p }`. Observe φ is continuous since `f` and distance are continuous. And `K` is clearly closed (use `isClosed_le`) hence compact since `X` is compact.

Then we discuss two possibilities using `eq_empty_or_nonempty`. If `K` is empty then we are clearly done (we can set δ `= 1` for instance). So let's assume `K` is not empty, and use the extreme value theorem to choose `(x₀, x₁)` attaining the infimum of the distance function on `K`. We can then set δ `= dist x₀ x₁` and check everything works.

```
example {X : Type*} [MetricSpace X] [CompactSpace X]
      {Y : Type*} [MetricSpace Y] {f : X → Y}
```

(continues on next page)

(continued from previous page)

```
    (hf : Continuous f) : UniformContinuous f := by
  sorry
```

11.2.5 Completeness

A Cauchy sequence in a metric space is a sequence whose terms get closer and closer to each other. There are a couple of equivalent ways to state that idea. In particular converging sequences are Cauchy. The converse is true only in so-called *complete* spaces.

```
example (u : ℕ → X) :
    CauchySeq u ↔ ∀ ε > 0, ∃ N : ℕ, ∀ m ≥ N, ∀ n ≥ N, dist (u m) (u n) < ε :=
  Metric.cauchySeq_iff

example (u : ℕ → X) :
    CauchySeq u ↔ ∀ ε > 0, ∃ N : ℕ, ∀ n ≥ N, dist (u n) (u N) < ε :=
  Metric.cauchySeq_iff'

example [CompleteSpace X] (u : ℕ → X) (hu : CauchySeq u) :
    ∃ x, Tendsto u atTop (𝓝 x) :=
  cauchySeq_tendsto_of_complete hu
```

We'll practice using this definition by proving a convenient criterion which is a special case of a criterion appearing in Mathlib. This is also a good opportunity to practice using big sums in a geometric context. In addition to the explanations from the filters section, you will probably need `tendsto_pow_atTop_nhds_zero_of_lt_one`, `Tendsto.mul` and `dist_le_range_sum_dist`.

```
theorem cauchySeq_of_le_geometric_two' {u : ℕ → X}
    (hu : ∀ n : ℕ, dist (u n) (u (n + 1)) ≤ (1 / 2) ^ n) : CauchySeq u := by
  rw [Metric.cauchySeq_iff']
  intro ε ε_pos
  obtain ⟨N, hN⟩ : ∃ N : ℕ, 1 / 2 ^ N * 2 < ε := by sorry
  use N
  intro n hn
  obtain ⟨k, rfl : n = N + k⟩ := le_iff_exists_add.mp hn
  calc
    dist (u (N + k)) (u N) = dist (u (N + 0)) (u (N + k)) := sorry
    _ ≤ Σ i  ∈ range k, dist (u (N + i)) (u (N + (i + 1))) := sorry
    _ ≤ Σ i  ∈ range k, (1 / 2 : ℝ) ^ (N + i) := sorry
    _ = 1 / 2 ^ N * Σ i  ∈ range k, (1 / 2 : ℝ) ^ i := sorry
    _ ≤ 1 / 2 ^ N * 2 := sorry
    _ < ε := sorry
```

We are ready for the final boss of this section: Baire's theorem for complete metric spaces! The proof skeleton below shows interesting techniques. It uses the `choose` tactic in its exclamation mark variant (you should experiment with removing this exclamation mark) and it shows how to define something inductively in the middle of a proof using `Nat.rec_on`.

```
open Metric

example [CompleteSpace X] (f : ℕ → Set X) (ho : ∀ n, IsOpen (f n)) (hd : ∀ n, Dense␣
↪(f n)) :
    Dense (⋂ n, f n) := by
  let B : ℕ → ℝ := fun n ↦ (1 / 2) ^ n
  have Bpos : ∀ n, 0 < B n
```

(continues on next page)

(continued from previous page)

```
  sorry
  /- Translate the density assumption into two functions `center` and `radius`␣
↪associating
     to any n, x, δ, δpos a center and a positive radius such that
     `closedBall center radius` is included both in `f n` and in `closedBall x δ`.
     We can also require `radius ≤ (1/2)^(n+1)`, to ensure we get a Cauchy sequence␣
↪later. -/
  have :
    ∀ (n : ℕ) (x : X),
      ∀ δ > 0, ∃ y : X, ∃ r > 0, r ≤ B (n + 1) ∧ closedBall y r ⊆ closedBall x δ ∩ f␣
↪n :=
    by sorry
  choose! center radius Hpos HB Hball using this
  intro x
  rw [mem_closure_iff_nhds_basis nhds_basis_closedBall]
  intro ε εpos
  /- `ε` is positive. We have to find a point in the ball of radius `ε` around `x`
     belonging to all `f n`. For this, we construct inductively a sequence
     `F n = (c n, r n)` such that the closed ball `closedBall (c n) (r n)` is included
     in the previous ball and in `f n`, and such that `r n` is small enough to ensure
     that `c n` is a Cauchy sequence. Then `c n` converges to a limit which belongs
     to all the `f n`. -/
  let F : ℕ → X × ℝ := fun n ↦
    Nat.recOn n (Prod.mk x (min ε (B 0)))
      fun n p ↦ Prod.mk (center n p.1 p.2) (radius n p.1 p.2)
  let c : ℕ → X := fun n ↦ (F n).1
  let r : ℕ → ℝ := fun n ↦ (F n).2
  have rpos : ∀ n, 0 < r n := by sorry
  have rB : ∀ n, r n ≤ B n := by sorry
  have incl : ∀ n, closedBall (c (n + 1)) (r (n + 1)) ⊆ closedBall (c n) (r n) ∩ f n␣
↪:= by
    sorry
  have cdist : ∀ n, dist (c n) (c (n + 1)) ≤ B n := by sorry
  have : CauchySeq c := cauchySeq_of_le_geometric_two' cdist
  -- as the sequence `c n` is Cauchy in a complete space, it converges to a limit `y`.
  rcases cauchySeq_tendsto_of_complete this with ⟨y, ylim⟩
  -- this point `y` will be the desired point. We will check that it belongs to all
  -- `f n` and to `ball x ε`.
  use y
  have I : ∀ n, ∀ m ≥ n, closedBall (c m) (r m) ⊆ closedBall (c n) (r n) := by sorry
  have yball : ∀ n, y ∈ closedBall (c n) (r n) := by sorry
  sorry
```

11.3 Topological spaces

11.3.1 Fundamentals

We now go up in generality and introduce topological spaces. We will review the two main ways to define topological spaces and then explain how the category of topological spaces is much better behaved than the category of metric spaces. Note that we won't be using Mathlib category theory here, only having a somewhat categorical point of view.

The first way to think about the transition from metric spaces to topological spaces is that we only remember the notion of open sets (or equivalently the notion of closed sets). From this point of view, a topological space is a type equipped with a collection of sets that are called open sets. This collection has to satisfy a number of axioms presented below (this

collection is slightly redundant but we will ignore that).

```
section
variable {X : Type*} [TopologicalSpace X]

example : IsOpen (univ : Set X) :=
  isOpen_univ

example : IsOpen (∅ : Set X) :=
  isOpen_empty

example {ι : Type*} {s : ι → Set X} (hs : ∀ i, IsOpen (s i)) : IsOpen (⋃ i, s i) :=
  isOpen_iUnion hs

example {ι : Type*} [Fintype ι] {s : ι → Set X} (hs : ∀ i, IsOpen (s i)) :
    IsOpen (⋂ i, s i) :=
  isOpen_iInter_of_finite hs
```

Closed sets are then defined as sets whose complement is open. A function between topological spaces is (globally) continuous if all preimages of open sets are open.

```
variable {Y : Type*} [TopologicalSpace Y]

example {f : X → Y} : Continuous f ↔ ∀ s, IsOpen s → IsOpen (f ⁻¹' s) :=
  continuous_def
```

With this definition we already see that, compared to metric spaces, topological spaces only remember enough information to talk about continuous functions: two topological structures on a type are the same if and only if they have the same continuous functions (indeed the identity function will be continuous in both direction if and only if the two structures have the same open sets).

However as soon as we move on to continuity at a point we see the limitations of the approach based on open sets. In Mathlib we frequently think of topological spaces as types equipped with a neighborhood filter `𝓝 x` attached to each point `x` (the corresponding function `X → Filter X` satisfies certain conditions explained further down). Remember from the filters section that these gadgets play two related roles. First `𝓝 x` is seen as the generalized set of points of `X` that are close to `x`. And then it is seen as giving a way to say, for any predicate `P : X → Prop`, that this predicate holds for points that are close enough to `x`. Let us state that `f : X → Y` is continuous at `x`. The purely filtery way is to say that the direct image under `f` of the generalized set of points that are close to `x` is contained in the generalized set of points that are close to `f x`. Recall this is spelled either `map f (𝓝 x) ≤ 𝓝 (f x)` or `Tendsto f (𝓝 x) (𝓝 (f x))`.

```
example {f : X → Y} {x : X} : ContinuousAt f x ↔ map f (𝓝 x) ≤ 𝓝 (f x) :=
  Iff.rfl
```

One can also spell it using both neighborhoods seen as ordinary sets and a neighborhood filter seen as a generalized set: "for any neighborhood `U` of `f x`, all points close to `x` are sent to `U`". Note that the proof is again `Iff.rfl`, this point of view is definitionally equivalent to the previous one.

```
example {f : X → Y} {x : X} : ContinuousAt f x ↔ ∀ U ∈ 𝓝 (f x), ∀ᶠ x in 𝓝 x, f x ∈␣
↪U :=
  Iff.rfl
```

We now explain how to go from one point of view to the other. In terms of open sets, we can simply define members of `𝓝 x` as sets that contain an open set containing `x`.

```
example {x : X} {s : Set X} : s ∈ 𝓝 x ↔ ∃ t, t ⊆ s ∧ IsOpen t ∧ x ∈ t :=
  mem_nhds_iff
```

To go in the other direction we need to discuss the condition that `𝓝 : X → Filter X` must satisfy in order to be the neighborhood function of a topology.

The first constraint is that `𝓝 x`, seen as a generalized set, contains the set `{x}` seen as the generalized set `pure x` (explaining this weird name would be too much of a digression, so we simply accept it for now). Another way to say it is that if a predicate holds for points close to `x` then it holds at `x`.

```
example (x : X) : pure x ≤ 𝓝 x :=
  pure_le_nhds x

example (x : X) (P : X → Prop) (h : ∀ᶠ y in 𝓝 x, P y) : P x :=
  h.self_of_nhds
```

Then a more subtle requirement is that, for any predicate `P : X → Prop` and any `x`, if `P y` holds for `y` close to `x` then for `y` close to `x` and `z` close to `y`, `P z` holds. More precisely we have:

```
example {P : X → Prop} {x : X} (h : ∀ᶠ y in 𝓝 x, P y) : ∀ᶠ y in 𝓝 x, ∀ᶠ z in 𝓝 y, P␣
↪z :=
  eventually_eventually_nhds.mpr h
```

Those two results characterize the functions `X → Filter X` that are neighborhood functions for a topological space structure on `X`. There is a still a function `TopologicalSpace.mkOfNhds : (X → Filter X) → TopologicalSpace X` but it will give back its input as a neighborhood function only if it satisfies the above two constraints. More precisely we have a lemma `TopologicalSpace.nhds_mkOfNhds` saying that in a different way and our next exercise deduces this different way from how we stated it above.

```
example {α : Type*} (n : α → Filter α) (H₀ : ∀ a, pure a ≤ n a)
    (H : ∀ a : α, ∀ p : α → Prop, (∀ᶠ x in n a, p x) → ∀ᶠ y in n a, ∀ᶠ x in n y, p␣
↪x) :
    ∀ a, ∀ s ∈ n a, ∃ t ∈ n a, t ⊆ s ∧ ∀ a' ∈ t, s ∈ n a' := by
  sorry
end
```

Note that `TopologicalSpace.mkOfNhds` is not so frequently used, but it still good to know in what precise sense the neighborhood filters is all there is in a topological space structure.

The next thing to know in order to efficiently use topological spaces in Mathlib is that we use a lot of formal properties of `TopologicalSpace : Type u → Type u`. From a purely mathematical point of view, those formal properties are a very clean way to explain how topological spaces solve issues that metric spaces have. From this point of view, the issues solved by topological spaces is that metric spaces enjoy very little functoriality, and have very bad categorical properties in general. This comes on top of the fact already discussed that metric spaces contain a lot of geometrical information that is not topologically relevant.

Let us focus on functoriality first. A metric space structure can be induced on a subset or, equivalently, it can be pulled back by an injective map. But that's pretty much everything. They cannot be pulled back by general map or pushed forward, even by surjective maps.

In particular there is no sensible distance to put on a quotient of a metric space or on an uncountable product of metric spaces. Consider for instance the type `ℝ → ℝ`, seen as a product of copies of `ℝ` indexed by `ℝ`. We would like to say that pointwise convergence of sequences of functions is a respectable notion of convergence. But there is no distance on `ℝ → ℝ` that gives this notion of convergence. Relatedly, there is no distance ensuring that a map `f : X → (ℝ → ℝ)` is continuous if and only if `fun x ↦ f x t` is continuous for every `t : ℝ`.

We now review the data used to solve all those issues. First we can use any map `f : X → Y` to push or pull topologies from one side to the other. Those two operations form a Galois connection.

```
variable {X Y : Type*}
```

(continues on next page)

(continued from previous page)

```
example (f : X → Y) : TopologicalSpace X → TopologicalSpace Y :=
  TopologicalSpace.coinduced f

example (f : X → Y) : TopologicalSpace Y → TopologicalSpace X :=
  TopologicalSpace.induced f

example (f : X → Y) (T_X : TopologicalSpace X) (T_Y : TopologicalSpace Y) :
    TopologicalSpace.coinduced f T_X ≤ T_Y ↔ T_X ≤ TopologicalSpace.induced f T_Y :=
  coinduced_le_iff_le_induced
```

Those operations are compatible with composition of functions. As usual, pushing forward is covariant and pulling back is contravariant, see `coinduced_compose` and `induced_compose`. On paper we will use notations f_*T for `TopologicalSpace.coinduced f T` and f^*T for `TopologicalSpace.induced f T`.

Then the next big piece is a complete lattice structure on `TopologicalSpace X` for any given structure. If you think of topologies as being primarily the data of open sets then you expect the order relation on `TopologicalSpace X` to come from `Set (Set X)`, i.e. you expect `t ≤ t'` if a set `u` is open for `t'` as soon as it is open for `t`. However we already know that Mathlib focuses on neighborhoods more than open sets so, for any `x : X` we want the map from topological spaces to neighborhoods `fun T : TopologicalSpace X ↦ @nhds X T x` to be order preserving. And we know the order relation on `Filter X` is designed to ensure an order preserving `principal : Set X → Filter X`, allowing to see filters as generalized sets. So the order relation we do use on `TopologicalSpace X` is opposite to the one coming from `Set (Set X)`.

```
example {T T' : TopologicalSpace X} : T ≤ T' ↔ ∀ s, T'.IsOpen s → T.IsOpen s :=
  Iff.rfl
```

Now we can recover continuity by combining the push-forward (or pull-back) operation with the order relation.

```
example (T_X : TopologicalSpace X) (T_Y : TopologicalSpace Y) (f : X → Y) :
    Continuous f ↔ TopologicalSpace.coinduced f T_X ≤ T_Y :=
  continuous_iff_coinduced_le
```

With this definition and the compatibility of push-forward and composition, we get for free the universal property that, for any topological space Z, a function $g : Y \to Z$ is continuous for the topology f_*T_X if and only if $g \circ f$ is continuous.

$$\begin{aligned} g \text{ continuous} &\Leftrightarrow g_*(f_*T_X) \leq T_Z \\ &\Leftrightarrow (g \circ f)_* T_X \leq T_Z \\ &\Leftrightarrow g \circ f \text{ continuous} \end{aligned}$$

```
example {Z : Type*} (f : X → Y) (T_X : TopologicalSpace X) (T_Z : TopologicalSpace Z)
      (g : Y → Z) :
    @Continuous Y Z (TopologicalSpace.coinduced f T_X) T_Z g ↔
      @Continuous X Z T_X T_Z (g ∘ f) := by
  rw [continuous_iff_coinduced_le, coinduced_compose, continuous_iff_coinduced_le]
```

So we already get quotient topologies (using the projection map as `f`). This wasn't using that `TopologicalSpace X` is a complete lattice for all `X`. Let's now see how all this structure proves the existence of the product topology by abstract non-sense. We considered the case of $\mathbb{R} \to \mathbb{R}$ above, but let's now consider the general case of `Π i, X i` for some `ι : Type*` and `X : ι → Type*`. We want, for any topological space `Z` and any function `f : Z → Π i, X i`, that `f` is continuous if and only if `(fun x ↦ x i) ∘ f` is continuous for all `i`. Let us explore that constraint "on

paper" using notation p_i for the projection `(fun (x : Π i, X i) ↦ x i)`:

$$\begin{aligned}(\forall i, p_i \circ f \text{ continuous}) &\Leftrightarrow \forall i, (p_i \circ f)_* T_Z \leq T_{X_i} \\ &\Leftrightarrow \forall i, (p_i)_* f_* T_Z \leq T_{X_i} \\ &\Leftrightarrow \forall i, f_* T_Z \leq (p_i)^* T_{X_i} \\ &\Leftrightarrow f_* T_Z \leq \inf \left[(p_i)^* T_{X_i}\right]\end{aligned}$$

So we see that what is the topology we want on `Π i, X i`:

```
example (ι : Type*) (X : ι → Type*) (T_X : ∀ i, TopologicalSpace (X i)) :
    (Pi.topologicalSpace : TopologicalSpace (∀ i, X i)) =
      ⨅ i, TopologicalSpace.induced (fun x ↦ x i) (T_X i) :=
  rfl
```

This ends our tour of how Mathlib thinks that topological spaces fix defects of the theory of metric spaces by being a more functorial theory and having a complete lattice structure for any fixed type.

11.3.2 Separation and countability

We saw that the category of topological spaces have very nice properties. The price to pay for this is existence of rather pathological topological spaces. There are a number of assumptions you can make on a topological space to ensure its behavior is closer to what metric spaces do. The most important is `T2Space`, also called "Hausdorff", that will ensure that limits are unique. A stronger separation property is `T3Space` that ensures in addition the *RegularSpace* property: each point has a basis of closed neighborhoods.

```
example [TopologicalSpace X] [T2Space X] {u : ℕ → X} {a b : X} (ha : Tendsto u atTop␣
↪(𝓝 a))
    (hb : Tendsto u atTop (𝓝 b)) : a = b :=
  tendsto_nhds_unique ha hb

example [TopologicalSpace X] [RegularSpace X] (a : X) :
    (𝓝 a).HasBasis (fun s : Set X ↦ s ∈ 𝓝 a ∧ IsClosed s) id :=
  closed_nhds_basis a
```

Note that, in every topological space, each point has a basis of open neighborhood, by definition.

```
example [TopologicalSpace X] {x : X} :
    (𝓝 x).HasBasis (fun t : Set X ↦ t ∈ 𝓝 x ∧ IsOpen t) id :=
  nhds_basis_opens' x
```

Our main goal is now to prove the basic theorem which allows extension by continuity. From Bourbaki's general topology book, I.8.5, Theorem 1 (taking only the non-trivial implication):

Let X be a topological space, A a dense subset of X, $f : A \to Y$ a continuous mapping of A into a T_3 space Y. If, for each x in X, $f(y)$ tends to a limit in Y when y tends to x while remaining in A then there exists a continuous extension φ of f to X.

Actually Mathlib contains a more general version of the above lemma, `IsDenseInducing.continuousAt_extend`, but we'll stick to Bourbaki's version here.

Remember that, given `A : Set X`, `↥A` is the subtype associated to `A`, and Lean will automatically insert that funny up arrow when needed. And the (inclusion) coercion map is `(↑) : A → X`. The assumption "tends to x while remaining in A" corresponds to the pull-back filter `comap (↑) (𝓝 x)`.

Let's first prove an auxiliary lemma, extracted to simplify the context (in particular we don't need Y to be a topological space here).

```
theorem aux {X Y A : Type*} [TopologicalSpace X] {c : A → X}
      {f : A → Y} {x : X} {F : Filter Y}
      (h : Tendsto f (comap c (𝓝 x)) F) {V' : Set Y} (V'_in : V' ∈ F) :
    ∃ V ∈ 𝓝 x, IsOpen V ∧ c ⁻¹' V ⊆ f ⁻¹' V' := by
  sorry
```

Let's now turn to the main proof of the extension by continuity theorem.

When Lean needs a topology on `↥A` it will automatically use the induced topology. The only relevant lemma is `nhds_induced (↑) : ∀ a : ↥A, 𝓝 a = comap (↑) (𝓝 ↑a)` (this is actually a general lemma about induced topologies).

The proof outline is:

The main assumption and the axiom of choice give a function φ such that `∀ x, Tendsto f (comap (↑) (𝓝 x)) (𝓝 (φ x))` (because `Y` is Hausdorff, φ is entirely determined, but we won't need that until we try to prove that φ indeed extends `f`).

Let's first prove φ is continuous. Fix any `x : X`. Since `Y` is regular, it suffices to check that for every *closed* neighborhood `V'` of φ `x`, `φ ⁻¹' V' ∈ 𝓝 x`. The limit assumption gives (through the auxiliary lemma above) some `V ∈ 𝓝 x` such `IsOpen V ∧ (↑) ⁻¹' V ⊆ f ⁻¹' V'`. Since `V ∈ 𝓝 x`, it suffices to prove `V ⊆ φ ⁻¹' V'`, i.e. `∀ y ∈ V, φ y ∈ V'`. Let's fix `y` in `V`. Because `V` is *open*, it is a neighborhood of `y`. In particular `(↑) ⁻¹' V ∈ comap (↑) (𝓝 y)` and a fortiori `f ⁻¹' V' ∈ comap (↑) (𝓝 y)`. In addition `comap (↑) (𝓝 y) ≠ ⊥` because `A` is dense. Because we know `Tendsto f (comap (↑) (𝓝 y)) (𝓝 (φ y))` this implies `φ y ∈ closure V'` and, since `V'` is closed, we have proved `φ y ∈ V'`.

It remains to prove that φ extends `f`. This is where the continuity of `f` enters the discussion, together with the fact that `Y` is Hausdorff.

```
example [TopologicalSpace X] [TopologicalSpace Y] [T3Space Y] {A : Set X}
    (hA : ∀ x, x ∈ closure A) {f : A → Y} (f_cont : Continuous f)
    (hf : ∀ x : X, ∃ c : Y, Tendsto f (comap (↑) (𝓝 x)) (𝓝 c)) :
    ∃ φ : X → Y, Continuous φ ∧ ∀ a : A, φ a = f a := by
  sorry

#check HasBasis.tendsto_right_iff
```

In addition to separation property, the main kind of assumption you can make on a topological space to bring it closer to metric spaces is countability assumption. The main one is first countability asking that every point has a countable neighborhood basis. In particular this ensures that closure of sets can be understood using sequences.

```
example [TopologicalSpace X] [FirstCountableTopology X]
      {s : Set X} {a : X} :
    a ∈ closure s ↔ ∃ u : ℕ → X, (∀ n, u n ∈ s) ∧ Tendsto u atTop (𝓝 a) :=
  mem_closure_iff_seq_limit
```

11.3.3 Compactness

Let us now discuss how compactness is defined for topological spaces. As usual there are several ways to think about it and Mathlib goes for the filter version.

We first need to define cluster points of filters. Given a filter `F` on a topological space `X`, a point `x : X` is a cluster point of `F` if `F`, seen as a generalized set, has non-empty intersection with the generalized set of points that are close to `x`.

Then we can say that a set `s` is compact if every nonempty generalized set `F` contained in `s`, i.e. such that `F ≤ 𝒫 s`, has a cluster point in `s`.

```
variable [TopologicalSpace X]

example {F : Filter X} {x : X} : ClusterPt x F ↔ NeBot (𝓝 x ⊓ F) :=
  Iff.rfl

example {s : Set X} :
    IsCompact s ↔ ∀ (F : Filter X) [NeBot F], F ≤ 𝒫 s → ∃ a ∈ s, ClusterPt a F :=
  Iff.rfl
```

For instance if `F` is `map u atTop`, the image under `u : ℕ → X` of `atTop`, the generalized set of very large natural numbers, then the assumption `F ≤ 𝒫 s` means that `u n` belongs to `s` for `n` large enough. Saying that `x` is a cluster point of `map u atTop` says the image of very large numbers intersects the set of points that are close to `x`. In case `𝓝 x` has a countable basis, we can interpret this as saying that `u` has a subsequence converging to `x`, and we get back what compactness looks like in metric spaces.

```
example [FirstCountableTopology X] {s : Set X} {u : ℕ → X} (hs : IsCompact s)
    (hu : ∀ n, u n ∈ s) : ∃ a ∈ s, ∃ φ : ℕ → ℕ, StrictMono φ ∧ Tendsto (u ∘ φ) atTop␣
↪(𝓝 a) :=
  hs.tendsto_subseq hu
```

Cluster points behave nicely with continuous functions.

```
variable [TopologicalSpace Y]

example {x : X} {F : Filter X} {G : Filter Y} (H : ClusterPt x F) {f : X → Y}
    (hfx : ContinuousAt f x) (hf : Tendsto f F G) : ClusterPt (f x) G :=
  ClusterPt.map H hfx hf
```

As an exercise, we will prove that the image of a compact set under a continuous map is compact. In addition to what we saw already, you should use `Filter.push_pull` and `NeBot.of_map`.

```
example [TopologicalSpace Y] {f : X → Y} (hf : Continuous f) {s : Set X} (hs :␣
↪IsCompact s) :
    IsCompact (f '' s) := by
  intro F F_ne F_le
  have map_eq : map f (𝒫 s ⊓ comap f F) = 𝒫 (f '' s) ⊓ F := by sorry
  have Hne : (𝒫 s ⊓ comap f F).NeBot := by sorry
  have Hle : 𝒫 s ⊓ comap f F ≤ 𝒫 s := inf_le_left
  sorry
```

One can also express compactness in terms of open covers: `s` is compact if every family of open sets that cover `s` has a finite covering sub-family.

```
example {ι : Type*} {s : Set X} (hs : IsCompact s) (U : ι → Set X) (hUo : ∀ i, IsOpen␣
↪(U i))
    (hsU : s ⊆ ⋃ i, U i) : ∃ t : Finset ι, s ⊆ ⋃ i ∈ t, U i :=
  hs.elim_finite_subcover U hUo hsU
```

CHAPTER

TWELVE

DIFFERENTIAL CALCULUS

We now consider the formalization of notions from *analysis*, starting with differentiation in this chapter and turning integration and measure theory in the next. In Section 12.1, we stick with the setting of functions from the real numbers to the real numbers, which is familiar from any introductory calculus class. In Section 12.2, we then consider the notion of a derivative in a much broader setting.

12.1 Elementary Differential Calculus

Let `f` be a function from the reals to the reals. There is a difference between talking about the derivative of `f` at a single point and talking about the derivative function. In Mathlib, the first notion is represented as follows.

```
open Real

/-- The sin function has derivative 1 at 0. -/
example : HasDerivAt sin 1 0 := by simpa using hasDerivAt_sin 0
```

We can also express that `f` is differentiable at a point without specifying its derivative there by writing `DifferentiableAt ℝ`. We specify $\mathbb{R}$ explicitly because in a slightly more general context, when talking about functions from $\mathbb{C}$ to $\mathbb{C}$, we want to be able to distinguish between being differentiable in the real sense and being differentiable in the sense of the complex derivative.

```
example (x : ℝ) : DifferentiableAt ℝ sin x :=
  (hasDerivAt_sin x).differentiableAt
```

It would be inconvenient to have to provide a proof of differentiability every time we want to refer to a derivative. So Mathlib provides a function `deriv f : ℝ → ℝ` that is defined for any function `f : ℝ → ℝ` but is defined to take the value `0` at any point where `f` is not differentiable.

```
example {f : ℝ → ℝ} {x a : ℝ} (h : HasDerivAt f a x) : deriv f x = a :=
  h.deriv

example {f : ℝ → ℝ} {x : ℝ} (h : ¬DifferentiableAt ℝ f x) : deriv f x = 0 :=
  deriv_zero_of_not_differentiableAt h
```

Of course there are many lemmas about `deriv` that do require differentiability assumptions. For instance, you should think about a counterexample to the next lemma without the differentiability assumptions.

```
example {f g : ℝ → ℝ} {x : ℝ} (hf : DifferentiableAt ℝ f x) (hg : DifferentiableAt ℝ␣
↪g x) :
    deriv (f + g) x = deriv f x + deriv g x :=
  deriv_add hf hg
```

Interestingly, however, there are statements that can avoid differentiability assumptions by taking advantage of the fact that the value of `deriv` defaults to zero when the function is not differentiable. So making sense of the following statement requires knowing the precise definition of `deriv`.

```
example {f : ℝ → ℝ} {a : ℝ} (h : IsLocalMin f a) : deriv f a = 0 :=
  h.deriv_eq_zero
```

We can even state Rolle's theorem without any differentiability assumptions, which seems even weirder.

```
open Set

example {f : ℝ → ℝ} {a b : ℝ} (hab : a < b) (hfc : ContinuousOn f (Icc a b)) (hfI :␣
↪f a = f b) :
    ∃ c ∈ Ioo a b, deriv f c = 0 :=
  exists_deriv_eq_zero hab hfc hfI
```

Of course, this trick does not work for the general mean value theorem.

```
example (f : ℝ → ℝ) {a b : ℝ} (hab : a < b) (hf : ContinuousOn f (Icc a b))
    (hf' : DifferentiableOn ℝ f (Ioo a b)) : ∃ c ∈ Ioo a b, deriv f c = (f b - f a) /␣
↪(b - a) :=
  exists_deriv_eq_slope f hab hf hf'
```

Lean can automatically compute some simple derivatives using the `simp` tactic.

```
example : deriv (fun x : ℝ ↦ x ^ 5) 6 = 5 * 6 ^ 4 := by simp

example : deriv sin π = -1 := by simp
```

12.2 Differential Calculus in Normed Spaces

12.2.1 Normed spaces

Differentiation can be generalized beyond $\mathbb{R}$ using the notion of a *normed vector space*, which encapsulates both direction and distance. We start with the notion of a *normed group*, which is an additive commutative group equipped with a real-valued norm function satisfying the following conditions.

```
variable {E : Type*} [NormedAddCommGroup E]

example (x : E) : 0 ≤ ‖x‖ :=
  norm_nonneg x

example {x : E} : ‖x‖ = 0 ↔ x = 0 :=
  norm_eq_zero

example (x y : E) : ‖x + y‖ ≤ ‖x‖ + ‖y‖ :=
  norm_add_le x y
```

Every normed space is a metric space with distance function $d(x, y) = \|x - y\|$, and hence it is also a topological space. Lean and Mathlib know this.

```
example : MetricSpace E := by infer_instance

example {X : Type*} [TopologicalSpace X] {f : X → E} (hf : Continuous f) :
```

(continues on next page)

(continued from previous page)

```
    Continuous fun x ↦ ‖f x‖ :=
  hf.norm
```

In order to use the notion of a norm with concepts from linear algebra, we add the assumption `NormedSpace ℝ E` on top of `NormedAddGroup E`. This stipulates that `E` is a vector space over $\mathbb{R}$ and that scalar multiplication satisfies the following condition.

```
variable [NormedSpace ℝ E]

example (a : ℝ) (x : E) : ‖a • x‖ = |a| * ‖x‖ :=
  norm_smul a x
```

A complete normed space is known as a *Banach space*. Every finite-dimensional vector space is complete.

```
example [FiniteDimensional ℝ E] : CompleteSpace E := by infer_instance
```

In all the previous examples, we used the real numbers as the base field. More generally, we can make sense of calculus with a vector space over any *nontrivially normed field*. These are fields that are equipped with a real-valued norm that is multiplicative and has the property that not every element has norm zero or one (equivalently, there is an element whose norm is bigger than one).

```
example (𝕜 : Type*) [NontriviallyNormedField 𝕜] (x y : 𝕜) : ‖x * y‖ = ‖x‖ * ‖y‖ :=
  norm_mul x y

example (𝕜 : Type*) [NontriviallyNormedField 𝕜] : ∃ x : 𝕜, 1 < ‖x‖ :=
  NormedField.exists_one_lt_norm 𝕜
```

A finite-dimensional vector space over a nontrivially normed field is complete as long as the field itself is complete.

```
example (𝕜 : Type*) [NontriviallyNormedField 𝕜] (E : Type*) [NormedAddCommGroup E]
    [NormedSpace 𝕜 E] [CompleteSpace 𝕜] [FiniteDimensional 𝕜 E] : CompleteSpace E :=
  FiniteDimensional.complete 𝕜 E
```

12.2.2 Continuous linear maps

We now turn to the morphisms in the category of normed spaces, namely, continuous linear maps. In Mathlib, the type of $𝕜$-linear continuous maps between normed spaces `E` and `F` is written `E →L[𝕜] F`. They are implemented as *bundled maps*, which means that an element of this type a structure that that includes the function itself and the properties of being linear and continuous. Lean will insert a coercion so that a continuous linear map can be treated as a function.

```
variable {𝕜 : Type*} [NontriviallyNormedField 𝕜] {E : Type*} [NormedAddCommGroup E]
  [NormedSpace 𝕜 E] {F : Type*} [NormedAddCommGroup F] [NormedSpace 𝕜 F]

example : E →L[𝕜] E :=
  ContinuousLinearMap.id 𝕜 E

example (f : E →L[𝕜] F) : E → F :=
  f

example (f : E →L[𝕜] F) : Continuous f :=
  f.cont

example (f : E →L[𝕜] F) (x y : E) : f (x + y) = f x + f y :=
  f.map_add x y
```

(continues on next page)

(continued from previous page)

```
example (f : E →L[𝕜] F) (a : 𝕜) (x : E) : f (a • x) = a • f x :=
  f.map_smul a x
```

Continuous linear maps have an operator norm that is characterized by the following properties.

```
variable (f : E →L[𝕜] F)

example (x : E) : ‖f x‖ ≤ ‖f‖ * ‖x‖ :=
  f.le_opNorm x

example {M : ℝ} (hMp : 0 ≤ M) (hM : ∀ x, ‖f x‖ ≤ M * ‖x‖) : ‖f‖ ≤ M :=
  f.opNorm_le_bound hMp hM
```

There is also a notion of bundled continuous linear *isomorphism*. Their type of such isomorphisms is `E ≃L[𝕜] F`.

As a challenging exercise, you can prove the Banach-Steinhaus theorem, also known as the Uniform Boundedness Principle. The principle states that a family of continuous linear maps from a Banach space into a normed space is pointwise bounded, then the norms of these linear maps are uniformly bounded. The main ingredient is Baire's theorem `nonempty_interior_of_iUnion_of_closed`. (You proved a version of this in the topology chapter.) Minor ingredients include `continuous_linear_map.opNorm_le_of_shell`, `interior_subset` and `interior_iInter_subset` and `isClosed_le`.

```
variable {𝕜 : Type*} [NontriviallyNormedField 𝕜] {E : Type*} [NormedAddCommGroup E]
  [NormedSpace 𝕜 E] {F : Type*} [NormedAddCommGroup F] [NormedSpace 𝕜 F]

open Metric

example {ι : Type*} [CompleteSpace E] {g : ι → E →L[𝕜] F} (h : ∀ x, ∃ C, ∀ i, ‖g i x‖
↪≤ C) :
    ∃ C', ∀ i, ‖g i‖ ≤ C' := by
  -- sequence of subsets consisting of those `x : E` with norms `‖g i x‖` bounded by
↪`n`
  let e : ℕ → Set E := fun n ↦ ⋂ i : ι, { x : E | ‖g i x‖ ≤ n }
  -- each of these sets is closed
  have hc : ∀ n : ℕ, IsClosed (e n)
  sorry
  -- the union is the entire space; this is where we use `h`
  have hU : (⋃ n : ℕ, e n) = univ
  sorry
  /- apply the Baire category theorem to conclude that for some `m : ℕ`,
       `e m` contains some `x` -/
  obtain ⟨m, x, hx⟩ : ∃ m, ∃ x, x ∈ interior (e m) := sorry
  obtain ⟨ε, ε_pos, hε⟩ : ∃ ε > 0, ball x ε ⊆ interior (e m) := sorry
  obtain ⟨k, hk⟩ : ∃ k : 𝕜, 1 < ‖k‖ := sorry
  -- show all elements in the ball have norm bounded by `m` after applying any `g i`
  have real_norm_le : ∀ z ∈ ball x ε, ∀ (i : ι), ‖g i z‖ ≤ m
  sorry
  have εk_pos : 0 < ε / ‖k‖ := sorry
  refine ⟨(m + m : ℕ) / (ε / ‖k‖), fun i ↦ ContinuousLinearMap.opNorm_le_of_shell ε_
↪pos ?_ hk ?_⟩
  sorry
  sorry
```

12.2.3 Asymptotic comparisons

Defining differentiability also requires asymptotic comparisons. Mathlib has an extensive library covering the big O and little o relations, whose definitions are shown below. Opening the `asymptotics` locale allows us to use the corresponding notation. Here we will only use little o to define differentiability.

```
open Asymptotics

example {α : Type*} {E : Type*} [NormedGroup E] {F : Type*} [NormedGroup F] (c : ℝ)
    (l : Filter α) (f : α → E) (g : α → F) : IsBigOWith c l f g ↔ ∀ᶠ x in l, ‖f x‖
↪≤ c * ‖g x‖ :=
  isBigOWith_iff

example {α : Type*} {E : Type*} [NormedGroup E] {F : Type*} [NormedGroup F]
    (l : Filter α) (f : α → E) (g : α → F) : f =O[l] g ↔ ∃ C, IsBigOWith C l f g :=
  isBigO_iff_isBigOWith

example {α : Type*} {E : Type*} [NormedGroup E] {F : Type*} [NormedGroup F]
    (l : Filter α) (f : α → E) (g : α → F) : f =o[l] g ↔ ∀ C > 0, IsBigOWith C l f
↪g :=
  isLittleO_iff_forall_isBigOWith

example {α : Type*} {E : Type*} [NormedAddCommGroup E] (l : Filter α) (f g : α → E) :
    f ~[l] g ↔ (f - g) =o[l] g :=
  Iff.rfl
```

12.2.4 Differentiability

We are now ready to discuss differentiable functions between normed spaces. In analogy the elementary one-dimensional, Mathlib defines a predicate `HasFDerivAt` and a function `fderiv`. Here the letter "f" stands for *Fréchet*.

```
open Topology

variable {𝕜 : Type*} [NontriviallyNormedField 𝕜] {E : Type*} [NormedAddCommGroup E]
  [NormedSpace 𝕜 E] {F : Type*} [NormedAddCommGroup F] [NormedSpace 𝕜 F]

example (f : E → F) (f' : E →L[𝕜] F) (x₀ : E) :
    HasFDerivAt f f' x₀ ↔ (fun x ↦ f x - f x₀ - f' (x - x₀)) =o[𝓝 x₀] fun x ↦ x - x₀
↪:=
  hasFDerivAtFilter_iff_isLittleO ..

example (f : E → F) (f' : E →L[𝕜] F) (x₀ : E) (hff' : HasFDerivAt f f' x₀) : fderiv
↪𝕜 f x₀ = f' :=
  hff'.fderiv
```

We also have iterated derivatives that take values in the type of multilinear maps `E [×n]→L[𝕜] F`, and we have continuously differential functions. The type $\mathbb{N}\infty$ is $\mathbb{N}$ with an additional element ∞ that is bigger than every natural number. So $\mathcal{C}^\infty$ functions are functions `f` that satisfy `ContDiff 𝕜 ⊤ f`.

```
example (n : ℕ) (f : E → F) : E → E[×n]→L[𝕜] F :=
  iteratedFDeriv 𝕜 n f

example (n : ℕ∞) {f : E → F} :
    ContDiff 𝕜 n f ↔
      (∀ m : ℕ, (m : WithTop ℕ) ≤ n → Continuous fun x ↦ iteratedFDeriv 𝕜 m f x) ∧
```

(continues on next page)

(continued from previous page)

```
        ∀ m : ℕ, (m : WithTop ℕ) < n → Differentiable ℝ fun x ↦ iteratedFDeriv ℝ m␣
↪f x :=
  contDiff_iff_continuous_differentiable
```

The differentiability parameter in `ContDiff` can also take value ω : `WithTop` $\mathbb{N}\infty$ to denote analytic functions.

There is a stricter notion of differentiability called `HasStrictFDerivAt`, which is used in the statement of the inverse function theorem and the statement of the implicit function theorem, both of which are in Mathlib. Over $\mathbb{R}$ or $\mathbb{C}$, continuously differentiable functions are strictly differentiable.

```
example {𝕂 : Type*} [RCLike 𝕂] {E : Type*} [NormedAddCommGroup E] [NormedSpace 𝕂 E]
↪{F : Type*}
    [NormedAddCommGroup F] [NormedSpace 𝕂 F] {f : E → F} {x : E} {n : WithTop ℕ∞}
    (hf : ContDiffAt 𝕂 n f x) (hn : 1 ≤ n) : HasStrictFDerivAt f (fderiv 𝕂 f x) x :=
  hf.hasStrictFDerivAt hn
```

The local inverse theorem is stated using an operation that produces an inverse function from a function and the assumptions that the function is strictly differentiable at a point `a` and that its derivative is an isomorphism.

The first example below gets this local inverse. The next one states that it is indeed a local inverse from the left and from the right, and that it is strictly differentiable.

```
section LocalInverse
variable [CompleteSpace E] {f : E → F} {f' : E ≃L[ℝ] F} {a : E}

example (hf : HasStrictFDerivAt f (f' : E →L[ℝ] F) a) : F → E :=
  HasStrictFDerivAt.localInverse f f' a hf

example (hf : HasStrictFDerivAt f (f' : E →L[ℝ] F) a) :
    ∀ᶠ x in 𝓝 a, hf.localInverse f f' a (f x) = x :=
  hf.eventually_left_inverse

example (hf : HasStrictFDerivAt f (f' : E →L[ℝ] F) a) :
    ∀ᶠ x in 𝓝 (f a), f (hf.localInverse f f' a x) = x :=
  hf.eventually_right_inverse

example {f : E → F} {f' : E ≃L[ℝ] F} {a : E}
  (hf : HasStrictFDerivAt f (f' : E →L[ℝ] F) a) :
    HasStrictFDerivAt (HasStrictFDerivAt.localInverse f f' a hf) (f'.symm : F →L[ℝ]␣
↪E) (f a) :=
  HasStrictFDerivAt.to_localInverse hf

end LocalInverse
```

This has been only a quick tour of the differential calculus in Mathlib. The library contains many variations that we have not discussed. For example, you may want to use one-sided derivatives in the one-dimensional setting. The means to do so are found in Mathlib in a more general context; see `HasFDerivWithinAt` or the even more general `HasFDerivAtFilter`.

CHAPTER

THIRTEEN

INTEGRATION AND MEASURE THEORY

13.1 Elementary Integration

We first focus on integration of functions on finite intervals in $\mathbb{R}$. We can integrate elementary functions.

```
open MeasureTheory intervalIntegral

open Interval
-- this introduces the notation `[[a, b]]` for the segment from `min a b` to `max a b`

example (a b : ℝ) : (∫ x in a..b, x) = (b ^ 2 - a ^ 2) / 2 :=
  integral_id

example {a b : ℝ} (h : (0 : ℝ) ∉ [[a, b]]) : (∫ x in a..b, 1 / x) = Real.log (b / a)␣
↪:=
  integral_one_div h
```

The fundamental theorem of calculus relates integration and differentiation. Below we give simplified statements of the two parts of this theorem. The first part says that integration provides an inverse to differentiation and the second one specifies how to compute integrals of derivatives. (These two parts are very closely related, but their optimal versions, which are not shown here, are not equivalent.)

```
example (f : ℝ → ℝ) (hf : Continuous f) (a b : ℝ) : deriv (fun u ↦ ∫ x : ℝ in a..u,␣
↪f x) b = f b :=
  (integral_hasStrictDerivAt_right (hf.intervalIntegrable _ _) (hf.
↪stronglyMeasurableAtFilter _ _)
        hf.continuousAt).hasDerivAt.deriv

example {f : ℝ → ℝ} {a b : ℝ} {f' : ℝ → ℝ} (h : ∀ x ∈ [[a, b]], HasDerivAt f (f' x)␣
↪x)
    (h' : IntervalIntegrable f' volume a b) : (∫ y in a..b, f' y) = f b - f a :=
  integral_eq_sub_of_hasDerivAt h h'
```

Convolution is also defined in Mathlib and its basic properties are proved.

```
open Convolution

example (f : ℝ → ℝ) (g : ℝ → ℝ) : f ⋆ g = fun x ↦ ∫ t, f t * g (x - t) :=
  rfl
```

13.2 Measure Theory

The general context for integration in Mathlib is measure theory. Even the elementary integrals of the previous section are in fact Bochner integrals. Bochner integration is a generalization of Lebesgue integration where the target space can be any Banach space, not necessarily finite dimensional.

The first component in the development of measure theory is the notion of a σ-algebra of sets, which are called the *measurable* sets. The type class `MeasurableSpace` serves to equip a type with such a structure. The sets `empty` and `univ` are measurable, the complement of a measurable set is measurable, and a countable union or intersection of measurable sets is measurable. Note that these axioms are redundant; if you `#print MeasurableSpace`, you will see the ones that Mathlib uses. As the examples below show, countability assumptions can be expressed using the `Encodable` type class.

```
variable {α : Type*} [MeasurableSpace α]

example : MeasurableSet (∅ : Set α) :=
  MeasurableSet.empty

example : MeasurableSet (univ : Set α) :=
  MeasurableSet.univ

example {s : Set α} (hs : MeasurableSet s) : MeasurableSet (sᶜ) :=
  hs.compl

example : Encodable ℕ := by infer_instance

example (n : ℕ) : Encodable (Fin n) := by infer_instance

variable {ι : Type*} [Encodable ι]

example {f : ι → Set α} (h : ∀ b, MeasurableSet (f b)) : MeasurableSet (⋃ b, f b) :=
  MeasurableSet.iUnion h

example {f : ι → Set α} (h : ∀ b, MeasurableSet (f b)) : MeasurableSet (⋂ b, f b) :=
  MeasurableSet.iInter h
```

Once a type is measurable, we can measure it. On paper, a measure on a set (or type) equipped with a σ-algebra is a function from the measurable sets to the extended non-negative reals that is additive on countable disjoint unions. In Mathlib, we don't want to carry around measurability assumptions every time we write an application of the measure to a set. So we extend the measure to any set `s` as the infimum of measures of measurable sets containing `s`. Of course, many lemmas still require measurability assumptions, but not all.

```
open MeasureTheory Function
variable {μ : Measure α}

example (s : Set α) : μ s = ∏ (t : Set α) (_ : s ⊆ t) (_ : MeasurableSet t), μ t :=
  measure_eq_iInf s

example (s : ι → Set α) : μ (⋃ i, s i) ≤ Σ' i, μ (s i) :=
  measure_iUnion_le s

example {f : ℕ → Set α} (hmeas : ∀ i, MeasurableSet (f i)) (hdis : Pairwise (Disjoint␣
↪on f)) :
    μ (⋃ i, f i) = Σ' i, μ (f i) :=
  μ.m_iUnion hmeas hdis
```

Once a type has a measure associated with it, we say that a property `P` holds *almost everywhere* if the set of elements

where the property fails has measure 0. The collection of properties that hold almost everywhere form a filter, but Mathlib introduces special notation for saying that a property holds almost everywhere.

```
example {P : α → Prop} : (∀ᵐ x ∂μ, P x) ↔ ∀ᶠ x in ae μ, P x :=
  Iff.rfl
```

13.3 Integration

Now that we have measurable spaces and measures we can consider integrals. As explained above, Mathlib uses a very general notion of integration that allows any Banach space as the target. As usual, we don't want our notation to carry around assumptions, so we define integration in such a way that an integral is equal to zero if the function in question is not integrable. Most lemmas having to do with integrals have integrability assumptions.

```
section
variable {E : Type*} [NormedAddCommGroup E] [NormedSpace ℝ E] [CompleteSpace E] {f :␣
↪α → E}

example {f g : α → E} (hf : Integrable f μ) (hg : Integrable g μ) :
    ∫ a, f a + g a ∂μ = ∫ a, f a ∂μ + ∫ a, g a ∂μ :=
  integral_add hf hg
```

As an example of the complex interactions between our various conventions, let us see how to integrate constant functions. Recall that a measure μ takes values in $\mathbb{R}\geq 0\infty$, the type of extended non-negative reals. There is a function `ENNReal.toReal : ℝ≥0∞ → ℝ` which sends $\top$, the point at infinity, to zero. For any `s : Set α`, if `μ s = ⊤`, then nonzero constant functions are not integrable on `s`. In that case, their integrals are equal to zero by definition, as is `(μ s).toReal`. So in all cases we have the following lemma.

```
example {s : Set α} (c : E) : ∫ x in s, c ∂μ = (μ s).toReal • c :=
  setIntegral_const c
```

We now quickly explain how to access the most important theorems in integration theory, starting with the dominated convergence theorem. There are several versions in Mathlib, and here we only show the most basic one.

```
open Filter

example {F : ℕ → α → E} {f : α → E} (bound : α → ℝ) (hmeas : ∀ n,␣
↪AEStronglyMeasurable (F n) μ)
    (hint : Integrable bound μ) (hbound : ∀ n, ∀ᵐ a ∂μ, ‖F n a‖ ≤ bound a)
    (hlim : ∀ᵐ a ∂μ, Tendsto (fun n : ℕ ↦ F n a) atTop (𝓝 (f a))) :
    Tendsto (fun n ↦ ∫ a, F n a ∂μ) atTop (𝓝 (∫ a, f a ∂μ)) :=
  tendsto_integral_of_dominated_convergence bound hmeas hint hbound hlim
```

Then we have Fubini's theorem for integrals on product type.

```
example {α : Type*} [MeasurableSpace α] {μ : Measure α} [SigmaFinite μ] {β : Type*}
    [MeasurableSpace β] {ν : Measure β} [SigmaFinite ν] (f : α × β → E)
    (hf : Integrable f (μ.prod ν)) : ∫ z, f z ∂ μ.prod ν = ∫ x, ∫ y, f (x, y) ∂ν ∂μ :=
  integral_prod f hf
```

There is a very general version of convolution that applies to any continuous bilinear form.

```
open Convolution

variable {𝕜 : Type*} {G : Type*} {E : Type*} {E' : Type*} {F : Type*}␣
```

(continues on next page)

(continued from previous page)

```
↪[NormedAddCommGroup E]
  [NormedAddCommGroup E'] [NormedAddCommGroup F] [NontriviallyNormedField ℝ]␣
↪[NormedSpace ℝ E]
  [NormedSpace ℝ E'] [NormedSpace ℝ F] [MeasurableSpace G] [NormedSpace ℝ F]␣
↪[CompleteSpace F]
  [Sub G]

example (f : G → E) (g : G → E') (L : E →L[ℝ] E' →L[ℝ] F) (μ : Measure G) :
    f ⋆[L, μ] g = fun x ↦ ∫ t, L (f t) (g (x - t)) ∂μ :=
  rfl
```

Finally, Mathlib has a very general version of the change-of-variables formula. In the statement below, `BorelSpace E` means the σ-algebra on `E` is generated by the open sets of `E`, and `IsAddHaarMeasure` μ means that the measure μ is left-invariant, gives finite mass to compact sets, and give positive mass to open sets.

```
example {E : Type*} [NormedAddCommGroup E] [NormedSpace ℝ E] [FiniteDimensional ℝ E]
    [MeasurableSpace E] [BorelSpace E] (μ : Measure E) [μ.IsAddHaarMeasure] {F :␣
↪Type*}
    [NormedAddCommGroup F] [NormedSpace ℝ F] [CompleteSpace F] {s : Set E} {f : E → E}
    {f' : E → E →L[ℝ] E} (hs : MeasurableSet s)
    (hf : ∀ x : E, x ∈ s → HasFDerivWithinAt f (f' x) s x) (h_inj : InjOn f s) (g : E␣
↪→ F) :
    ∫ x in f '' s, g x ∂μ = ∫ x in s, |(f' x).det| • g (f x) ∂μ :=
  integral_image_eq_integral_abs_det_fderiv_smul μ hs hf h_inj g
```

CHAPTER

FOURTEEN

INDEX

INDEX