Formalising Mathematics

Release 0.1

Kevin Buzzard

May 23, 2024

CONTENTS:

A handbook for mathematicians. Written by Kevin Buzzard, for his Formalising Mathematics course. Thanks to Imperial College London for letting the course happen.

Note: This document is written for people with some mathematical experience (e.g., people in the final year of an undergraduate mathematics degree). Its aim is to show such people how to formalise mathematics in the Lean theorem prover. We make extensive use of Lean's mathematical library `mathlib`.

If you're new here, start at the *introduction*.

Here's a link to the repository for this course containing all the code.

CHAPTER

ONE

OVERVIEW OF THESE NOTES

Part A is somehow an abortive attempt to explain more information about various sections in the course Lean repository, although for most of the sections I just write comments directly in the Lean code. It can thus be regarded as a failed experiment, although I leave the information here on the basis that it might still be useful.

Part B is a collection of notes on various more "computer-sciency" topics which I wrote for the course. They are written with mathematicians in mind rather than computer scientists.

Part C is a list of many basic tactics including documentation and examples. I am not totally happy with the state of tactic documentation in Lean 4, so hopefully this is of some use to people trying to figure out how to use tactics in Lean 4.

Here is a pdf of this document.

1.1 Introduction

These are the notes for a course which is delivered to final year undergraduates and MSc students in the mathematics department at Imperial College London. I hope that they will also be useful to other mathematicians.

1.1.1 Aims and objectives

The aim of the course is to teach people how to *formalise* undergraduate and MSc level mathematics in a *computer proof assistant*. A computer proof assistant is a computer program which knows the axioms of mathematics and is capable of understanding and checking mathematical proofs. To formalise mathematics means to type it into a computer proof assistant.

Another way of looking at it is that a computer proof assistant turns the statement of a theorem of mathematics into a level of a puzzle game, and proving the theorem corresponds to solving the level. Hence another way of thinking about the aims and objectives of this course are that I am teaching you how to set and how to solve levels of a computer puzzle game.

The methods used in this course are extremely "hands-on". You will be learning by doing. Every topic will be introduced via examples and you will be expected to learn it by solving problems.

1.1.2 Why formalise?

My personal belief is the following. Software of this nature will become more normalised in mathematics departments, and libraries of formalised theorems will begin to grow quite large. Computer theorem provers will ultimately give rise to tools which will help humans to learn and to do mathematics, from undergraduate level to the frontiers of modern research. Complex tactics, perhaps backed by AI, will in the future start to help human researchers. PhD students will be able to search these libraries to get hints on how to proceed, or find references for claims which they hope are true. Other applications will appear as mathematicians begin to understand the potential of formalised libraries of mathematics and of computer programs which can understand them.

1.1.3 The Lean Theorem Prover

There are many computer proof assistants out there; the one we shall be using is called Lean. We will be using not just Lean, but also Lean's mathematics library, `mathlib`, which contains a substantial number of proofs of mathematical theorems already, as well as many high-powered *tactics*, tools which make theorem proving easier. There are many other theorem provers out there; Isabelle/HOL and Coq are two other provers which have also been used to formalise a substantial amount of mathematics. The reason we are using Lean is simply that it is the prover which I myself am most familiar with; I do not see any obstruction in theory to porting this entire course over to another theorem prover.

1.1.4 Prerequisites

Experience has shown that trying to teach people new mathematics at the same time as teaching them Lean is asking too much from most people. I will hence be assuming that the reader/player is comfortable with the material covered in the first and second year of a traditional mathematics degree. Later on, when the reader is more familiar with Lean I will introduce more mathematically challenging material.

This course is focussed on mathematics, or to be more precise, classical mathematics. We will not be concentrating on the non-mathematical side of the story. Examples of topics we will say very little about are: type theories, functional programming, the lambda calculus, and constructivism.

1.1.5 How to read this document

This book comes in three parts: Part A, Part B and Part C. Part A is the mathematical part of the book. In each of the various sections of Part A we will be talking about a mathematical idea and how to implement it and work with it in Lean. We will be learning by doing: you will be writing the code and proving the theorems yourself. To do this you need to download and install the course repository .

Part B is the non-mathematical background which you will need in order to make sense of what is going on. It covers basic material of a more "computer-science" nature. I will flag in the exercises the time when it might be helpful to read sections from this part.

Part C is a glossary of many common tactics. The problem sheets in the course repository will flag which tactics you need to learn about and when.

1.2 Installing Lean

Here's an example of a simple logic proof in Lean. It's a proof that that if `P` and `Q` are propositions (that is, true/false statements), and if `P` is true and `P ⇒ Q` is true, then `Q` is true.

```
example (P Q : Prop) (h1 : P) (h2 : P → Q) : Q := by
  apply h2 at h1
  exact h1
```

The first line of the code states the theorem. Hypothesis `h1` is that `P` is true, and hypothesis `h2` is that `P` implies `Q` (note that Lean uses a regular arrow for implication rather than the more common ⇒ sign). The conclusion, after the colon, is that `Q` is true. The proof is after the `by`, which puts Lean into "tactic mode" (`apply` and `exact` are tactics), and it's clear that it somehow uses both hypothesis `h1` and hypothesis `h2`. But just reading the proof, it's hard to see exactly what is going on. We can't learn Lean this way.

The whole point of using a theorem prover is that it makes proofs like this *interactive*. So, before we get going, we need to get this and other proofs running on your computer somehow. There are several ways to do it.

1.2.1 The best way: install Lean on your computer

If you are doing this course in 2024 at Imperial, and your computer is powerful enough to run Lean reasonably quickly, and you have about 4.5 gigabytes of space available to install Lean and the course repository, then I recommend this method.

You will first need to install Lean, and then you'll need to install the course repository.

Instructions on how to get Lean installed on your computer are here (right click and open in new tab if you don't want to lose your place).

Once you have Lean up and running, you can install the repository `ImperialCollegeLondon/formalising-mathematics-2024` associated with this course. Fire up VS Code, click on the upside-down *A* in the bar at the top, hover on "Open Project", click on "Project: Download project", and then in the text box type `https://github.com/ImperialCollegeLondon/formalising-mathematics-2024`. Navigate to the directory where you want to put the course repository, type in a name (e.g. `formalising-mathematics`), click on "Create project folder", and then for a minute or two for everything to download.

You can now use VS Code's "open folder" functionality to open the course repository and it should all work fine.

1.2.2 Lean in a web browser

If your computer is too slow to run Lean satisfactorily, or you don't have enough space on your computer, you can use Lean via a web browser. Note: I am not in any position to guarantee that with either of the approaches below, any code you write will survive between browser sessions. I recommend that if you write your course projects using a browser approach then you *save all your work locally*. You will probably need to register an account of some form (for example a GitHub account) with either of the two approaches below.

1.2.3 Browser approach 1: via gitpod.

Right click here <https://gitpod.io/#/https://github.com/ImperialCollegeLondon/formalising-mathematics-2024/>`_ and "open link in new tab" to access the repository using Gitpod. At some point it will give you some options and ask you to continue; accepting the default options is fine. I strongly recommend that you do not do *anything* until all the downloading has finished and the output in the terminal window has completely stopped; this may take several minutes. When it's done, you should see a VS Code session in your browser, and the output should end with something like `Decompressing 4053 file(s)` and `unpacked in 21359 ms` as the last couple of lines. Open the file *FormalisingMathematics2024/Section01logic/Sheet1.lean* to check it's working. Wait to check that the orange bars disappear and the infoview responds as you click between the different examples in the file.

Note: I believe there is some time limit for the amount that you can use gitpod for free. You might find that you can extend this limit by registering as a student on github.com

1.2.4 Browser approach 2: via Codespaces

Right click here <https://github.com/ImperialCollegeLondon/formalising-mathematics-2024>`_ and "open link in new tab" to go to the github page where the course repository is stored. I think that you'll need to have an account on github for this to work? Click on the green "Code" button and then click on "Codespaces". Then click on the green "Create codespace on main". Again, wait for several minutes until everything is completely finished. Open the file *FormalisingMathematics2024/Section01logic/Sheet1.lean* to check it's working. Wait to check that the orange bars disappear and the infoview responds as you click between the different examples in the file.

1.3 Part A: the mathematics

The material in this course is divided into sections, each of which corresponds to a mathematical topic. For each section there is some Lean code in the course's github repository. For some of the earlier sections (which should probably be done in order) there are some hints and notes here in the course notes.

Note that, in contrast, the material in parts B and C of these notes can be read in any order.

1.3.1 Section 1 : Logic

By "logic" here we mean the study of propositions. A proposition is a true/false statement. For example `2+2=4`, `2+2=5`, and the statement of the Riemann Hypothesis are all propositions.

Basic mathematics with propositions involves learning about how functions like $\to$, $\neg$, $\wedge$, $\leftrightarrow$ and $\vee$ interact with propositions such as `True` (the true-by-definition proposition), `False` (the false-by-definition proposition), and also with "variable" propositions with names like P and Q.

In Lean we can prove mathematical theorems using *tactics* such as `intro` and `apply`. The purpose of this section is to teach you ten or so basic tactics which can be used to solve logic problems.

Fire up a Lean session, open up the `formalising-mathematics-2024` Lean repository, find `section01logic` within the `FormalisingMathematics2024` directory of the repository, and try doing the problem sheets! If you need help getting started, you can read the below.

Lean's notation for logic.

In Lean, `P : Prop` means "`P` is a proposition", and `P → Q` is the proposition that "`P` implies `Q`". Note that Lean uses a single arrow → rather than the double arrow ⇒.

The notation `h : P` means any of the following equivalent things:

- `h` is a proof of `P`;
- `h` is the assumption that `P` is true;
- `P` is true, and this fact is called `h`.

Here `h` is just a variable name. We will often call proofs of `P` things like `hP` but you can call them pretty much anything.

WARNING: do not confuse `P : Prop` with `hP : P`. The former means that `P` is a true-false statement; the latter means that it is a true statement (and `hP` is its proof).

Lean's tactic state.

Lean's "tactic state", or "local context", is what you see on the right hand side of the screen when you have Lean up and running. In the middle of a proof it might look something like this:

```
P Q : Prop
hP : P
hPQ : P → Q
⊢ Q
```

The proposition after the "sideways T" at the bottom (it's called a "turnstile" apparently) is the thing you are supposed to be *proving* – this is the *goal* of the level of the game. The stuff above the turnstile is the stuff you are *assuming*. In the example above, P and Q are propositions, and we are assuming that P is true and that P implies Q, and we are supposed to be proving that Q is true. If you succeed in proving the goal, Lean will display a "no goals" message and, assuming you didn't use *sorry* at any point (which is cheating), you've solved the level.

How then do we manipulate the tactic state? We do this using tactics, which you type in on the left hand side of the screen, after the keyword `by` which puts Lean into "tactic mode". Take a look at some of the tactic documentation in Part C of this book to learn more about tactics. If you're just getting started, then try reading about *intro*, *apply* and *exact*. Those are the tactics you'll need to solve the levels in the first problem sheet of the course, logic sheet 1 in section 1.

1.3.2 Section 2 : An introduction to the real numbers

The real real numbers

Lean's maths library `mathlib` has the real numbers. This isn't surprising, a lot of programming languages like Python etc have the real numbers. But actually there is a difference between Python's real numbers and Lean's real numbers. Python's real numbers are actually what a computer scientist would call `Float`s, i.e. floating point numbers. Are floating point numbers different to real numbers? Yes, definitely! There are only finitely many floating point numbers, for example. Floats are "real numbers stored up to a finite precision". Addition on floats is not associative; if `N` is a really large floating point number and ε is a really small floating point number, then $N+\varepsilon=N$ because of rounding errors, so $(-N)+(N+\varepsilon)=0$ but $((-N)+N)+\varepsilon=\varepsilon$.

Lean's real numbers are the *real* real numbers. Lean's real numbers are defined under the hood as equivalence classes of Cauchy sequences (although you will never need to know this). You can prove that Lean's real numbers are uncountable. Remember that in Lean, we use types not sets. So the real numbers are a *type*. They have an official name of `Real`, but we never use this; we always use the notation, which is ℝ. Type this in VS Code with `\R`.

Lean's maths library `mathlib` doesn't just have the real numbers – it also has what is known to computer scientists as an "Application Programming Interface" for the real numbers, otherwise known as an "API" or "interface". This sounds extremely intimidating, until you discover what this actually means in practice: it means that Lean knows a whole bunch of theorems about the real numbers (for example, a nonempty bounded set of reals has a least upper bound), and Lean also has a whole bunch of definitions of standard functions on the real numbers like the cosine function `Real.cos : ℝ → ℝ` and the square root function `Real.sqrt : ℝ → ℝ` and so on, and it knows a bunch of theorems about these functions too.

Garbage in, garbage out

Did you spot what looked like a mistake in what I just said? I just claimed that the `mathlib` function `Real.sqrt` took in a real number and outputted a real number. So what happens if you feed in `-1` or some other negative number? Lean doesn't give you an error! It just spits out some arbitrary answer – for all I know, `Real.sqrt (-1) = 37`. I have no idea what `Real.sqrt (-1)` actually is, and it doesn't even matter, because I am a mathematician, so I only run `Real.sqrt` on non-negative real numbers. If I want to take the square root of a negative number, I use `Complex.sqrt`. Perhaps you are worried that such a cavalier attitude towards square roots of negative numbers leads to contradictions in the system – but it does not. The simplest way to explain why is the following. Let me define a function `f` on the real numbers, by `f(x) = sqrt(x)` for `x ≥ 0`, and `f(x) = 37` if `x<0`. Does *defining such a function* lead to a contradiction in mathematics? Of course it does not! That's what `Real.sqrt` looks like (although they might have chosen `0` or `sqrt(-x)` for the output when `x` was negative – who knows, and who cares). The trick is that it is in the *theorems* about `Real.sqrt` where the hypotheses of nonnegativity appear. For example, it is probably not true that `Real.sqrt (a * b) = Real.sqrt (a) * Real.sqrt (b)` in general, because `a` or `b` might be negative. However it *is* true that `Real.sqrt (a * b) = Real.sqrt (a) * Real.sqrt (b)` if `a ≥ 0` and `b ≥ 0`, and this is the theorem in the library. So you *do* have to check that you're not taking the square roots of negative numbers – just not where you might expect.

Lean is not a computer

The last thing I would like to say about the real numbers in this introduction is that Lean 4 is *not designed to compute* right now. This might all change one day but right now you cannot try and "evaluate" `Real.sqrt 2` and expect the answer to be equal to `1.41421356...`. That `...` has no meaning in Lean, because real numbers have infinite precision. You could prove that `1.41421356 < Real.sqrt 2 < 1.41421357` by, for example, squaring up and multiplying out. I think it would be an interesting and possibly tricky challenge to prove something like `0.54 < Real.cos 1 ∧ Real.cos 1 < 0.55` (Lean's cosines are of course in radians; one might want to try and prove this using the power series expansion of cosine but it might well be painful). Someone is currently working on a tactic to do this, but for the 2024 version of this course this should be regarded as "not ready yet".

Working with reals

How do we state that `2+2=4`? We can do it like this:

```
example : (2 : ℝ) + 2 = 4 := by
  sorry
```

What's going on here? If I hadn't put that `ℝ` in there, Lean would have assumed that I meant the natural number `2` – Lean's "default" `2`. The naturals, integers, rationals, reals and complexes (and p-adic numbers for all primes p, if you know what they are) are all different types, so they all have different `2` s. Writing `(2 : ℝ)` tells Lean exactly which `2` we mean.

So how come we didn't have to write `(2 : ℝ) + (2 : ℝ) = (4 : ℝ)`? The reason for this is that addition in Lean takes two terms of a type `X` and outputs a term of the same type `X`, so once Lean knows that the first term has type `ℝ` it figures out that all the other terms must have type `ℝ` as well for things to make sense.

How do we *prove* that `2+2=4`? The answer is the *norm_num* tactic, which "normalises numerical expressions" – in other words it proves equalities and inequalities *as long as only numbers, and no variables, are involved*.

To solve the `sorry` s in section 2 you'll need to know about the *norm_num* and *ring* tactics, as well as the `specialize` tactic. Read about them in the tactic documentation in Part C, and, if you like, in the unofficial mathlib community tactic documentation .

1.3.3 Section 3 : functions

This short section is an introduction to "abstract" functions in Lean. We've seen functions from the naturals to the reals, but here we'll be talking about functions between types `X` and `Y`.

Lean is a functional programming language, and functional programmers realised a long time ago that brackets which mathematicians put into their work are often not necessary. If `f : X → Y` and `x : X` then write `f x` in Lean instead of `f(x)` (which doesn't actually work in Lean at all). Brackets can be used – but you have to leave a space (so `f (x)`) and they are optional anyway.

Be warned though – functions in Lean are *greedy*: they will eat the next thing they see. For example `f n + 1` means `(f n) + 1` and not `f (n + 1)`. If you want `f (n+1)` then you need to use the brackets. Similarly, composite of functions `f : X → Y` and `g : Y → Z` evaluated at `x : X` needs to be written `g (f x)` otherwise `g` would eat `f` and then choke because `f` doesn't have the right type.

The problem sheets

For the first sheet you should be able to get away with what you already know. In the second one I explain how to make types with finitely many terms, and how to make functions from these types. In the last sheet I show how to use these to make counterexamples to various assertions about injective and surjective functions.

1.3.4 Section 4 : Sets

Lean uses type theory, not set theory. Lean has sets, but they're always subsets of a type. For example, the real numbers $\mathbb{R}$ are a type in Lean, so you can make sets of real numbers. The type of sets of real numbers is called `Set ℝ` in Lean. More generally if `X` is any type, `Set X` is the type of subsets of `X`.

Remember that in Lean's type theory, every object lives at one of three "levels". There are the two universes `Type` and `Prop` at the top, and then there are the types and the theorem statements one level below them, and then there are the terms and the theorem proofs at the bottom. The type of all real numbers $\mathbb{R}$ is a type, so $\mathbb{R}$ lives at the middle level, and real numbers like `7` are terms; we write `7 : ℝ` to indicate that 7 is a real number.

Here's how sets fit into the picture, and it might not be how you think. The type `Set ℝ` is the type of sets of real numbers, so `Set ℝ : Type`. A term of this type is hence a term, so at the bottom level. Let `S : Set ℝ` be a set of real numbers (for example S could be $\{1, \sqrt{2}, \pi\}$ or the set of prime numbers or whatever). Then `S` is a term, not a type, so `3 : S` *doesn't make sense* (because `t : T` means that `t` is a term and `T` is a type). So how do we talk about elements of sets? We do it the same way that mathematicians do. If `x : ℝ` is a real number, then we write `x ∈ S` to mean that `x` is an element of the set `S`. Note that both `x` and `S` in `x ∈ S` are terms. Note also that `x ∈ S : Prop`, i.e. `x ∈ S` is a true-false statement.

The `ext` tactic is a useful one to know when dealing with equalities of sets. Extensionality is a mathematical principle which states that two objects are equal if they're made from the same things. For example, sets are extensional objects in mathematics, which means that two sets are equal if and only if they have the same elements. If `S : Set ℝ` and `T : Set ℝ` are sets, and your goal is `⊢ S = T` then the tactic `ext x` will introduce a new arbitrary real number `x : ℝ` into the tactic state, and will turn the goal into `x ∈ S ↔ x ∈ T`.

1.3.5 Section 5 : An introduction to group theory

Overview of the sheets

Lean has a bunch of group theory already (Sylow's theorems, nilpotent and solvable groups and other "final year undergraduate level mathematics"). Should we use what is already there, or rebuild from scratch?

In sheet 1 of section 5 we go through a brief tour of how to use Lean's groups. In sheet 2 we make new "class"es `WeakGroup` and `BadGroup` (so now you can guess how to make groups), and show that a `WeakGroup` is the same as a `Group` but that there are `BadGroup` s which aren't groups. In sheet 3 we give a brief overview of subgroups and group homomorphisms.

Structures and classes

Groups, subgroups, and group homomorphisms in Lean are all encoded via inductive types. These inductive types are of a particular boring nature: they only have one constructor. Thus in contrast to `P ∨ Q`, which can be proved in two ways (you either prove `P` or you prove `Q`), there is only one way to make a subgroup of a group: you have to give a subset of the group, plus proofs that it satisfies the axioms of a subgroup. You can read about the details of *structures* and *classes* by following those links; I will not say any more about them here.

What you need to know to do the sheets

Sheet 1: There will be a lot of `rw` involved, because we are constantly using the group axioms. Sheet 2 is perhaps mathematically tricky. Sheet 3 is `refine` and `ext` and `apply` practice.

1.3.6 Section 6 : quotients

A few years ago, I started to understand much better the two completely different ways in which mathematicians *construct new objects*. One way to define an object is to say *what it is*. The other way is to say *what it does*. Let's take a look at some examples.

The natural numbers

Here is a "concrete" definition of the natural numbers. We can define `0` to be the empty set. We define `1` to be `{0}`, we define `2` to be `{0,1}` and so on. An axiom of mathematics tells us that we can put all these things together to make an infinite set `{0,1,2,3,...}`. One can define `n+1` to be the union of `n` and `{n}`, and then prove the principle of mathematical induction for this *concrete model* of the natural numbers.

Another definition of the natural numbers, just as old, goes back to Peano. He defines them via postulates: 0 is a natural, the successor of a natural is a natural, and that's it. By "that's it" we mean that "this is the only way to define naturals"; more precisely we mean that if you want to prove a theorem about naturals, then it suffices to prove it for `0`, and also to show that if you've proved it for `n` then you can prove it for `n+1`. In other words, we are *stating without proof* that the principle of induction holds.

The first definition of a natural is a "what it is" definition; the second is a "what it does" definition. To a mathematician it *simply does not matter* which approach we use. All that a mathematician cares about is that there is an object called $\mathbb{N}$ in which we can do arithmetic and which satisfies the principle of mathematical induction (and, strictly speaking, also the principle of mathematical recursion). Whether induction is an axiom or a theorem is of no interest to us, just as it was of no interest to people like Gauss and Euler. Furthermore, anyone who believes that "secretly they are using the concrete model all the time" will have a big shock when it comes to formalising; there are copies of the natural numbers embedded in the integers, the rationals, the reals, the complexes, the quaternions, the octonions, and the p-adic numbers for all prime numbers p, and anyone who has seen the construction of a sensible concrete model for any of these mathematical ideas

will know that, for example, zero is not actually the empty set in any of them; and yet it turns out that we still use induction and recursion on these copies of the naturals (for example, when adding them).

Products

Let `X` and `Y` be sets or types or whatever you want to call a "collection of abstract elements". A crucial construction in mathematics is the *product* of `X` and `Y`, typically denoted `X × Y`. We are often told what looks like a concrete model for this object; an element of `X × Y` consists of an ordered pair of elements `(x,y)`, with `x` an element of `X` and `y` an element of `Y`. But what is an ordered pair? Well, there are two approaches: we can either say what it is, or what it does.

Wikipedia explains three ways to set up the theory of ordered pairs in set theory (due to Wiener, Hausdorff and Kuratowski), and of course there are many more. I will not explain any of them here because clearly we do not care about "what it is" here, we only care about "what it does", which is that two ordered pairs (x_1, y_1) and (x_2, y_2) are equal if and only if $x_1 = y_1$ and $x_2 = y_2$. What it is, is an uninteresting implementation issue rather than a mathematical one.

On the other hand, when it comes to `X × Y`, what it *does* is the following: there are projection maps to `X` and `Y`, and for any `Z` we know that to give a map from `Z` to `X × Y` is to give a map from `Z` to `X` and a map from `Z` to `Y`, with the dictionary given via composition with the projections. However, imagine developing the theory of linear maps on $\mathbb{R}^2$ knowing only this; it's really helpful to know what it is, namely ordered pairs of reals (x, y). We conclude that sometimes mathematicians do use what it is, but sometimes they only use what it does. We also see that sometimes there is more than one choice for what it is, and in a situation where we only care about what it does, we really don't care what it actually is.

There are many other situations in more advanced mathematics where we don't care what it is, but only what it does: examples are tensor products of vector spaces (and more generally of modules), pullbacks of sheaves on topological spaces. Cohomology is a situation where we do sometimes care about what it is, however our definition of what it is depends on what we're doing; if doing abstract homological algebra our model might involve injective objects, and if doing a calculation it might involve cycles and boundaries (possibly homogeneous, possibly inhomogeneous). Mathematicians are extremely good at switching between models at will, or using no model at all, depending on the situation.

I'll now explain how quotients work in Lean, using a "what it does" approach.

Quotients

Here's the set-up. We have `X` and an equivalence relation $\approx$ on `X`. Equivalence relations are some kind of generalisation of equality, and we want a new object `Q` where $\approx$ actually becomes equality; in other words we want a surjection `⟦.⟧ : X → Q` and the theorem that $x_1 \approx x_2 \leftrightarrow ⟦x_1⟧ = ⟦x_2⟧$. In Cambridge I learnt what `Q` *is*, namely the set of equivalence classes of `X`, and the function `⟦.⟧` is called "equivalence class of". Thirty years later Patrick Massot pointed out to me that actually I never once used "what it is", I only ever used "what it does", and it did not take me long to realise that he was exactly right. Just like the other examples above, this construction of the quotient `Q` as a set of sets is not "the answer" but in fact just *a model*. The reason I am bringing this up is that *Lean does not use this model*. In fact Lean uses a "secret" model for `Q` – a type called `quotient s` where `s` is a term which bundles together the binary relation $\approx$ and the proof that it's an equivalence relation. We cannot "inspect" the terms of this type and ask if they are subsets of `X`; it is an opaque construction. However we know what `quotient s` *does* because Lean provides the API to do it. In particular, the fact that `⟦.⟧` is a surjection and that $x_1 \approx x_2 \leftrightarrow ⟦x_1⟧ = ⟦x_2⟧$ are available to us as theorems in Lean's library.

In theory, we could prove everything about quotients using these two facts, however there is a construction in mathematics which is so common that Lean singles it out and gives us extra API for it. We finish by talking about it. In Lean it is called `quotient.lift` which is a great shame because mathematically it is a descent, not a lift. In mathematics it goes by the far more unwieldy name of "the way to show that a function defined from a quotient using a certain method is well-defined".

What does it mean to be "well-defined"?

One thing which I've seen students struggle with over my decades of teaching is the concept of a function out of a quotient being "well-defined". We want to define a function `f` from `Q` to some other type/set `Z` and the strategy is as follows. Take an element `q` of `Q` and pretend it's an equivalence class. Now choose a random element of the equivalence class; this is now an element `x` of `X`. Using `x`, construct an element of `Z`. Now claim that this element of `Z` is "well-defined", and decree that `f(q)` is this element. What does this well-definedness boil down to? It's the idea that if we had chosen a different element `y` of the equivalence class, and used that instead of `x` to construct the element of `Z`, we would have got the same element.

I find this very confusing to write, so no wonder students find it hard to comprehend. It's much easier to explain it in the following way. Our "recipe" to make an element of `Z` from `x` is just a function `F : X → Z`. The check that if we'd chosen a different element `y` in the equivalence class of `q` instead of `x` we'd get the same answer, is simply the assertion that `F(y)=F(x)`. Finally, the fact that `x` and `y` are in the same equivalence class is just the assertion that `x ≈ y`.

The upshot of all this is that the mathematician's "principle of defining a function and then checking it's well-defined" boils down to finding a term of the following type:

```
∀ (F : X → Z) (h : ∀ (x y : X), x ≈ y → F x = F y), (Q → Z)
```

This term is called `quotient.lift` in Lean. You feed it `F : X → Z` and a proof `h` that `F` is constant on equivalence classes, and it descends `F` to a function `Q → Z` called `quotient.lift F h`.

The only question left is how to prove that `(quotient.lift F h) (⟦x⟧) = F x`, that is, that `quotient.lift F h` really is a descent of `F` down to `Q`. And in Lean, not only is this true, but the proof is `refl`.

1.3.7 Section 13 : Modules

When you learn vector spaces at Imperial College London, you start off with the theory of real vector spaces, on the basis that this is "more concrete" or something. You do examples in two and three dimensions and talk about vectors and matrices. Multiplication of matrices is explained via some weird formula.

Later on, you begin to abstract away from the concrete model $\mathbb{R}^3$ all of whose elements have a unique "formula" like $(1, 2, 3)$, and move on to the general space with a basis $\mathbb{R}^n$, and finally to the general real vector space V, an abstraction of the concrete axioms satisfied by $mathbbR^2$ and $mathbbR^3$. You can still add two vectors: $v_1 + v_2 \in V$, and multiply a vector by a scalar $r \cdot v \in V$ if $r \in R$ and $v \in V$, but there is no longer a "concrete model" – an lement of V has now become an opaque abstraction; the elements no longer have "formulas" like $(\sin(\theta), \cos(\theta), 37)$ and if you want them back (for example if you want to do an explicit computation like they do in applied maths) then you need to pick a basis.

1.4 Part B: Lean tips

The purpose of this part is to explain some non-mathematical details about Lean. The sections are independent of each other, and the primary use of this part is as a reference.

1.4.1 Types and terms

Sets and their elements

In mathematics you have seen many examples of sets and their elements. For example the real numbers $\mathbb{R}$ is a set, and it has elements such as `37` and `12345`. It is difficult to give a formal definition of a set. Typically a student thinks of a set as a "collection of things", and the elements of the set are the things.

Lean uses something called type theory as a foundation of mathematics rather than set theory. We will not be launching into a deep study of type theory in this course; the idea of this section is to give you a working knowledge of the key differences between type theory and set theory.

In Lean the *type* plays the role of the "collection of things", and the things in the type are called *terms*. For example, in Lean the real numbers $\mathbb{R}$ are a type, not a set, and specific real numbers like `37` and `12345` are called terms of this type.

The notation used is also different to what you have usually seen. In set theory, we write `37 ∈ ℝ` to mean that `37` is a real number. More formally we might say "`37` is an element of the set of real numbers". In type theory the notation is different. In type theory we express the idea that `37` is a real number by writing `37 : ℝ`, and more formally we would say "`37` is a term of the type of real numbers". Basically the colon `:` in type theory plays the role of the "is an element of" symbol $\in$ in set theory.

The universe of all types

Some of you might know that whilst it's unproblematic to talk about the set of real numbers in set theory, it *is* problematic to talk about the set of all sets. Russell's Paradox is the observation that if `X` is the set of all sets which are not elements of themselves, then `X ∈ X` if and only if `X ∉ X`, a contradiction. For similar reasons one cannot expect there to be a type of all types – this type of "self-referentiality" can lead to logical problems. In Lean, there is a *universe* of all types, and this universe is called `Type`. The statement that the real numbers are a type can be expressed as `ℝ : Type`.

Here is another example. If `G` is a group, and `g` is an element of this group, then in set theory one might say that `G` is a set and `g ∈ G` is an element of the set. In type theory one says instead that `G : Type` and that `g : G`.

The mental model which you should have in your mind is that in Lean, every object exists at one of three "levels". There are universes, such as `Type`, there are types such as `ℝ` or `G` and there are terms such as `37` and `g`. Every mathematical object you know fits neatly into this heirarchy. For example rings, fields and topological spaces are all types in Lean, and their elements are terms.

Function types

Say `X` and `Y` are types. The standard notation which you have seen for a function from `X` to `Y` is `f : X → Y`. This is also the notation used in Lean. A mathematician might write `Hom(X,Y)` for the set of all functions from `X` to `Y`. In Lean this set is of course a type, and the notation for this type is `X → Y`. So `f : X → Y` says that `f` is a term of type `X → Y`, i.e., the type theory version of the idea that `f` is an element of the set `Hom(X,Y)`, or equivalently that `f` is a function from `X` to `Y`.

The universe *Prop*

Mathematicians define objects such as the real numbers and groups, but they also prove theorems about these objects. What is a theorem? It has two parts, a *statement* and a *proof*. Lean needs to be able to manipulate theorem statements and proofs as well as being able to manipulate objects such as the real numbers and groups. How do theorem statements and theorem proofs fit into the picture?

The answer to this question is beautifully simple. Lean regards a theorem statement as a type, not living in the `Type` universe, but in another universe called `Prop` – the universe of true-false statements. A true-false statement, otherwise known as a *proposition* in this course, is a statement such as `2+2=4` or `2+2=5` or the Riemann hypothesis. Note in particular that we are reclaiming the word "proposition" from its traditional usage in other mathematics courses. You might have seen the word "proposition" being used to mean the same thing as "lemma" or "theorem" or "corollary" or "sublemma" or… . We don't need so many words to express the same idea, so in this course we will use the word "proposition" to mean the same thing as the logicians and the computer scientists: propositions, unlike theorems, can be false! A proposition is the same thing as a true-false statement. The notation `P : Prop` means that `P` is a proposition. For example `2 + 2 = 4 : Prop` and `2 + 2 = 5 : Prop`.

The idea that a proposition can be thought of as a type means in particular that a proposition has somehow got "elements". This is not the way that true-false statements are usually thought of by mathematicians, but it is a key idea in Lean's type theory. The "elements" (or, to use Lean's language, the terms) of a proposition are its proofs! Every proposition in Lean has *at most one term*. The true propositions have one term, and the false propositions have no terms. To give a concrete example, we have `2 + 2 = 4 : Prop`, because 2+2=4 is a true-false statement. we will learn in this course how to make a term `h : 2 + 2 = 4`; this term `h` should be thought of as a proof that 2+2=4. You could read it as the hypothesis that `2 + 2 = 4` or however you like, but under the hood what is happening is that `h` is a term of the type `2 + 2 = 4`.

We also have the proposition `2 + 2 = 5 : Prop`. It is however impossible to make a term whose type is `2 + 2 = 5`, because 2+2=5 is a false proposition. If you like, you can think of `2 + 2 = 4` as a set with one element, and `2 + 2 = 5` as a set with no elements. This is initially a rather bizarre way of thinking about true-false statements, however you will soon get used to it.

The reason it is important to start thinking of elements of sets and proofs of propositions as "the same sort of thing", is that when formalising mathematics one frequently runs into things like the type of non-negative real numbers. To give a term of this type is to give a pair `(x,h)` consisting of a real number `x` and a proof `h` that `x ≥ 0`, or, to put it in Lean's language, a term `x : ℝ` and a term `h : x ≥ 0`. When doing mathematics like this in Lean, one just gets used to the fact that some variables are representing elements of sets and others are representing proofs of propositions.

`P ⇒ Q` is `P → Q`

Here's an interesting analogy.

In the usual set-theoretic language which mathematicians use, we might say the following: If `X` and `Y` are sets, then we can consider the set `Hom(X,Y)` of functions from `X` to `Y`, and an element `f ∈ Hom(X,Y)` is a function from `X` to `Y`.

In Lean's type theory we say it like this: if `X` and `Y` are types in the `Type` universe, then we can consider the type `X → Y` of functions from `X` to `Y`, and a term `f : X → Y` of this type is a function from `X` to `Y`.

In usual mathematical logic, we might say the following: If `P` and `Q` are true-false statements, then `P ⇒ Q` is also a true-false statement (for example if `P` is true and `Q` is false, then `P ⇒ Q` is false). If we have a hypothesis `h` that says that `P ⇒ Q` is true, we might write `h : P ⇒ Q`.

In Lean's type theory we say it like this. If `P` and `Q` are types in the `Prop` universe, i.e., propositions, then we can consider the type `P → Q` of functions from proofs of `P` to proofs of `Q`. If we have such a function `h`, which takes as input a proof of `P` and spits out a proof of `Q`, then `h` can be thought of as a proof that `P ⇒ Q`. In Lean the function type `P → Q` lives in the `Prop` universe – it's also a true-false statement.

What Lean's type theory is suggesting here is that an interesting model for a true/false statement is a set with at most one element. If the set has an element, it correponds to a true statement, and if it has no elements then it corresponds to a false statement.

As an exercise, imagine that `P` and `Q` are sets with either 0 or 1 element, and try and work out in each of the four cases the size of the set `Hom(P,Q)`, which in Lean we would write as `P → Q`. The answer you should get is that the size of `Hom(P,Q)` should be either 0 or 1, and it is 0 if `P ⇒ Q` is false, and 1 if `P ⇒ Q` is true.

Summary

Types and their terms unify two mathematical concepts: sets and their elements, and theorem statements and their proofs. The universe `Type` is where the sets/elements types live, and the universe `Prop` is where the theorems/proofs types live. An implication `h : P ⇒ Q` can be thought of as a function `h : P → Q` from proofs of `P` to proofs of `Q`.

1.4.2 Equality

Tip: "Syntactic equality is they look identical, definitional equality is they are the same, propositional equality is they turn out to be the same." – Bhavik Mehta

---

As mathematicians we tend not to fuss too much about equality, at least at undergraduate level. When formalising mathematics in Lean's type theory, it turns out that one has to think a bit more carefully about what is going on. In Lean there are three different kinds of equality which one has to be aware of, and the differences between them are "non-mathematical". The strongest kind of equality is syntactic equality; this is the kind of equality that tactics like `rw` and `simp` care about. Then there is definitional equality; this is he kind of equality that tactics like `exact` and `intro` and `rfl` care about. Finally, there is propositional equality; this is the "usual" kind of equality as understood by mathematicians.

Overview

To give you a flavour of what this document is about, and in particular to indicate that these refined notions of equality are in some sense not "mathematical", here is an example. Let `x` be a natural number. As mathematicians we would all agree that `0 + x = x` and `x + 0 = x`. Both of these are propositional equalities. However only one is a definitional equality, and neither of them are syntactic equalities.

Syntactic equality

Syntactic equality is the strongest kind of equality there is. Two expressions are *syntactically equal* if they are literally made by pressing the same keys on your keyboard in the same order.

Example: `x + 0` and `x + 0` are syntactically equal.

Non-example: `x + 0` and `x` are not syntactically equal (even though they are mathematically equal).

The `rewrite` tactic works at the syntactic equality level. For example, let's say that your tactic state looks like this:

```
a b x : ℕ
h : x + 0 = a
⊢ x = b
```

Then `rw [h]` will *fail*, even though `x + 0 = x`. The reason it will fail is that `x + 0` and `x` are not *syntactically* equal, so the `rw` tactic will fail to find the left hand side of `h` in the goal.

Definitional equality

Definitional equality is a weaker kind of equality than syntactic equality – two things can sometimes be definitionally equal without being syntactically equal. A simple example is the following. In Lean, `¬P` is notation for `not P`, and `not P` is *defined* to mean `P → False`. So whilst `¬P` and `P → False` are not syntactically equal, they are definitionally equal.

As the name suggests, definitional equality depends on definitions, and in particular depends on implementation details (that is, on exactly how things are defined under the hood). As such, definitional equality is in some sense "not a mathematical concept". Here is an example to show you what I mean.

Addition on the natural numbers is defined "by induction", or, more precisely, by recursion. If `x` and `y` are natural numbers, then in the definition of `x + y` we have to choose which one to induct on. The designers of Lean chose to induct on `y`. This means that `x + 0` is *defined* to be `x`, and `x + succ(y)` is *defined* to be `succ(x + y)`.

This means that `x + 0` and `x` are equal by the very definition of `+`. To put it another way, `x + 0` and `x` are *definitionally equal*.

However, players of the Natural number game will know, if we use this as the definition of addition, then to prove that `0 + x = x` we need to use induction. The problem is that we cannot "unfold" the definition of `0 + x` any further; the definition of `0 + x` depends on whether `x = 0` or `x = succ y` for some `y`, so to make any progress in the proof of `0 + x = x` we need to use more than just unfolding definitions; we need to use induction (to split into the cases `x=0` and `x=succ y`, when we can start simplifying `0 + x`). As a result, although `0 + x = x` is true, it is not *definitionally* true.

The fact that `x + 0 = x` is a definitional equality, but `0 + x = x` is not, means that definitional equality is in some sense not a mathematical concept. Furthermore, if the designers of Lean had decided to define addition by recursion on the first variable instead of the second, then of course our conclusions would be the other way around.

Note also: the fact that `x + 0` and `x` are definitionally equal is specific to the natural numbers. Addition of real numbers is not defined by induction, it is defined in a far more complicated way using Cauchy sequences and quotients, and if `r : ℝ` then none of `r + 0`, `r` or `0 + r` are definitionally equal to each other.

Tactics like `exact` and `rfl` work up to definitional equality. For example, the following proof works in Lean:

```
example (x : ℕ) : x + 0 = x := by
  rfl
```

which is perhaps not what you would expect if you have played the natural number game; I explicitly disabled this hack there. However the following does not work:

```
example (x : ℕ) : 0 + x = x := by
  rfl -- type mismatch
```

Similarly, this code works:

```
example (x y : ℕ) (h : x + 0 = y) : x = y := by
  exact h
```

because hypothesis `h` is definitionally equal to the goal `x = y`.

`intro` is another tactic which works up to definitional equality. If `P` is a proposition, then `¬ P` is notation for `not P`, and the *definition* of `not P` is `P → False`, so the `intro` tactic works here:

```
example (P : Prop) : ¬ P := by
  intro h
  /-
  tactic state now
```

(continues on next page)

(continued from previous page)

```
P : Prop
h : P
⊢ False
-/
sorry
```

(although the goal is of course not provable).

Propositional equality

This is the weakest kind of equality, and the kind most familiar to mathematicians. Two terms `a` and `b` are *propositionally equal* if you can prove `a = b`, or equivalently if you can construct a term `h : a = b` of type `a = b`. For example, if `x` is a natural then `x`, `x + 0`, `0 + x` and `x + 3 - 3` are all propositionally equal.

Appendix: syntactic equality again

What I said about syntactic equality is not strictly speaking true. The below paragraph fixes it, but can be ignored by everyone other then pedants.

There are actually a couple of ways that things can be syntactically equal without literally being made by pressing the same keys in the same order. Firstly, *notation* can be unfolded without breaking syntactic equality. For example the = sign in `x = y` is actually notation for the `eq` function, and the terms `x = y` and `eq x y` are syntactically equal. Secondly, the names of globally quantified variables can change without breaking syntactical equality; for example `∃ x, x^2 = 4` and `∃ y, y^2 = 4` are syntactically equal. This is because Lean "uses de Bruijn indices" under the hood, something we won't be talking about.

1.4.3 The three kinds of types

This is some background on Lean's type theory.

Introduction

Recall that every expression in Lean lives at one of three "levels" – it is either a universe, a type or a term. The universes are easy to understand; as far as this course is concerned, there are only two, namely `Type` and `Prop`. This document is about the next level down – types. It turns out that at the end of the day there are only three kinds of ways that you can make types; there are function types, inductive types and quotient types. I will go through these three kinds of types in this section, explaining abstractly how to make the type, how to make terms of that type, and how to make functions whose domain is the type.

[A technical footnote about the "meaning" of this section. You might be wondering about the sets and theorem statements you know in mathematics (recall that in Lean the type plays the role of both), and asking yourself which kind of type each of these things is. However such a question is not really mathematically meaningful; for example the type of group homomorphisms between two groups can be made either as an inductive type or a function type, and the quotient of a group by a subgroup can be made as either an inductive type or a quotient type. In fact, it is not completely clear to me that mathematicians need to know the ins and outs of these constructions if all they want to do is to prove theorems, but here is a brief overview anyway. For more detailed information, read Lean's type theory bible, Theorem Proving In Lean (sections 2, 3, 7 and 12).]

[A technical footnote about universes. if you look in the source code of `mathlib` you will find more general Type universes called `Type u` (our `Type` is just `Type 0`), and you might even see `Sort u`, which means "either `Type u`

or `Prop`. These "higher universes" are a consequence of the fact that *everything* in Lean has to have a type, so `Type` has to have a type, and this type is called `Type 1`, which has type `Type 2` etc etc. We're not doing any category theory in this course, so we will ignore these higher universes.]

Function types (a.k.a. Pi types)

Set-theoretically, if `X` and `Y` are sets, we can consider the set `Hom(X,Y)` of maps from `X` to `Y`. In Lean the corresponding type is called `X → Y` so you can think of the → symbol as "the way to make this type".

We make terms of this type using something called `fun`. You can also use λ (they keep meaning to deprecate it but it doesn't seem to have happened yet. with the annoying consequence that nobody can use λ as a variable – it is a *reserved symbol* in Lean – it means one thing, and one thing only: it's the λ in "lambda-calculus", if you've ever heard of that). We'll use `fun` below.

If you want to make the function `f(x)` from the reals to the reals sending `x` to `x^2+3` then in Lean the definition of `f` looks like this:

```
def f : ℝ → ℝ := fun x ↦ x^2+3
```

The "mapsto" symbol ↦ is typeset with `\mapsto`.

In fact Lean has a slightly more general kind of function type. The idea (expressed set-theoretically) is that if we have infinitely many sets Y_0, Y_1, Y_2,... then we can make the type of functions from the natural numbers to the union of the Y_n, but with the condition that the natural number `i` is sent to an element of Y_i. More generally (and switching back to type-theoretic language), say `I` is some index type, and for each `i : I` we have a type `Y i`. Then Lean will let you make the type of "generalised functions" which take as input a term `i` of type `I` and which spit out a term of type `Y i` – so the type of the output depends on the term which is input. Lean's notation for the type of these functions is `∀ (i : I), Y i`, or `∀ i, Y i` for short. If `f` is such a function, then the fact that the type of the output of `f` depends on the term `i` which is input means that `f` is called a "dependent function", and the type `∀ i, Y i` of `f` is called a "dependent type", or a "Pi type". Not all theorem provers have dependent types – for example the Isabelle/HOL theorem prover uses a logic called Higher Order Logic, which does not have dependent types.

Dependent types in the `Type` universe started showing up in mathematics in the middle of the 20th century. Those of you who have done some differential geometry might have seen this sort of thing before (if you haven't then perhaps ignore this paragraph!). The tangent space T_x of a manifold at a point `x` is a vector space, and these tangent spaces "glue together" to make a tangent bundle, the union of the tangent spaces; a section `s` of the tangent bundle is a function from the manifold to the union of the tangent spaces with the extra hypothesis that `s(x)` is an element of T_x for all `x`. So a section of the tangent bundle is a term of type `Π (x : X), T x`. They also show up in algebraic geometry when you start doing scheme theory (for example Hartshorne's definition of the structure sheaf on an affine scheme involves the dependent type of functions sending a prime ideal of a commutative ring to an element of the localisation of the ring at this prime ideal).

If the simplest examples I can come up with in mathematics are some fancy differential geometry and algebraic geometry examples, you might wonder whether we need to think about Pi types at all in this course. But in fact in the `Prop` universe they show up all the time! Let's consider a proof by induction. We have infinitely many true-false statements `P 0`, `P 1`, `P 2` ``,..., and we want to prove them all. In other words, we want to prove the proposition ``∀ n, P n``. This proposition, like all propositions, is a type (in the `Prop` universe) and in fact it is a Pi type, because to give a term of this type, we need to come up with a function which takes as input a natural number `n` and gives as output a proof of `P n`, that is, a term of type `P n`. So different input terms give terms in different output types. In short, whilst you can do a lot of mathematics without dependent types, we see dependent propositions everywhere. In fact the statement of Fermat's Last Theorem is a dependent type, because the type `x ^ n + y ^ n = z ^ n` depends on the terms `x`, `y`, `z` and `n`. Of course it's possible to state Fermat's Last Theorem in Isabelle/HOL or another HOL system, however the lack of dependent types might make doing modern algebraic geometry in such a system far more inconvenient.

Inductive types

Inductive types are an extremely flexible kind of type; you make them by basically listing the rules which you will allow to make terms of this type. For example, if you want Lean's version of "a set `X` with three elements `a`, `b`, and `c`" then you can make it like this:

```
inductive X : Type
| a : X
| b : X
| c : X
```

We now have a new type `X` in the system, with three terms `X.a : X`, `X.b : X` and `X.c : X`.

So that's now to make an inductive type, and how to make terms of this type. The remaining question is how to define functions from this type, or equivalently how to use terms of this type. If you want to make a function from this type, then instead of using `fun` you can use Lean's "equation compiler". Here's how to define the function from `X` to the naturals sending `X.a` to `37`, `X.b` to `42` and `X.c` to 0:

```
def f : X → ℕ
| X.a => 37
| X.b => 42
| X.c => 0
```

Note that if you `open X` then you don't have to keep putting `X.` everywhere. It is a source of some annoyance to some people that you can't use `\mapsto` here, and have to use this ASCII art `=>` instead.

You might think that this kind of construction can only make finite types, but in fact the theory of inductive types is far more powerful than this, and in particular we can make many infinite types with them. For example the definition of the natural numbers in Lean looks like this:

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

(Peano observed that these two construtions were enough to define all natural numbers) and then mathlib sets up the notation ℕ for `Nat` later. If you've played the natural number game you'll know that we can define addition and multiplication on the natural numbers, and once one has these set up one can define functions from the naturals to the naturals or other types using `fun`, for example `fun (n : ℕ) ↦ 2*n+3` defines a function from ℕ to ℕ. However we can also use the equation compiler to inductively define (or more precisely, recursively define) functions from the naturals. For example the sequence defined by `a(0)=3` and `a(n+1)=a(n)^2+37` could be defined like this:

```
def a : ℕ → ℕ
| Nat.zero => 3
| Nat.succ n => (a n)^2 + 37
```

You can make inductive propositions too. For example here are the definitions of `True` and `False` in Lean:

```
inductive True : Prop where
  | intro : True

inductive False : Prop
```

The inductive type `True` has one constructor (called `True.intro`); the inductive type `False` has no constructors. Remember that we model truth and falsehood of propositions in Lean by whether the corresponding type has a term or not. Faced with a goal of `⊢ True` you can prove it with `exact True.intro`. You cannot make a term of type `False` "absolutely" – the only time it can happen is if you are in a "relative" situation where you have hypotheses, some of which are contradictory.

If `P` and `Q` are Propositions, then you can use inductive types to make the propositions `And P Q` and `Or P Q`, with notations `P ∧ Q` and `P ∨ Q`. Here are their definitions:

```
inductive And (P Q : Prop) : Prop where
| intro (hp : P) (hq : Q) : And P Q

inductive Or (P Q : Prop) : Prop where
| intro_left (hP : P) : Or P Q
| intro_right (hQ : Q) : Or P Q
```

If you do `cases h` with `h : P ∧ Q` then, because `And` has one constructor (`And.intro`), you end up with one goal. If you do `cases h` with `h : P ∨ Q` then you end up with two goals, because `Or` has two constructors (`Or.intro_left` and `Or.intro_right`). If you do `cases h` with `h : False` then you end up with no goals, because `False` has no constructors. When Lean sees that there are no goals left, it prints `no goals`; if you have no goals left, you've proved the result you were trying to prove. It took me some time to recalibrate my thinking to this "inductive" way of thinking about logic.

We say "let `G` be a group" in Lean using inductive types, but the types involved are very simple inductive types with only one constructor. The type `Group G` is the type of group structures on `G`. There is only way to make a term of type `Group G` – you have to give a multiplication on `G`, an identity and an inverse function, and then check that it satisfies the group axioms. So the inductive type `Group G` has just one constructor which takes all of this data as input and then outputs a term of type `Group G`. We will talk more about how to make the inductive type `Group G` when we get on to groups in section 5.

Quotient types

The third kind of type which you can make in Lean is a quotient type, which I mention here only for completeness. Lean does not actually need this kind of type – it is possible to make quotient types explicitly using inductive types. However for technical reasons (which mathematicians don't need to worry about) they are a distinct primitive kind of type in Lean. The basic set-up is that you have a type `X` and an equivalence relation `R` on `X` (for some reason this is referred to as a term of type `Setoid X` in Lean), and you want to make the quotient of `X` by `R`. This is the type which mathematicians would typically refer to as "the set of equivalence classes of `R`". In Lean it's called `Quotient R`, and the map from `X` to `Quotient R` is called `Quotient.mk R : X → Quotient R`. In particular you can make terms of type `Quotient R` by applying `Quotient.mk` to terms of type `X` (this is just the construction sending an element of `X` to its equivalence class). To define a function *from* `Quotient R` we use the `Quotient.lift` function; more on this later, when we construct some quotient types familiar to mathematicians.

1.4.4 Brackets in function inputs

This is about the different types of brackets which we see in Lean's functions.

If we type `#check mul_assoc` into Lean (assuming we've done `import Mathlib.Tactic` or some other import which imports some group theory) then we get the following output:

```
mul_assoc.{u_1} {G : Type u_1} [inst✝ : Semigroup G] (a b c : G) : a * b * c = a * (b
↪* c)
```

At first glance, this makes some kind of sense: `a * b * c` means by definition `(a * b) * c` so we can see that this is some sort of claim that multiplication is associative. Looking more carefully, what is going on is that `mul_assoc` is a function, which takes as input a type `G`, a semigroup structure with the weird name `inst✝` on `G` and three terms `a`, `b` and `c` of `G`, and returns a proof that `(a * b) * c = a * (b * c)`. But what's with all the different kinds of brackets? We can see *{}*, *[]* and *()*. There's even a fourth kind, although it's rarer: try `#check DirectSum.linearMap_ext` to even see some {|} brackets. These are the four kinds of brackets which you can use for function input variables. Here's a simple explanation of what they all mean.

() brackets

These brackets are the easiest to explain. An input to a function in `()` brackets is an input which the user is expected to apply. For example, if we have a theorem `double (x : ℕ) : 2 * x = x + x` then `double 37` is the theorem that `2 * 37 = 37 + 37`.

{} and {|} brackets

These brackets exist because Lean's type theory is dependent type theory, meaning that some inputs to functions can be completely determined by other inputs.

For example, the term `Subgroup.mul_mem` is a proof of the theorem stating that if two elements of a group are in a given subgroup, then their product is also in this subgroup. The type of this term is the following:

```
Subgroup.mul_mem.{u} {G : Type u} [inst : Group G] (H : Subgroup G) {x y : G} (hx :␣
↪x ∈ H) (hy : y ∈ H) : x * y ∈ H
```

So `subgroup.mul_mem` takes as input the following rather long list of things. First it wants a type `G` (the *u* is a universe – ignore it). Then it wants a group structure on `G`. Next it wants a subgroup `H` of `G`, then two elements `x` and `y` of `G`, and finally two proofs; first a proof that `x ∈ H` and second a proof that `y ∈ H`. Given all of these inputs, it then outputs a proof that `x * y ∈ H`.

Now let's imagine we're actually going to use this proof-emitting function to prove some explicit statement. We have some explicit group, for example the symmetric group S_5, and some explicit subgroup H and some explicit permutations `x` and `y` in S_5, and proofs `hx` and `hy` that `x ∈ H` and `y ∈ H`. At the point where we feed in the input `hx` into `subgroup.mul_mem`, Lean can look at `hx` and see immediately what `x` is (by looking at the type of `hx`) and what `G` is (by looking at the type of `x`). So, when you think about it, it's a bit pointless asking the *user* to explicitly supply `G` and `x` as inputs, even though the function needs them as inputs, because actually the type of the input `hx` (namely `x ∈ H`) contains enough information to uniquely determine them.

Calculations like are what Lean's *unifier* does, and the `{}` and {|} brackets are for this purpose; they mean that they are inputs to the function which the user need not actually supply at all; Lean will figure them out.

Technical note: The difference between `{}` and {|} is that one is more *eager* than the other; this is all about the exact timing of the unifier. Basically if you have `f (a : X) {b : Y} (c : Z)` and `g (a : X) {|b : Y|} (c : Z)` then the unifier will attempt to figure out `b` in `f` the moment you have given it `f a`, but it will only start worrying about `b` in `g` when you have given it `g a c`. For an example where this matters, see section 6.5 of Theorem Proving In Lean . If you want a rule of thumb: use `{}`.

[] brackets

Like `{}` brackets, these square brackets are inputs which the user does not supply, but which the system is going to figure out by itself. The `{}` brackets above were figured out by Lean's unification system. The `[]` brackets in this section are figured out by Lean's type class inference system.

Lean's type class inference system is a big list of facts. For example Lean knows that the reals are a field, that the natural numbers are an additive monoid, that the rationals have a 0 and a 1, etc etc. Which facts does this system know? The facts it knows are "instances of classes", or "instances of typeclasses" to give them their full name.

Let's take a look at `add_comm`, the proof that addition is commutative. You can see its type with `#check add_comm`. Its type is this:

```
add_comm.{u} {G : Type u} [inst : AddCommMagma G] (a b : G) : a + b = b + a
```

An `AddCommMagma` is something a bit weaker than an additive commutative group; any abelian group with group law + is an `AddCommMagma`.

The only inputs in round brackets to this proof are `a` and `b`. Here's a short script which gives `add_comm` all the inputs it needs.

```
import Mathlib.Tactic

def a : ℝ := 37
def b : ℝ := 42

#check add_comm a b -- add_comm a b : a + b = b + a
```

The `add_comm` function was given `a`, and Lean knows that `a` has type `ℝ` because that's part of the definition of `a`. So the unifier figures out that `G` must be `ℝ`. The one remaining input to the function is a variable with the weird name of `inst✝`, whose type is `AddCommMagma ℝ`; you can think of it as "a proof that the reals are an additive commutative magma" but it's actually more than just a proof – it's the data of the addition as well; the functions and constants as well as the proofs. Where does Lean get this variable `inst✝` from?

The answer is that in `mathlib` somewhere, someone proved that the real numbers were a field, and they tagged that result with the `@[instance]` attribute, meaning that the typeclass inference system now knows about it. The typeclass inference system knows that every field is an additive commutative group, and that every additive commutative group is an additive commutative magma. So the system throws this package together and fills in the `inst✝` input automatically for you. Basically this system is in charge of keeping all the group and ring proofs which we don't want to bother about ourselves.

You can add new facts into the typeclass system. Here's a way of telling Lean that you want to work with an abstract additive commutative group `G`.

```
import Mathlib.Tactic

variable (G : Type) [AddCommGroup G] (x y : G)

#check add_comm x y -- add_comm x y : x + y = y + x
```

Instead of using concrete types like the reals, we make a new abstract type `G`, give it the structure of an additive commutative group, and let `x` and `y` be abstract elements of `G` (or more precisely terms of type `G`). The `variable` line has square brackets in too – this means "add the fact that `G` is an additive commutative group to the typeclass system". Then when `add_comm` runs, the system will supply the proof that `G` is an additive commutative magma, so the function `add_comm x y` runs successfully and outputs a proof that `x + y = y + x`.

Overriding brackets

You may well never need to do this in this course, but I put it here for completeness.

Sometimes the system goes wrong, and Lean cannot figure out the inputs it was supposed to figure out by itself. If this happens, you can override the system like this:

```
import Mathlib.Tactic

/-
add_comm {G : Type} [inst✝ : AddCommMagma G] (a b : G) : a + b = b + a
-/

-- override `{}` input
#check add_comm (G := ℝ) -- add_comm : ∀ (a b : ℝ), a + b = b + a
```

1.4.5 Notation and precedence

This is about why `a + b + c` means `(a + b) + c` but why `a + b * c` means `a + (b * c)` in Lean.

Functions and brackets

As you might have spotted, functions in Lean don't need brackets for their inputs.

```
example (f : ℕ → ℕ) (a : ℕ) : f (a) = f a := by
  rfl
```

In fact Lean is keen to drop brackets wherever they are not required: if you look at the goal before `rfl` in the proof above it says `⊢ f a = f a`. Similarly if you check that multiplication is associative on the naturals:

```
import Mathlib.Tactic

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

then the goal before `ring` is displayed as `⊢ a * b * c = a * (b * c)`, with `a * b * c` instead of `(a * b) * c`. What's going on here?

Lean's *parser* has the job of changing human input (strings) into abstract Lean terms, and for reasons we'll get to later, it parses `a * b * c` as `(a * b) * c`. Lean's *pretty printer* has the job of changing Lean terms back into strings so that the tactic state can be displayed on the screen, and it will drop brackets wherever possible, so it will change `(a * b) * c` back into `a * b * c`. If you try the above example in VS Code and hover over each of the `*` s in the infoview (the tactic state), you will see a way of figuring out where Lean internally puts the brackets (try it to see what I mean).

So what is going on here? When are brackets needed, and where does Lean put them?

Functions are greedy

Lots of things in Lean are functions (in fact probably more things than you expect are functions). And in Lean, functions are *greedy* – they will eat the next thing they see. So here is an example where one pair of brackets are needed:

```
example (X Y Z : Type) (f : X → Y) (g : Y → Z) (x : X) : Z := g (f x)
```

If you just write $g\ f\ x$ then g will eat f instead of $f\ x$, and then it will complain that what it ate didn't have the right flavour, i.e. the right type. Try the example above in Lean, and then remove the brackets and read the error message – this is an error message you will see a lot as you're learning Lean, so it's a good one to understand.

Of course it's fine to put in too many brackets: Lean will be happy with `g (f (x))`, it's just that the brackets around the `x` are not necessary.

Functions vs notation

In Lean + is a thing, but it's not actually the name of a function. There is no `definition + : X → X → X` anywhere in Lean because + is just *notation*. The actual function is called `hAdd` (or, to be completely pedantic, `HAdd.hAdd`), and then somewhere in core Lean there is the line

```
infixl:65 " + "   => HAdd.hAdd
```

which assigns *notation* to the `hAdd` function. Basically this just means that if the parser sees `a + b` it will parse it as `hAdd a b` (or, to be completely pedantic, `HAdd.hAdd a b`). Now all functions are equally greedy – but notation is not. Both addition and multiplication are functions, but as we know from BIDMAS, we want Lean to parse multiplication before addition, so the system will somehow need to know that notation for multiplication has a higher "score" than notation for addition. And if we look in core Lean, in the file *Notation.lean*, just below the definition of + we see

```
infixl:70 " * "   => HMul.hMul
```

which should give you a clue. The *binding power* of a notation is a number associated to it when the notation is defined, and the larger the number, the more tightly the notation binds. Functions are greedy – function application has binding power 1024. Because `1024 > 70 > 65`, this is why Lean's parser parses `f x * y` as `f(x) * y` and `a + b * c` as `a + (b * c)`. Also the `l` in `infixl` means "make this notation left associative"; compare this to

```
infixr:80 " ^ "   => HPow.hPow
```

which makes exponentiation right associative (the reason for this is that `(a^b)^c` can be simplified to `a^(b*c)` so is not as useful as `a^(b^c))`.

Examples of binding power

As we've seen, + has binding power 65, and binary − (that is, the function which takes two variables `a` and `b` and returns `a - b`) also has binding power 65. There is also unary −, which is notation for the function which takes one variable `a` and returns `-a`, and this has binding power 75. Multiplication and division have binding power 70. As a result, `-b / c` means `(-b) / c` but `a - b / c` means `a - (b / c)`.

All notation has a binding power! Lean's version of BIDMAS does not just work for arithmetic operators like + and *, notation like ∧ and even = has a binding power. Here are some examples of binding powers:

```
∨ : 30
∧ : 35
¬ : 40
= : 50
+ : 65
- : 65 (binary)
* : 70
/ : 70
- : 75 (unary)
^ : 80
⁻¹ : 1024
```

1.4.6 Structures

A lot of the time in this course we are concerned with proving theorems. However sometimes it's interesting to make a new definition, and then prove theorems about the definition. In section 3 we made things like groups, subgroups, and group homomorphisms from first principles, even though mathlib has them already. We made them using structures and classes.

Let's start by talking about the structure `Equiv X Y`, with notation `X ≃ Y`.

Equiv – two inverse bijections

Let X and Y be types. Here's how `mathlib` defines a type `X ≃ Y` whose elements are bijections from `X` to `Y`.

```
/-- `X ≃ Y` is the type of functions from `X` to `Y` with a two-sided inverse. -/
structure Equiv (X Y : Type) where
  toFun : X → Y
  invFun : Y → X
  left_inv : LeftInverse invFun toFun
  right_inv : RightInverse invFun toFun

infixl:25 " ≃ " => Equiv
```

What's going on here? The actual name of the type is `Equiv X Y`, and we set up a (left-associative, binding power 25) notation `X ≃ Y` for it (as a rule of thumb, if it's a funny maths character then it's notation, and if it's spelt out in normal letters it's the official name). The line starting with `/--` is the docstring for the definition; this ensures that when you hover over `Equiv` you can see its definition. Note that every definition you write should have a docstring!

The structure we're defining here has four *fields*: two functions and two proofs. To make a term of this type, you have to supply four things:

(1) a map from `X` to `Y`

(2) a map from `Y` to `X`

(3) a proof that if you do the map from `X` to `Y` and then the map from `Y` to `X` you get back to where you started;

(4) same as (3) but start at `Y` then go to `X` then back to `Y`.

In other words, you need to supply two maps and then a proof that one is the two-sided inverse of the other.

If you have read something about inductive types – `X ≃ Y` is an inductive type, with one constructor (called `Equiv.mk` internally) which takes as inputs the four things above, and then outputs a term of type `X ≃ Y`.

Let's let `X` and `Y` both be the integers, and let's let the map from `X` to `Y` send `n` to `n+37`. Then the inverse map sends `n` to `n-37`, and the `ring` tactic will be able to prove that `(n+37)-37=n`, so we should be able to make a term of type `ℤ ≃ ℤ` using this data. Let's call it `e`. By the way, I started writing this definition by typing `def e : ℤ ≃ ℤ := _` and then clicking on the blue lightbulb and selecting the option which mentioned structures.

```
def e : ℤ ≃ ℤ where
  toFun n := n + 37
  invFun m := m - 37
  left_inv := by
    intro n
    -- some messy goal with a bunch of ``fun`` in
    dsimp only -- tidy up
    -- ⊢ n + 37 - 37 = n
    ring
  right_inv := by
    intro m
    ring -- don't actually need to tidy up
```

You can think of `e` as a 4-tuple: two functions, and two proofs. How do you get this information from the term `e`? You can use functions like `Equiv.toFun` etc, which were created under the hood when `Equiv` was defined. Here are some examples of how to use the "parts" of `e`. Note that because the type of `e` is `Equiv [something]`, `e.toFun` is short for `Equiv.toFun e`. This is Lean's *dot notation* in action.

```
#check Equiv.toFun e -- ℤ → ℤ
#check e.toFun -- ℤ → ℤ
```

(continues on next page)

(continued from previous page)

```
example : e.toFun 1 = 38 := by
  simp only [e] -- replace ``e`` with its definition
  -- ⊢ 1 + 37 = 38
  norm_num

example (x : ℤ) : e.invFun x = x - 37 := by
  rfl

example : ∀ x : ℤ, e.invFun (e.toFun x) = x :=
  e.left_inv
```

The final thing I'll explain is the *coercion* associated to the bijection. We sometimes want to think of `e` as a function from $\mathbb{Z}$ to $\mathbb{Z}$, and forget the fact that it's really an `Equiv`. We could do this by talking about `e.toFun` all the time, but Lean will just let you use `e` as a function:

```
example : e 1 = 38 := by
  dsimp [e]
  -- ⊢ 1 + 37 = 38
  norm_num
```

Lean has things called *coercions*. This is a way that given a term `x` of one type, you can "pretend" that it has a different type. What is actually happening is that a function which is essentially invisible to mathematicians is being called; it sends a bijection to the associated function! There's a coercion defined from `Equiv X Y` to `X → Y` and this coercion is applied automatically when `e` recieves the "input" `1`.

1.4.7 Classes

In the previous section, we saw how to make structures. Here is another example of a structure: the structure of a group homomorphism.

```
import Mathlib.Tactic

structure GroupHom (G H : Type) [Group G] [Group H] where
/-- The underlying function -/
toFun : G → H
/-- The proposition that the function preserves multiplication -/
map_mul : ∀ x y : G, toFun (x * y) = toFun x * toFun y
```

If `G` and `H` are groups, then a to give a group homomorphism from `G` to `H` is to give a function from `G` to `H` which preserves multiplication, so a term of type `GroupHom G H` is a *pair*, consisting of a function and a proof. But what does that `[Group G]` mean and why is it in weird brackets?

So the deal is that `Group` is a *class*, which is simply a structure tagged with the `@[class]` attribute. The definition of a group might look like this (I'll call it `MyGroup` because `Group` is already used):

```
import Mathlib.Tactic

class MyGroup (G : Type) extends Mul G, One G, Inv G where
mul_assoc (a b c : G) : (a * b) * c = a * (b * c)
one_mul (a : G) : 1 * a = a
mul_one (a : G) : a * 1 = a
mul_left_inv (a : G) : a⁻¹ * a = 1
mul_right_inv (a : G) : a * a⁻¹ = 1
```

The `extends Mul G` etc stuff just means "assume `G` already has a `*` defined on it", so `a * b` makes sense if `a b : G`. Similarly we assume there's an element of `G` called `1`, and that if `a : G` then there's an element a^{-1} `: G`. The structure of a group on `G` is then, on top of this data, proofs of the axioms of a group.

So why is this a `class`, whereas group homomorphisms were a `structure`? Mathematically a group and a group homomorphism are just the same sort of object: both are a "list of stuff". The difference is in how we want to use this stuff. If φ is a group homomorphism then we would expect to mention φ explicitly when talking about it, and there can be more than one group homomorphism between two groups. In contrast, we would only usually expect one group structure (with group law `*`) on a type `G`, rather than the option of moving between several. Furthermore, if we have a group structure `i` on a type `G` then we would *not* expect to keep having to mention `i` (which is a big list consisting of a multiplication, identity, inverse and proof of five axioms) explicitly, we just want to write `g * h` if `g h : G` and don't want to talk about `i.mul` or whatever it would be called. We just want the multiplication notation to *work*.

Lean's *typeclass inference system* (or "square bracket system") informally keeps track of classes and instances, meaning that we can write mathematics naturally without having to write down names of structures which are invisible to mathematicians. We don't say "let `G` be a set and let `i` be a group structure on `G`"; similarly we don't have to name `i : Group G` here; we just write `[Group G]` and Lean assigns some internal name like `inst✝ : Group G` to the package. Let's take a look at the type of `mul_left_inv`:

```
mul_left_inv {G : Type} [inst✝ : Group G] (a : G) : a⁻¹ * a = 1
```

If we try to use this theorem (e.g. with `rw [mul_left_inv]` when our goal is `x = a⁻¹ * a`) then Lean looks at the type of `a` (which could be an abstract type like `G` or a concrete type like $\mathbb{N}$, supplies the type of `a` as the first input `G` to `mul_left_inv`, and then asks the square bracket machine to look through its big list of *instances* to see if it can find a term of type `Group G`. If it can, then it will use that term to fill in the second input to `mul_left_inv`. If it can't, then it will complain that it can't find a group structure on `G`. So the square bracket system is a way of telling Lean "I want to use the group structure on `G`" without having to name it.

Summary

We use structures when there is likely to be more than one term of the relevant type (e.g. it's fine to consider several group homomorphisms from `G` to `H`) and we use classes when there is likely to be only one term of the relevant type (e.g. it would be very confusing to have two group structures on `G` both with group law written `*`, so it's safe to make `Group G` into a class). Under the assumption that there is only ever going to be at most one instance of a class, the typeclass inference system (or square bracket system) can be used to find it.

1.4.8 Dot notation

Say you have a type, like `Subgroup G`, and a term of that type, like `H : Subgroup G`. Say you have a function in the `Subgroup` namespace which takes as input a term of type `Subgroup G`, for example `Subgroup.inv_mem`, which has as an explicit input a term `H : Subgroup G` and spits out a proof that `x ∈ H → x⁻¹ ∈ H`. Then instead of writing `Subgroup.inv_mem H : x ∈ H → x⁻¹ ∈ H` you can just write `H.inv_mem : x ∈ H → x⁻¹ ∈ H`.

In general the rule is that if you have a term `H : Foo ...` of type `Foo` or `Foo A B` or whatever, and then you have a function called `Foo.bar` which has an explicit (i.e., round brackets) input of type `Foo ...`, then instead of `Foo.bar H` you can just write `H.bar`. This is why it's a good idea to define theorems about `Foo``s in the ``Foo` namespace.

This sort of trick can be used all over the place; it's surprisingly powerful. For example Lean has a proof `Eq.symm : x = y → y = x`. If you have a term `h : a = b` then, remembering that `=` is just notation for `Eq` so that `h` really has type `Eq a b`, you can write `h.symm : b = a` as shorthand for `Eq.symm h`.

1.4.9 Coercions

Sometimes you have a term of a type, and you really want it to have another type (because that's what we do in maths; we are liberal with our types, unlike Lean). For example you might have a natural number `n : ℕ` but a function `f : ℝ → ℝ` (like `Real.sqrt` or similar), and you want to consider `f n`. This is problematic in a strongly typed language like Lean: `n` has type `Nat` and not `Real`, so `f n` does not make sense. However, if you try it…

```
import Mathlib.Tactic

def a : ℕ := 37

#check Real.sqrt a -- real.sqrt ↑a : ℝ
```

…it works anyway! But actually looking more closely, something funny is going on; what is that ↑ by the `a`? That up-arrow is Lean's notation for the completely obvious function from ℕ to ℝ which doesn't have a name in mathematics but which Lean needs to apply in order for everything to typecheck.

Coercion to function

Here's another example of something which shouldn't work but which does:

```
import Mathlib.Tactic

variable (G H : Type) [Group G] [Group H] (φ : G →* H) (g : G)

#check φ g -- ↑φ g : H
```

Here φ is a group homomorphism, so in particular it is *not a function*, it is a pair consisting of a function and a proof that the function preserves multiplication. But we treat it as a function and it works anyway, because there is a coercion from the type `G →* H` to the type `G → H`, indicated by an arrow.

Coercion to type

A subset of the reals is a term, not a type. The type is `Set ℝ` of *all* subsets of the reals, so here `s : Set ℝ` is a term, not a type, and so `a : s` shouldn't even make sense. But if you look carefully, you see that the type of `a` is in fact `↑s`, because `s` has been coerced from a term to the corresponding subtype `{x : ℝ // x ∈ s}`.

```
import Mathlib.Tactic

example (s : Set ℝ) (a : s) : a = a := by
  /-
  s : Set ℝ
  a : ↑s
  ⊢ a = a
  -/
  rfl
```

A term of the subtype `{x : ℝ // x ∈ s}` is a pair consisting of a term `x : ℝ` and a proof that `x ∈ s`.

1.4.10 The axiom of choice

The axiom of choice was something I found quite complicated as an undergraduate; it was summarised as "you can make infinitely many choices at the same time" but because I didn't really know much about set theory it took me a while before this description made any sense.

In Lean's type theory, the situation is much simpler (at least in my mind). There are two universes in Lean (at least for the purposes of this course) – `Prop` and `Type`. The `Prop` universe is for proofs, and the `Type` universe is for data. The axiom of choice is a route from the `Prop` universe to the `Type` universe. In other words, it is a way of getting data from a proof.

`Nonempty X`

If `X : Type` then `Nonempty X : Prop` is the true-false statement asserting that `X` is nonempty, or equivalently, that there's a term of type `X`. For example here's part of the API for `Nonempty`:

```
import Mathlib.Tactic

example (X : Type) : (∃ x : X, True) ↔ Nonempty X := by
  exact exists_true_iff_nonempty
```

This example shows that given a term of type `Nonempty X`, you can get a term of type `∃ x : X, True` (i.e., "there exists an element of `X` such that a true statement is true"). We would now like to go from this to actually *get* a term of type `X`! In constructive mathematics this is impossible: because `h` lives in the `Prop` universe, Lean forgot how it was proved. However in classical mathematics we can pass from the `Prop` universe to the `Type` universe with `Classical.choice h`, a term of type `` `X` ``. This gives us a "noncomputable" term of type `X`, magically constructed only from the proof `h` that `X` was nonempty. Mathematically, "noncomputable" means "it exists, but we don't actually have an algorithm or a formula for it". This is a subtlety which is often not explicitly talked about in mathematics courses, probably because it is often not relevant in a mathematical argument; a proof in classical mathematics is not a program, so we don't need an algorithm or formula for the terms involved.

You might wonder what the code for this `Classical.choice` function looks like, but in fact there isn't any code for it; Lean simply declares that `Classical.choice` is an `axiom`, just like how in set theory the axiom of choice is declared to be an axiom.

When I first saw this axiom, it felt to me like it was way weaker than the set theory axiom of choice; in set theory you can make infinitely many choices of elements of nonempty sets all at once, whereas in Lean we're just making one choice. But later on I realised that in fact you could think of it as much *stronger* than the set theory axiom of choice, because you can interpret `Classical.choice` as a function which makes, once and for all, a choice of an element of *every single nonempty type*, so it easily implies the usual set-theoretic axiom of choice.

Classical.choose

In my experience, the way people want to use the axiom of choice when doing mathematics in Lean is to get an element of X not from a hypothesis `∃ x : X, true`, but from a hypothesis like `∃ x : ℝ, x^2 = 2` or more generally `∃ x : X, p x` where `p : X → Prop` is a predicate on `X`. The way to do this is as follows: you run `Classical.choose` on `h : ∃ x : X, p x` to get the element of `X`, and the proof that this element satisfies `p` is `Classical.choose_spec h`. Here's a worked example.

```
import Mathlib.Tactic
import Mathlib.Analysis.Complex.Polynomial -- import proof of fundamental theorem of␣
↪algebra

open Polynomial -- so I can use notation ℂ[X] for polynomial rings
```

(continues on next page)

(continued from previous page)

```
                -- and so I can write `X` and not `polynomial.X`

suppress_compilation -- because everything is noncomputable

def f : ℂ[X] := X^5 + X + 37 -- a random polynomial

lemma f_degree : degree f = 5 := by
  unfold f
  compute_degree -- polynomial degree computing tactic
  norm_num

theorem f_has_a_root : ∃ (z : ℂ), f.IsRoot z := by
  apply Complex.exists_root -- the fundamental theorem of algebra
  -- ⊢ 0 < degree f
  rw [f_degree]
  -- ⊢ 0 < 5
  norm_num

-- let z be a root of f (getting data from a theorem)
def z : ℂ := Classical.choose f_has_a_root

-- proof that z is a root of f (the "API" for `Classical.choose`)
theorem z_is_a_root_of_f : f.IsRoot z := by
  exact Classical.choose_spec f_has_a_root
```

1.4.11 How to format your code well

The first time this course ran I did not emphasize good code layout and when I was marking the projects I regretted this. Formatting your code correctly helps a great deal with readability (as I discovered) and so I now look more favourably on people who do this properly. The code I write in the course repository should always conform to the correct standards. Here are the basics.

Indentation

Code in a tactic block gets indented two spaces.

```
import Mathlib.Tactic

example (a b : ℕ) : a = b → a ^ 2 = b ^ 2 := by -- `by` at the end
  intro h -- proof is indented two spaces in
  rw [h]
```

If you have a long theorem statement and want to write it over two or more lines then you should indent subsequent lines with *four* spaces, for example:

```
import Mathlib.Data.Nat.Basic

theorem le_induction {P : Nat → Prop} {m}
    (h0 : P m) (h1 : ∀ n, m ≤ n → P n → P (n + 1)) :
    ∀ n, m ≤ n → P n := by
  apply Nat.le.rec
  · exact h0
  · exact h1 _
```

Spaces between operators

`a = b`, not `a=b`. See above. Similarly `a + b`, `a ^ b` and so on. Also `x : T` not `x:T`, and `foo := bar` not `foo:=bar`.

Comments

You don't have to put a comment on every line of code, but please feel free to put comments at points where something is actually happening. Code with comments is easier to read. Just one line `-- write explanation here` above a tactic is already helpful. For multi-line comments, use `/-` and `-/`

Have only one goal

Sometimes you can end up with more than one goal. This can happen for two reasons. Firstly, perhaps you manually created a new goal. For example, perhaps you wrote `have intermediate_result : a = b + c := by` `` or ``suffices h : a = b + c by``. You just created an extra goal on top of the goal which was already there, so the proof should be indented two more spaces.

```
import Mathlib.Tactic

example (a b c : ℕ) (h : a = b) : a ^ 2 + c = b ^ 2 + c := by
  have h2 : a ^ 2 = b ^ 2 := by -- you made a second goal
    rw [h]
    -- other lines of code would go here, also indented
  rw [h2] -- back to only two space indentation
```

The other way it can happen is if you use a tactic or apply a function which changes your old goal into more than one goal.

```
import Mathlib.Tactic

example (P Q : Prop) (hP : P) (hQ : Q) : P ∧ Q := by
  constructor -- this tactic replaced the goal we were working on with two goals
  · -- so use the right kind of dot (`\.`) and work with one goal at a time
    exact hP
  · exact hQ -- note also extra indentation
```

Module docstrings

For your projects, you might want to consider writing "module docstrings", which is a fancy name for "a big comment at the top of each file explaining what happens in the file." Look at any file in mathlib to see an example, or you can see one here .

Want to know more?

Check out the mathlib library style guidelines on the Lean community pages – not all of it applies to us, but most of the section on tactic mode does.

1.5 Part C: Tactics

Here is some documentation for the tactics which we will be using in the course. Note also that the Lean community website has extensive documentation on all the tactics in Lean and mathlib. Check out the tactic page on the community website here if you want to explore beyond the tactics below, or see another take on what they do.

1.5.1 Tactic cheatsheet

If you think you know which part of the tactic state your "next move" should relate to (i.e. you know whether you should be manipulating the goal or using a hypothesis) then the below table might give you a hint as to which tactic to use.

Table 1: Cheat sheet

Form of proposition	In the goal?	Hypothesis named `h`?
`P → Q`	`intro hP`	`apply h` (if goal is `Q`) or `apply h at...`
`True`	`trivial`	(can't be used)
`False`	(can't be used)	`exfalso, exact h` or `cases h`
`¬P`	`intro hP`	`apply h` (if goal is `False`)
`P ∧ Q`	`constructor`	`cases' h with hP hQ`
`P ↔ Q`	`constructor`	`rw h` (or `cases' h with h1 h2`)
`P ∨ Q`	`left` or `right`	`cases' h with hP hQ`
`∀ (a : X), ...`	`intro x`	`specialize h x`
`∃ (a : X), ...`	`use x`	`cases' h with x hx`

1.5.2 Tactic documentation

apply

Warning: The `apply` tactic does *two very specific things*, which I explain below. I have seen students trying to use this tactic to do all sorts of other things, based solely on the fact that they want to "apply a theorem" or "apply a technique". The English word "apply" has **far more uses** than the Lean `apply` tactic.

Used alone, the `apply` tactic "argues backwards", using an implication to reduce the goal to something closer to the hypotheses. `apply h` is like saying "because of `h`, it suffices to prove this new simpler thing".

Used "at" another hypothesis, it "argues forwards", using the proof of an implication `P → Q` to change a hypothesis of `P` to one of `Q`.

Summary

If your local context is

```
h : P → Q
⊢ Q
```

then `apply h` changes the goal to `⊢ P`.

And if you have two hypotheses like this:

```
h : P → Q
h2 : P
```

then `apply h at h2` changes `h2` to `h2 : Q`.

Note: `apply h` and `apply h at h'` will *not work* unless `h` is of the form `P → Q`. `apply h` will also *not work* unless the goal is equal to the conclusion `Q` of `h`, and `apply h at h2` will *not work* unless `h2` is a proof of the premise `P` of `h`. You will get an obscure error message if you try using `apply` in situations which do not conform to the pattern above.

Mathematically, `apply ... at ...` is easy to understand; we have a hypothesis saying that `P` implies `Q`, so we can use it to change a hypothesis `P` to `Q`. The bare `apply` tactic is a little harder to understand. Say we're trying to prove `Q`. If we know that `P` implies `Q` then of course it would suffice to prove `P` instead, because `P` implies `Q`. So `apply` *reduces* the goal from `Q` to `P`. If you like, `apply` executes the *last* step of a proof of `Q`, rather than what many mathematicians would think of as the "next" step.

If instead of an implication you have an iff statement `h : P ↔ Q` then `apply h` won't work. You might want to "apply" `h` by using the *rw* (rewrite) tactic.

Examples

1) If you have two hypotheses like this:

```
h : x = 3 → y = 4
h2 : x = 3
```

then `apply h at h2` will change `h2` to a proof of `y = 4`.

2) If your tactic state is

```
h : a ^ 2 = b → a ^ 4 = b ^ 2
⊢ a ^ 4 = b ^ 2
```

then `apply h` will change the goal to `⊢ a ^ 2 = b`.

3) `apply` works up to *definitional equality*. For example if your local context is

```
h : ¬P
⊢ False
```

then `apply h` works and changes the goal to `⊢ P`. This is because `h` is definitionally equal to `P → False`.

4) `apply` can also be used in the following situation:

```
import Mathlib.Tactic

example (X Y : Type) (φ ψ : X → Y) (h : ∀ a, φ a = ψ a) (x : X) :
   φ x = ψ x := by
apply h
done
```

Here `h` can be thought of as a function which takes as input an element of `X` and returns a proof, so `apply` makes sense.

5) The `apply` tactic does actually have one more trick up its sleeve: in the situation

```
h : P → Q → R
⊢ R
```

the tactic `apply h` will work (even though the brackets in `h` which Lean doesn't print are `P → (Q → R)`), and the result will be two goals `⊢ P` and `⊢ Q`. Mathematically, what is happening is that `h` says "if `P` is true, then if `Q` is true, then `R` is true", hence to prove `R` is true it suffices to prove that `P` and `Q` are true.

Further notes

The *refine* tactic is a more refined version of `apply`. For example, if `h : P → Q` and the goal is `⊢ Q` then `apply h` does the same thing as `refine h ?_`.

assumption

Summary

The `assumption` tactic closes a goal `⊢ P` when there is a hypothesis `h : P`. Note that `exact h` also closes this goal, and is shorter too, so only use this when the name of the hypothesis is five or more letters long ;-) Other uses include when one is trying to be clever and using `<;>` to solve more than one goal at once.

Examples

1) Faced with the goal

```
reallylonghypothesisname : P
h1 : Q
h2 : 2 + 2 = 5
...
h100 : x = x
⊢ P
```

you could do `exact reallylonghypothesisname` but `assumption` works as well. Although what are you doing with a really long hypothesis name?

2) If your tactic state is

```
hP : P
hQ : Q
⊢ P ∧ Q
```

you can do `constructor <;> assumption`. The `<;>` means "do `constructor` and then do `assumption` on all goals produced by `constructor`".

Further notes

If you decide that you are interested in learning how to *write* tactics then `assumption` is usually a good one to try and write (you loop through the hypotheses, trying to close the goal with each one). Note that there will be nothing about tactic writing in this course because the author doesn't have the first clue about it.

by_cases

Summary

All propositional logic problems can in theory be solved by just throwing a truth table at them. The `by_cases` tactic is a simple truth table tactic: `by_cases P` turns one goal into two goals, with `P` is assumed in the first, and `¬P` in the second.

Examples

1) If `P` is a proposition, then `by_cases hP : P` turns your goal into two goals, and in each of your new tactic states you have one extra hypothesis. In the first one you have a new hypothesis `hP : P` and in the second you have a new hypothesis `hP : ¬P`.
2) Note that if you just write `by_cases P` then Lean will call the hypothesis `h` which is its way of saying "if you don't name your hypotheses then I will give them names which you can't use".

Details

For those of you interested in constructive mathematics (a weakened form of mathematics much beloved by some computer scientists), the `by_cases` tactic (like the *by_contra* tactic) is not valid constructively. We are doing a case split on `P ∨ ¬P`, and to prove `P ∨ ¬P` in general we need to assume the law of the excluded middle. In this course we will not be paying any attention to any logic other than classical logic, meaning that this case split is mathematically valid.

Further notes

There's a much higher powered tactic which does case splits on all propositions and grinds everything out: the `tauto` tactic. The `tauto` tactic probably proves every goal in all the questions in section 1 of the course repo, however you won't learn any other tactics if you keep using it!

by_contra

Summary

The `by_contra` tactic is a "proof by contradiction" tactic. If your goal is `⊢ P` then `by_contra h` introduces a hypothesis `h : ¬P` and changes the goal to `False`.

Example

Here is a proof of `¬ ¬ P → P` using `by_contra`

```
example (P : Prop) : ¬ ¬ P → P := by
  intro hnnP -- assume ¬ ¬ P
  by_contra hnP -- goal is now `False`
  apply hnnP -- goal is now ¬ P
  exact hnP
```

Make a new Lean file in a Lean project and cut and paste the above code into it. See if you can understand the logic.

Further notes

The `by_contra` tactic is strictly stronger than the `exfalso` tactic in that not only does it change the goal to `False` but it also throws in an extra hypothesis.

cases'

Summary

`cases'` is a general-purpose tactic for "deconstructing" hypotheses. If `h` is a hypothesis which somehow "bundles up" two pieces of information, then `cases' h with h1 h2` (note the dash!) will make hypothesis `h` vanish and will replace it with the two "components" which made the proof of `h` in the first place. Variants are `cases` (which does the same thing but with a far more complicated syntax which computer scientists seem to get very excited about) and `rcases` (which is "recursive `cases`" and which has its own page here: *rcases*)

Examples

1) The way to make a proof of `P ∧ Q` is to use a proof of `P` and a proof of `Q`. If you have a hypothesis `h : P ∧ Q`, then `cases' h` will delete the hypothesis and replace it with hypotheses `left✝ : P` and `right✝ : Q`. That weird dagger symbol in those names means that you can't use these hypotheses explicitly! It's better to type `cases' h' with hP hQ`.
2) The way to make a proof of `P ↔ Q` is to prove `P → Q` and `Q → P`. So faced with `h : P ↔ Q` one thing you can do is `cases' h with hPQ hQP` which removes `h` and replaces it with `hPQ: P → Q` and `hQP: Q → P`. Note however that this might not be the best way to proceed; whilst you can `apply hPQ` and `hQP`, you lose the ability to rewrite `h` with `rw h`. If you really want to deconstruct `h` but also want to keep a copy around for rewriting later, you could always try `have h2 := h` then `cases' h with hPQ hQP`.
3) There are two ways to make a proof of `P ∨ Q`. You either use a proof of `P`, or a proof of `Q`. So if `h : P ∨ Q` then `cases' h with hP hQ` has a different effect to the first two examples; after the tactic you will be left with *two* goals, one with a new hypothesis `hP : P` and the other with `hQ : Q`. One way of understanding why this happens is that the *inductive type* `Or` has two constructors, whereas `And` only has one.
4) There are two ways to make a natural number `n`. Every natural number is either `0` or `succ m` for some natural number `m`. So if `n : ℕ` then `cases' n with m` gives two goals; one where `n` is replaced by `0` and the other where it is replaced by `succ m ``. Note that this is a strictly weaker version of the ``induction` tactic, because `cases'` does not give us the inductive hypothesis.
5) If you have a hypothesis `h : ∃ a, a^3 + a = 37` then `cases' h with x hx` will give you a number `x` and a proof `hx : x^3 + x = 37`.

Further notes

Note that ∧ is right associative: `P ∧ Q ∧ R` means ``P ∧ (Q ∧ R)`. So if `h : P ∧ Q ∧ R` then `cases' h with h1 h2` will give you `h1 : P` and `h2 : Q ∧ R` and then you have to do `cases' h2` to get to the proofs of `Q` and `R`. The syntax `cases' h with h1 h2 h3` doesn't work (`h3` just gets ignored). A more refined version of the `cases'` tactic is the *rcases* tactic (although the syntax is slightly different; you need to use pointy brackets ⟨,⟩ with `rcases`). For example if `h : P ∧ Q ∧ R` then you can do `rcases h with ⟨hP, hQ, hR⟩`.

change

Summary

If your goal is `⊢ P`, and if `P` and `Q` are *definitionally equal*, then `change Q` will change your goal to `Q`. You can use it on hypotheses too: `change Q at h` will change `h : P` to `h : Q`. Note: `change` can sometimes be omitted, as many (but not all) tactics "see through" definitional equality.

Example

1) In Lean, `¬P` is *defined* to mean `P → False`, so if your goal is `⊢ ¬P` then `change P → False` will change it the goal to `⊢ P → False`.
2) The `rw` tactic works up to syntactic equality, not definitional equality, so if your tactic state is

```
h : ¬P ↔ Q
⊢ P → False
```

then `rw h` doesn't work, even though the left hand side of `h` is definitionally equal to the goal. However

```
change ¬P,
rw h
```

works, and changes the goal to `Q`.

3) `change` also works on hypotheses: if you have a hypothesis `h : ¬P` then `change P → False at h` will change `h` to `h : P → False`.

Details

Definitionally equal propositions are logically equivalent (indeed, they are equal!) so Lean allows you to change a goal `P` to a definitionally equal goal `Q`, because `P` is true if and only if `Q` is true.

Further notes

Many tactics work up to definitional equality, so sometimes `change` is not necessary. For example if your goal is `⊢ ¬P` then `intro h` works fine anyway, as `intro` works up to definitional equality.

`show` does the same thing as `change` on the goal, but it doesn't work on hypotheses.

choose

Summary

The `choose` tactic is a relatively straightforward way to go from a proof of a proposition of the form `∀ x, ∃ y, P(x,y)` (where `P(x,y)` is some true-false statement depend on `x` and `y`), to an actual *function* which inputs an `x` and outputs a `y` such that `P(x,y)` is true.

Basic usage

The simplest situation where you find yourself wanting to use `choose` is if you have a function `f : X → Y` which you know is surjective, and you want to write down a one-sided inverse `g : Y → X`, i.e., such that `f(g(y))=y` for all `y : Y`. Here's the set-up:

```
import Mathlib.Tactic

/-
`X` is an abstract type and `P` is an abstract true-false
statement depending on an element of `X` and a real number.
-/
example (X : Type) (P : X → ℝ → Prop)
    /-
    `h` is the hypothesis that given some `ε > 0` you can find
    an `x` such that the proposition is true for `x` and `ε`
    -/
    (h : ∀ ε > 0, ∃ x, P x ε) :
  /-
  Conclusion: there's a sequence of elements of `X` satisfying the
  condition for smaller and smaller ε
  -/
  ∃ u : ℕ → X, ∀ n, P (u n) (1/(n+1)) := by
  choose g hg using h
  /-
  g : Π (ε : ℝ), ε > 0 → X
  hg : ∀ (ε : ℝ) (H : ε > 0), P (g ε H) ε
  -/
  -- need to prove 1/(n+1)>0 (this is why I chose 1/(n+1) not 1/n, as 1/0=0 in Lean!)
  let u : ℕ → X := fun n ↦ g (1/(n+1)) (by positivity)
  use u -- `u` works
  intro n
  apply hg
```

constructor

Summary

If your goal is a proof which is "made up of subproofs" (for example a goal like `⊢ P ∧ Q`; to prove this you have to prove `P` and `Q`) then the `constructor` tactic will turn your goal into these multiple simpler goals.

Examples

1) Faced with the goal `⊢ P ∧ Q`, the `constructor` tactic will turn it into two goals `⊢ P` and `⊢ Q`.
2) Faced with the goal `⊢ P ↔ Q`, `constructor` will turn it into two goals `⊢ P → Q` and `⊢ Q → P`.
3) Something which always amuses me – faced with `⊢ True`, `constructor` will solve the goal. This is because `True` is made up of 0 subproofs, so `constructor` turns it into 0 goals.

Further notes

The `refine` tactic is a more refined version of `constructor`; faced with a goal of `⊢ P ∧ Q`, `constructor` does the same as `refine ⟨?_, ?_⟩,`. In fact `refine` is more powerful than `constructor`; faced with `⊢ P ∧ Q ∧ R` you would have to use `constructor` twice to break it into three goals, whereas `refine ⟨?_, ?_, ?_⟩` does the job in one go.

Historical remark

`constructor` was called `split` in Lean 3, but `split` in Lean 4 now does what Lean 3's `split_ifs` tactic does.

convert

Summary

If a hypothesis `h` is nearly but not quite equal to your goal, then `exact h` will fail, because theorem provers are fussy about details. But `convert h` might well succeed, and leave you instead with goals corresponding to the places where `h` and the goal did not quite match up.

Examples

1) Here the hypothesis `h` is really close to the goal; the only difference is that one has a `d` and the other has an `e`. If you already have a proof `hde : d = e` then you can `rw hde at h` and then `exact h` will close the goal. But if you don't have this proof to hand and would like Lean to reduce the goal to `d = e` then `convert` is exactly the tactic you want.

```
import Mathlib.Tactic

example (a b c d e : ℝ) (f : ℝ → ℝ) (h : f (a + b) + f (c + d) = 37) :
    f (a + b) + f (c + e) = 37 := by
  convert h
  -- ⊢ e = d
  sorry
```

2) Sometimes *convert* can go too far. For example consider the following (provable) goal:

```
a b c d e : ℝ
f : ℝ → ℝ
h : f (a + b) + f (c + d) = 37
⊢ f (a + b) + f (d + c) = 37
```

If you try `convert h` you will be left with two goals `⊢ d = c` and `⊢ c = d` (and these goals are not provable from what we have). You can probably guess what happened; `convert` was too eager. You can "tone `convert` down" with `convert h using 2` or some other appropriate small number, to tell it when to stop converting. Experiment with the number to get it right. Here is an example.

```
import Mathlib.Tactic

example (a b c d : ℝ) (f : ℝ → ℝ) (h : f (a + b) + f (c + d) = 37) :
    f (a + b) + f (d + c) = 37 := by
  convert h using 3 -- `using 4` gives unprovable goals
  -- ⊢ d + c = c + d
  ring
```

exact

Summary

If your goal is `⊢ P` then `exact h` will close the goal if `h : P` has type `P`.

Examples

1) If the local context is

```
h : P
⊢ P
```

then the tactic `exact h` will close the goal.

2) `exact` works up to *definitional equality*. So for example, if the local context is

```
h : ¬ P
⊢ P → False
```

then `exact h` will work, because the type of `h` is definitionally equal to the goal.

Further notes

A common mistake amongst beginners is trying `exact P` to close a goal of type `P`. The goal is a type, but the `exact` tactic takes a term of that type, not the type itself. Remember `: P` is the *statement* of the problem. To solve the goal you need to supply the *proof*.

exfalso

Summary

The `exfalso` tactic changes your goal to `False`. Why might you want to do that? Usually because at this point you can deduce a contradiction from your hypotheses (for example because you are in the middle of a proof by contradiction).

Examples

If your tactic state is like this:

```
hP : P
h : P → False
⊢ Q
```

then this might initially look problematic, because we don't have any facts about `Q` to hand. However, `False → Q` regardless of whether `Q` is true or false, so and `hP` and `h` between them are enough to prove `False`. So you can solve the goal with

```
exfalso -- goal now `False`
apply h -- goal now `P`
exact hP -- goal solved
```

Warning: Don't use this tactic unless you can deduce a contradiction from your hypotheses! If your hypotheses are not contradictory then `exfalso` will leave you with an unsolvable goal.

Details

What is actually happening here is that there's a theorem in Lean called `False.elim` which says that for all propositions `P`, `False → P`. Under the hood this tactic is just doing `apply False.elim`, but `exfalso` is a bit shorter.

ext

Summary

The `ext` tactic applies "extensionality lemmas". An extensionality lemma says "two things are the same if they are built from the same stuff". Examples: two subsets of a type are the same if they have the same elements, two subgroups of a group are the same if they have the same elements, two functions are the same if they take the same values on all inputs, two group homomorphisms are the same if they take the same values on all inputs.

Examples

1) Here we use the `ext` tactic to prove that two group homomorphisms are equal if they take the same values on all inputs.

```
import Mathlib.Tactic

example (G H : Type) [Group G] [Group H] (φ ψ : G →* H)
    (h : ∀ a, φ a = ψ a) : φ = ψ := by
  -- ⊢ φ = ψ
  ext g
  -- ⊢ φ g = ψ g
  apply h -- goals accomplished
```

2) Here we use it to prove that two subgroups of a group are equal if they contain the same elements.

```
import Mathlib.Tactic

example (G : Type) [Group G] (K L : Subgroup G)
    (h : ∀ a, a ∈ K ↔ a ∈ L) : K = L := by
  -- ⊢ K = L
  ext g
  -- ⊢ g ∈ K ↔ g ∈ L
  apply h
  done
```

Details

What the `ext` tactic does is it tries to `apply` lemmas which are tagged with the *@[ext]* attribute. For example if you try `#check Subgroup.ext` you can see that it's exactly the theorem that if `∀ (x : G), x ∈ H ↔ x ∈ K` then `H = K`. This theorem was tagged with `@[ext]` when it was defined, which is why the `ext` tactic makes progress on the goal.

Further notes

Sometimes `ext` applies more lemmas than you want it to do. In this case you can use the less aggressive tactic `ext1`, which only applies one lemma.

have

Summary

The `have` tactic lets you introduce new hypotheses into the system.

Examples

1) If you have hypotheses

```
hPQ : P → Q
hP : P
```

then from these hypotheses you know that you can prove `Q`. If your *goal* is `Q` then you can just `apply hPQ` then `exact hP`, or `exact hPQ hP`. But if you need `Q` for some other reason (e.g. perhaps `Q` is of the form `x = y` and you want to rewrite it) then one way of making it is by writing `have hQ : Q := by`. This creates a *new goal* of `Q`, which you can prove with `apply hPQ`, `exact hP`, and after this you'll find that you have a new hypothesis `hQ : Q` in your tactic state.

2) If you can directly write the term of the type that you want to have, then you can do it using `have hQ : Q := ...`. For instance, in the example above you could write `have hQ : Q := hPQ hP`, because `hPQ` is a function from proofs of `P` to proofs of `Q` so you can just feed it a proof of `P`.

Further notes

The `let` and `set` tactics are related; they are however used to construct data rather than proofs.

induction

Summary

The `induction` tactic will turn a goal of the form `⊢ P n` (with `P` a predicate on naturals, and `n` a natural) into two goals `⊢ P 0` and `⊢ ∀ d, P d → P (succ d)`.

Overview

The `induction` tactic does exactly what you think it will do; it's a tactic which reduces a goal which depends on a natural to two goals, the base case (always zero, in Lean) and the step case. In computer science it is very common to see a lot of other so-called inductive types (for example `List` and `Expr`) and the `induction` tactic can get quite a lot of use; however in this course we only ever use `induction` on naturals.

Example

We show $\sum_{i=0}^{n-1} i^2 = n(n-1)(2n-1)/6$ by induction on n.

```
import Mathlib.Tactic

open BigOperators -- 𝕊 notation

open Finset -- so I can say `range n` for the finite set `{0,1,2,...,n-1}`

-- sum from i = 0 to n - 1 of i^2 is n(n-1)(2n-1)/6
-- note that I immediately coerce into rationals in the statement to get the correct␣
↪subtraction and
-- division (natural number subtraction and division are pathological functions)
example (n : ℕ) : 𝕊 i in range n, (i : ℚ) ^ 2 = (n : ℚ) * (n - 1) * (2 * n - 1) / 6␣
↪:= by
induction' n with d hd
· -- base case says that an empty sum is zero times something, and this sort of goal
  -- is perfect for the simplifier
  simp
·  -- inductive step
  rw [sum_range_succ] -- the lemma saying
                      -- `𝕊 i in range (n.succ), f i = (𝕊 i in range n, f i) + f n`
  rw [hd] -- the inductive hypothesis
  simp -- change `d.succ` into `d + 1`
  ring
```

Details

The definition of the natural numbers in Lean is this:

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

When this code is run, four new objects are created: the type `Nat`, the term `Nat.zero`, the function `Nat.succ : Nat → Nat` and the *recursor* for the naturals, a statement automatically generated from the definition and which can be described as saying that to do something for all naturals, you only have to do it for `zero` and `succ n`, and in the `succ n` case you can assume you already did it for `n`. The `induction` tactic applies this recursor and then does some tidying up. This is why you end up with goals containing `n.succ` rather than the more mathematically natural `n + 1`. In the example above I use `simp` to change `n.succ` to `n + 1` (and also to push casts as far in as possible).

intro

Summary

The `intro` tactic makes progress on goals of the form `⊢ P → Q` or `⊢ ∀ x, P x` (where `P x` is any proposition that depends on `x`). Mathematically, it says "to prove that `P` implies `Q`, we can assume that `P` is true and then prove `Q`", and also "to prove that `P x` is true for all `x`, we can let `x` be arbitrary and then prove `P x`.

Examples

`intro h` turns

```
⊢ P → Q
```

into

```
h : P
⊢ Q
```

Similarly, `intro x` turns

```
⊢ ∀ (a : ℕ), a + a = 2 * a
```

into

```
x : ℕ
⊢ x + x = 2 * x
```

Details

`intro` works when the goal is what is known as a "Pi type". This is a fancy computer science word for some kind of generalised function type. One Pi type goal which mathematicians often see is an implication type of the form `P → Q` where `P` and `Q` are propositions. The other common Pi type goal is a "for all" goal of the form `∀ a, a + a = 2 * a`. Read more about Pi types in the Part B explanation of Lean's three different types.

Variants of `intro` include the *intros* tactic (which enables you to `intro` several variables at once) and the *rintro* tactic (which enables you to do case splits on variables whilst introducing them).

intros

Summary

The `intros` tactic is a variant of the `intro` tactic, which can be used if the user wants to use `intro` several times. The tactic `intros a b c` is equivalent to `intro a, intro b, intro c`.

Examples

`intros h` turns

```
⊢ P → Q
```

into

```
h : P
⊢ Q
```

just as `intro h` would do. However

`intros hP hQ hR` turns

```
⊢ P → Q → R → S
```

into

```
hP : P
hQ : Q
hR : R
⊢ S
```

Similarly, `intros x y z` turns

```
⊢ ∀ (a b c : ℕ), a + b + c = c + b + a
```

into

```
x y z : ℕ
⊢ x + y + z = z + y + x
```

Further notes

A variant of `intros` is *rintro*, which also enables multiple intros at once, as well as allowing the user do case splits on variables.

left

Summary

There are two ways to prove `⊢ P ∨ Q`; you can either prove `P` or you can prove `Q`. If you want to prove `P` then use the `left` tactic, which changes `⊢ P ∨ Q` to `⊢ P`.

Details

It's a theorem in Lean that `P → P ∨ Q`. The `left` tactic applies this theorem, thus reducing a goal of the form `⊢ P ∨ Q` to the goal `⊢ P`.

Further notes

See also `right`. More generally, if your goal is an inductive type with two constructors, `left` applies the first constructor, and `right` applies the second one.

linarith

Summary

The `linarith` tactic solves certain kinds of linear equalities and inequalities in concrete types such as the naturals or reals. Unlike the `ring` tactic, `linarith` uses hypotheses in the tactic state. It's very handy for epsilon-delta proofs.

Examples

1) If your local context looks like this:

```
a b c d : ℝ
h1 : a < b
h2 : b ≤ c
h3 : c = d
⊢ a + a < d + b
```

then you would like to say that the goal "obviously" follows from the conclusions. but actually proving it from first principles is a little bit messy, and will involve knowing or discovering the names of lemmas such as `lt_of_lt_of_le` and `add_lt_add`. Fortunately, you don't have to do this: `linarith` closes this goal immediately. Note that in contrast to `ring`, `linarith` does have access to the hypotheses in your local context.

2) If you have a goal of the form `|x| < ε` then `linarith` might not be much help yet, because it doesn't know about absolute values. So you could `rw abs_lt` and get a goal of the form `-ε < x ∧ x < ε`. Well, `linarith` still won't be of any help, because it doesn't know about goals with ∧ in either! However after you try `constructor`, perhaps `linarith` will be able to help you.

```
import Mathlib.Tactic

example (x ε : ℝ) (hε : 0 < ε) (h1 : x < ε / 2) (h2 : -x < ε / 2) : |x| < ε := by
rw [abs_lt] -- `⊢ -ε < x ∧ x < ε`
constructor <;> -- <;> means "do next tactic on all the goals this tactic produces"
linarith -- solves both goals
```

nlinarith

Summary

The `nlinarith` tactic is a stronger version of *linarith* which can deal with some nonlinear goals (for example it can solve `0 ≤ x^2` if `x : ℝ`).

Just as for *linarith*, you can feed extra information into the mix (for example, explicit proofs `h1` and `h2` that various things are non-negative or other relevant information can be inserted into the algorithm with `nlinarith [h1, h2]`).

Read the documentation of the `linarith` tactic to see the sorts of goals which this tactic can solve. In brief, it uses equalities and inequalities in the hypotheses to try and prove the goal (which can also be an inequality or an equality).

norm_num

Summary

The `norm_num` tactic solves equalities and inequalities which involve only normalised numerical expressions. It doesn't deal with variables, but it will prove things like `2 + 2 = 4`, `2 ≤ 3`, `3 < 15/2` and `3 ≠ 4`. We emphasize that this is a tool which closes goals involving *numbers*.

Note

I often see students misusing this tactic. If your goal has a variable like *x* in, then *norm_num* is not the tactic you should be using. Note that *norm_num* calls *simp* under the hood so it might solve a goal like *(x : ℝ) + 0 = x* anyway – but this is not the correct usage of *norm_num* – all that's happening here is that it is solving the goal using *simp* but taking longer to do so.

Examples

(with `import Mathlib.Tactic` imported)

```
example : (1 : ℝ) + 1 = 2 := by
  norm_num

example : (1 : ℚ) + 1 ≤ 3 := by
  norm_num

example : (1 : ℤ) + 1 < 4 := by
  norm_num

example : (1 : ℂ) + 1 ≠ 5 := by
  norm_num
```

(continues on next page)

(continued from previous page)

```
example : (1 : ℕ) + 1 ≠ 6 := by
  norm_num

example : (3.141 : ℝ) + 2.718 = 5.859 := by
  norm_num
```

`norm_num` also knows about a few other things; for example it seems to know about the absolute value on the real numbers.

```
example : |(3 : ℝ) - 7| = 4 := by
  norm_num
```

nth_rw

Summary

If `h : a = b` then `rw h` turns *all* `a` s in the goal to `b` s. If you only want to turn one of the `a` s into a `b`, use `nth_rw`. For example `nth_rw 2 [h]` will change only the second `a` into `b`.

Examples

1) Faced with

```
h : x = y
⊢ x * x = a
```

the tactic `nth_rw 1 [h]` will turn the goal into `⊢ y * x = a` and `nth_rw 2 [h]` will turn it into `⊢ x * y = a`. Compare with `rw [h]` which will turn it into `⊢ y * y = a`.

2) `nth_rw` works on hypotheses too. If `h : x = y` is a hypothesis and `h2 : x * x = a` then `nth_rw 1 [h] at h2` will change `h2` to `y * x = a`.

3) Just like `rw`, `nth_rw` accepts `← h` if you want to change the right hand side of `h` to the left hand side.

obtain

Summary

The `obtain` tactic can be used to do `have` and `cases` all in one go. It uses the same `⟨x, y⟩` / `(x | y)` syntax as the *rcases* tactic.

Example

If you have a hypothesis `h : ∀ ε > 0, ∃ (N : ℕ), (1 : ℝ) / (N + 1) < ε`, then you could specialize `h` to the case `ε = 0.01` with `have h2 := h 0.01 (by norm_num)` (or `specialize h 0.01 (by norm_num)` if you're prepared to change `h`) and then you can get to `N` and the hypothesis `hN : 1 / (N + 1) < ε` with `cases' h2 with N hN` or `rcases h2 with ⟨N, hN⟩`. But you can do both steps in one go with `obtain ⟨N, hN⟩ := h 0.01 (by norm_num)`.

To make your code more readable you can explicitly mention the type of `⟨N, hN⟩`. In the above example you could write `obtain ⟨N, hN⟩ : ∃ (N : ℕ), (1 : ℝ) / (N + 1) < 0.01 := h 0.01 (by norm_num)`, meaning that the reader can instantly see what the type of `hN` is.

Notes

Note that, like `rcases` and `rintro`, `obtain` works up to *definitional equality*.

As with `rcases` and `rintro`, there is a `rfl` trick, where you can eliminate a variable using `rfl` instead of a hypothesis name.

positivity

Summary

This a tactic which solves certain goals of the form `0 ≤ x`, `0 < x` and `x ≠ 0`.

Details

`positivity` knows certain "tricks" (for example it knows that if `a>0` and `b>0` then `a+b>0`), and tries to solve the goal using only these tricks. More generally it takes the goal apart, tries to prove the relevant analogue of the goal for all the pieces, and then tries to find lemmas which glue everything together.

Examples

1) `positivity` will solve this:

```
x y : ℝ
hx : x > 37
hy : y > 42
⊢ x * y > 0
```

2) It will also solve this:

```
x y : ℝ
hx : x ≠ 0
hy : y ≠ 0
⊢ 2 * x * y ≠ 0
```

3) And this:

```
x y : ℝ
hy : y ≥ 0
⊢ 37 * x ^ 2 * y ≥ 0
```

rcases

Summary

The `rcases` tactic can be used to do multiple `cases'` tactics all in one line. It can also be used to do certain variable substitutions with a `rfl` trick.

Examples

1) If `ε : ℝ` is in your tactic state, and also a hypothesis `h : ∃ δ > 0, δ^2 = ε` then you can take `h` apart with the `cases'` tactic. For example you can do this:

```
cases' h with δ h1 -- h1: δ > 0 ∧ δ ^ 2 = ε
cases' h1 with hδ h2
```

which will leave you with the state

```
ε δ : ℝ
hδ : δ > 0
h2 : δ ^ 2 = ε
```

However, you can get there in one line with `rcases h with ⟨δ, hδ, h2⟩`.

2) In fact you can do a little better. The hypothesis `h2` can be used as a *definition* of ε, or a *formula* for ε, and the `rcases` tactic has an extra trick which enables you to completely remove ε from the tactic state, replacing it everywhere with `δ ^ 2`. Instead of calling the hypothesis `h2` you can instead type `rfl`. This has the effect of rewriting `← h2` everywhere and thus replacing all the ε s with `δ ^ 2`. If your tactic state contains this:

```
ε : ℝ
h : ∃ δ > 0, δ ^ 2 = ε
⊢ ε < 0.1
```

then `rcases h with ⟨δ, hδ, rfl⟩` turns the state into

```
δ : ℝ
hδ : δ > 0
⊢ δ ^ 2 < 0.1
```

Here ε has vanished, and all of the other occurrences of ε in the tactic state are now replaced with `δ ^ 2`.

3) If `h : P ∧ Q ∧ R` then `rcases h with ⟨hP, hQ, hR⟩` directly decomposes `h` into `hP : P`, `hQ : Q` and `hR : R`. Again this would take two moves with `cases'`.

4) If `h : P ∨ Q ∨ R` then `rcases h with (hP | hQ | hR)` will replace the goal with three goals, one containing `hP : P`, one containing `hQ : Q` and the other `hR : R`. Again `cases'` would take two steps to do this. Note: the syntax is different for $\wedge$ and $\vee$ because doing cases on an $\vee$ turns one goal into two (because the inductive type `Or` has two constructors).

Notes

`rcases` works up to *definitional equality*.

Other tactics which use the ⟨`hP, hQ, hR`⟩ / `(hP | hQ | hR)` syntax are the *rintro* tactic (`intro` + `rcases`) and the *obtain* tactic (`have` + `rcases`).

refine

Summary

The `refine` tactic is "`exact` with holes". You can use an incomplete term containing one or more underscores `?_` and Lean will give you these terms as new goals.

Examples

1) Faced with (amongst other things)

```
hQ : Q
⊢ P ∧ Q ∧ R
```

you can see that we already have a proof of `Q`, but we might still need to do some work to prove `P` and `R`. The tactic `refine ⟨?_, hQ, ?_⟩` replaces this goal with two new goals `⊢ P` and `⊢ R`.

2) As well as being a generalization of the `exact` tactic, `refine` is also a generalization of the `apply` tactic. If your tactic state is

```
h : P → Q
⊢ Q
```

then you can change the goal to `⊢ P` with `refine h ?_`.

3) `refine ?_` does nothing at all; it leaves the goal unchanged.

4) Faced with `⊢ ∃ (n : ℕ), n ^ 4 = 16`, the tactic `refine ⟨2, ?_⟩` turns the goal into `⊢ 2 ^ 4 = 16`, so it does the same as `use 2`. In fact here, because `⊢ 2 ^ 4 = 16` can be solved with `norm_num`, the entire goal can be closed with `exact ⟨2, by norm_num⟩`.

5) If your tactic state looks (in part) like this:

```
f : ℝ → ℝ
x y ε : ℝ
hε : 0 < ε
⊢ ∃ δ > 0, |f y - f x| < δ
```

then `refine ⟨ε^2, by positivity, ?_⟩` changes the goal to `⊢ |f y - f x| < ε ^ 2`. Here we use the *positivity* tactic to prove `ε^2 > 0` from the hypothesis `0 < ε`.

Further notes

`refine` works up to definitional equality.

rfl

Summary

The `rfl` tactic proves goals of the form `⊢ x = y` where `x` and `y` are *definitionally equal*. It also proves goals of the form `P ↔ Q` if `P` and `Q` are definitionally equal.

Examples

1) `rfl` will prove `⊢ x = x`.
2) `rfl` will prove `⊢ P ↔ P`.
3) `rfl` will prove `⊢ ¬ P ↔ (P → False)` because even though the two sides of the iff are not syntactically equal, they are definitionally equal.
4) `rfl` will prove `⊢ 2 + 2 = 4` if the 2s and the 4 are natural numbers (i.e. have type ℕ). This is because both sides are definitionally equal to `succ(succ(succ(succ(0))))`. It will not prove `⊢ 2 + 2 = 4` if the `2``s and the ``4` are real numbers however; one would have to use a more powerful tactic such as `norm_num` to do this.

Further notes

Checking definitional equality can be extremely difficult. In fact it is a theorem of logic that checking definitional equality in Lean is undecidable in general. Of course this doesn't necessarily mean that it's hard in practice; in the examples we will see in this course `rfl` should work fine when it is supposed to work. There is a pathological example in Lean's reference manual of three terms `A`, `B` and `C` where `` `⊢ A = B `` and `` `⊢ B = C `` can both be proved by `rfl`, but `rfl` fails to prove `⊢ A = C` (even though they are definitionally equal). Such pathological examples do not show up in practice when doing the kind of mathematics that we're doing in this course though.

If you're doing harder mathematics in Lean then you can be faced with goals which look simple but which under the hood are extremely long and complex terms; sometimes `rfl` can take several seconds (or even longer) to succeed in these cases. In this course I doubt that we will be seeing such terrifying terms.

right

Summary

There are two ways to prove `⊢ P ∨ Q`; you can either prove `P` or you can prove `Q`. If you want to change the goal to `Q` then use the `right` tactic.

Details

It's a theorem in Lean that `Q → P ∨ Q`, and the `right` tactic applies this theorem.

Further notes

See also `left`. More generally, if your goal is an inductive type with two constructors, `left` applies the first constructor, and `right` applies the second one.

ring

Summary

The `ring` tactic proves identities in commutative rings such as `(x+y)^2=x^2+2*x*y+y^2`. It works on concrete rings such as ℝ and abstract rings, and will also prove some results in "semirings" such as ℕ (which isn't a ring because it doesn't have additive inverses).

Note that `ring` does not and cannot look at hypotheses. See the examples for various ways of working around this.

Examples

1) A basic example is

```
```

example (x y
[ℝ)][x^3-y^3=(x-y)*(x^2+x*y+y^2) := by] ring

2) Note that `ring` cannot use hypotheses – the goal has to be an identity. For example faced with

```
x y : ℝ
h : x = y ^ 2
⊢ x ^ 2 = y ^ 4
```

the `ring` tactic will not close the goal, because it does not know about `h`. The way to solve this goal is `rw [h]` and *then* `ring`.

3) Sometimes you are in a situation where you cannot `rw` the hypothesis you want to use. For example if the tactic state is

```
x y : ℝ
h : x ^ 2 = y ^ 2
⊢ x ^ 4 = y ^ 4
```

then `rw h` will fail (of course we know that `x^4=(x^2)^2`, but `rw` works up to syntactic equality and it cannot see an `x^2` in the goal). In this situation we can use `ring` to prove an intermediate result and then rewrite our way out of trouble. For example this goal can be solved in the following way.

```
```

example (x y
[ℝ) (h][x ^ 2 = y ^ 2)][x ^ 4 = y ^ 4 := by] rw [show x ^ 4 = (x ^ 2) ^ 2 by ring] – prove x^4=(x^2)^2, rewrite with it, forget it – goal now ⊢ *(x ^ 2) ^ 2 = y ^ 4* rw [h] – goal now ⊢ *(y ^ 2) ^ 2 = y ^ 4* ring

It can also be solved in the following way:

```
example (x y : ℝ) (h : x ^ 2 = y ^ 2) : x ^ 4 = y ^ 4 := by
convert_to (x ^ 2) ^ 2 = y ^ 4
-- creates new goal ⊢ x ^ 4 = (x ^ 2) ^ 2
· ring -- solves this new goal
-- goal now `⊢ (x ^ 2) ^ 2 = y ^ 4`
rw [h]
-- goal now `⊢ (y ^ 2) ^ 2 = y ^ 4`
ring
```

Further notes

`ring` is a "finishing tactic"; this means that it should only be used to close goals. If `ring` does not close a goal it will issue a warning that you should use the related tactic `ring_nf`.

The algorithm used in the `ring` tactic is based on the 2005 paper "Proving Equalities in a Commutative Ring Done Right in Coq" by Benjamin Grégoire and Assia Mahboubi.

rintro

Summary

The `rintro` tactic can be used to do multiple `intro` and `cases'` tactics all in one line.

Examples

1) Faced with the goal

```
⊢ P → (Q ∧ R) → S
```

one could use the following tactics

```
intro hP
intro h
cases' h with hQ hR
```

to get the tactic state

```
hP : P
hQ : Q
hR : R
⊢ S
```

However, one can get there in one line with

```
rintro hP ⟨hQ, hR⟩
```

(the pointy brackets ⟨⟩ can be obtained by typing \< and \> in VS Code.)

2) Faced with a goal of the form

```
⊢ P ∨ Q → R
```

the tactic `rintro (hP | hQ)` will do the same as `intro h` then `cases' h with hP hQ`. In particular, after the application of the tactic there will be two goals, one with hypothesis `hP : P` and the other with hypothesis `hQ : Q`. Note the round brackets for "or" goals.

3) There is a `rfl` easter egg with the `rintro` tactic. If the goal is

```
⊢ a = 2 * b → 2 * a = 4 * b
```

then you can solve the goal with

```
intro h
rw [h]
ring
```

but there's a trick: `rintro rfl` will, instead of naming the hypothesis `a = 2 * b` and putting it in the tactic state, instead *define* `a` to be `2 * b`, leaving the goal as

```
⊢ 2 * (2 * b) = 4 * b
```

so now `ring` solves the goal immediately.

Details

Note that `rintro` works up to *definitional equality*, like `intro`, so for example if the goal is `⊢ ¬P` then `rintro hP` works fine (because `¬P` is definitionally equal to `P → False`), leaving the tactic state as

```
hP : P
⊢ False
```

Further notes

The syntax for `rintro` with pointy and round brackets is the same as the syntax for `rcases`, where more examples are given.

rw

Summary

The `rw` or `rewrite` tactic is a "substitute in" tactic. If `h : x = y` is a hypothesis then `rw [h]` changes all of the `x` s in the goal to `y` s. `rw` also works with iff statements: if `h : P ↔ Q` then `rw [h]` will replace all the `P` s in the goal with `Q` s.

Notes

`rw` works up to syntactic equality, i.e. if `h : x = y` then `rw [h]` will only work if the goal contains something which is literally the same as `x`.

A common mistake I see is people trying `rw [h]` when `h` is *not* an equality, or an iff statement; most commonly people try it when `h` has type `P → Q`. This won't work; you use the *apply* tactic in this situation.

Examples

1) Faced with

```
h : x = y
⊢ x ^ 2 = 1369
```

the tactic `rw [h]` will turn the goal into `y ^ 2 = 1369`.

2) `rw` works with ↔ statements as well. Faced with

```
h : P ↔ Q
⊢ P ∧ R
```

the tactic `rw [h]` will turn the goal into `⊢ Q ∧ R`.

3) `rw` can also be used to rewrite in hypotheses. For example given `h : x = y` and `h2 : x ^ 2 = 289`, the tactic `rw [h] at h2` will turn `h2` into `h2 : y ^ 2 = 289`. You can rewrite in multiple places at once – for example `rw [h] at h1 h2 h3` will change all occurrences of the left hand side of `h` into the right hand side, in all of `h1`, `h2` and `h3`. If you want to rewrite in a hypothesis and a goal at the same time, try `rw [h] at h1 ⊢` (type the turnstile ⊢ symbol with `\|-`).

4) `rw` doesn't just eat hypotheses – the theorem `zero_add` says `∀ x, 0 + x = x`, so if you have a goal `⊢ 0 + t = 37` then `rw [zero_add]` will change it to `t = 37`. Note that `rw` is smart enough to fill in the value of `x` for you.

5) If you want to replace the right hand side of a hypothesis `h` with the left hand side, then `rw [← h]` will do it. Type ← with `\l`, noting that `l` is a small letter `L` for left, and not a number `1`.

6) You can chain rewrites in one tactic. Equivalent to `rw [h1], rw [h2], rw [h3]` is `rw [h1, h2, h3]`.

7) `rw` will match on the first thing it finds. For example, `add_comm` is the theorem that `∀ x y, x + y = y + x`. If the goal is `⊢ a + b + c = 0` then `rw add_comm` will change it to `⊢ c + (a + b) = 0`, because if we strip away the notation then `a + b + c = add (add a b) c`, which gets changed to `add c (add a b)`. If you wanted to switch `a` and `b` then you can tell Lean to do this explicitly with `rw add_comm a b`.

Further notes

1) `rw` tries a weak version of `rfl` when it's finished. Beginners can find this confusing; indeed I disabled this in the natural number game because users found it confusing. As an example, if your tactic state is

```
h : P ↔ Q
⊢ P ∧ R ↔ Q ∧ R
```

then `rw [h]` will close the goal, because after the rewrite the goal becomes `⊢ Q ∧ R ↔ Q ∧ R` and `rfl` closes this goal.

2) A variant of `rw` is `rwa`, which is `rw` followed by the `assumption` tactic. For example if your tactic state is

```
h1 : P ↔ Q
h2 : Q ↔ R
⊢ P ↔ R
```

then `rwa [h1]` closes the goal, because it turns the goal into `Q ↔ R` which is one of our assumptions.

3) If `h : x = y` and your goal is `⊢ x * 2 + z = x`, then `rw [h]` will turn *both* `x``s into ``y``s. If you only want to turn one of the ``x``s into a ``y` then you can use `nth_rw 1 [h]` (for the first one) or `nth_rw 2 [h]` (for the second one).

4) `rw` works up to *syntactic equality*. This means that if `h : (P → False) ↔ Q` and the goal is `⊢ ¬P` then `rw [h]` will fail, even though `P → False` and `¬P` are definitionally equal. The left hand side has to match *on the nose*.

5) `rw` does not work under binders. Examples of binders are ∀ and ∃. For example, if your goal is `⊢ ∀ (x : ℕ), x + 0 = 0 + x` then `rw [add_zero]` will not work. In situations like this you can either do `intro x` first, or you can use the more powerful `simp_rw` tactic; `simp_rw [add_zero]` works and changes the goal to `∀ (x : ℕ), x = 0 + x`.

simp

Summary

There are many lemmas of the form `x = y` or `P ↔ Q` in `mathlib` which are "tagged" with the `@[simp]` tag. Note that these kinds of lemmas are ones for which the `rw` tactic can be used. A lemma tagged with `@[simp]` is called a "simp lemma".

When Lean's simplifier `simp` is run, it tries to find simp lemmas for which the left hand side of the lemma is in the goal. It then rewrites the lemma and continues. Ultimately what happens is that the goal ends up simplified, and, ideally, solved.

Overview

There are hundreds of lemmas in `mathlib` of the form `x + 0 = x` or `x * 0 = 0`, where one side is manifestly simpler than the other (in the sense that a mathematician would instinctively replace the more complex side by the simpler side if they were trying to prove a theorem). In Lean, such a lemma might be tagged with the `@[simp]` tag. A tag on a lemma or definition does nothing mathematical; it is just a flag for certain tactics. The convention for `@[simp]` lemmas in Lean is that the left hand side of such a lemma should be the more complicated side. For example Lean has

```
@[simp] add_zero (x) : x + 0 = x := ...
@[simp] zero_add (x) : 0 + x = x := ...
```

(proofs omitted). Because these lemmas are tagged with `@[simp]`, the following proof works:

```
example (x : ℝ) : 0 + 0 + x + 0 + 0 = (x + (0 + 0)) := by
  simp
```

If you want to know which lemmas `simp` used, you can instead use `simp?`, which gives an output listing the theorems which `simp` rewrote to make the progress which it made.

Examples

1) If you are doing a proof by induction, then `simp` will often deal with the base case, because it knows lots of ways to simplify goals with a `0` in. Here is `simp` being used to prove that $\sum_{i=0}^{n-1} i = n(n-1)/2$:

```
import Mathlib.Tactic

open BigOperators Finset -- ∑ notation and access to `Finset.range`

example (n : ℕ) :
   ∑ i in range n, (i : ℝ) = n * (n - 1) / 2 := by
induction' n with d hd
·  -- base case: sum over empty type is 0 * (0 - 1) / 2
   simp
· -- inductive step
   rw [sum_range_succ, hd]
   simp -- tidies up and reduces the goal to
        -- ⊢ ↑d * (↑d - 1) / 2 + ↑d = (↑d + 1) * ↑d / 2
   ring -- a more appropriate tactic to finish the job
```

Details

Note that `simp`, like `rw`, works up to *syntactic equality*. In other words, if your goal mentions `x` and there is a simp lemma `h : x' = y` where `x'` is definitionally, but not syntactically, equal to `x`, then `simp` will not do the rewrite; this is because `rw [h]` fails.

You can use `simp` on goals too: `simp at h` will run the simplifier on `h`.

There's quite a lot to say about the `simp` tactic. More details of how to use it can be found in Theorem Proving In Lean, in the section "using the simplifier" here .

Further notes

The related tactic `simp?` attempts to figure out exactly which lemmas `simp` used.

simpa

Summary

If you have a goal and a hypothesis `h`, and if Lean's simplifier `simp`, if run on both of them, will turn them into the same thing, then you could solve the goal in three lines with `simp`, `simp at h`, `exact h`, or even in two lines with `simp at *`, `exact h`. But you could also solve it in one line with `simpa using h`. In fact `h` doesn't need to be a hypothesis, it can be any proof you like (e.g. a proof you made using some lemmas and some hypotheses).

Examples

1)

```
import Mathlib.Tactic

example (x y z : ℝ) (h : x = y + z + 0) : x * 1 = y + z := by
  -- Lean's simplifier knows that a + 0 = 0 and a * 1 = a
  simpa using h
```

Here the simplifier can do some work on both the hypothesis `h` (removing the `+ 0`) and on the goal (removing `* 1`). Once this work is done, `h` and the goal become equal.

2) Like with `simp`, you can also feed `simpa` a list of extra lemmas for the simplifier to use. For the example below `simpa using h` won't work because the simplifier doesn't know `hxy` (the simplifier doesn't use hypotheses in the tactic state).

```
import Mathlib.Tactic

example (x y z : ℕ) (hxy : x = y) (h : z = y + 0) : z = x * 1 := by
simpa [hxy] using h
```

Further notes

Easter egg: If your hypothesis is called `this` then you don't have to write `using this` at all, you can just write `simpa`.

specialize

Summary

The `specialize` tactic can be used specialize hypotheses which are *function types*, by giving some or all of the inputs to the function. Its two main uses are with hypotheses of type `P → Q` and of type `∀ x, ...` (that is, hypotheses where you might use `intro` if their type was the goal).

Examples

1) If you have a hypothesis `h : P → Q` and another hypothesis `hP : P` (that is, a proof of `P`) then `specialize h hP` will change the type of `h` to `h : Q`.

Note: `h` is a function from proofs of `P` to proofs of `Q` so if you just want to quickly access a proof of `Q` for use elsewhere, then you don't have to use `specialize` at all, you can make a proof of `Q` with the term `h (hP)` or, as the functional programmers would prefer us to write, `h hP`.

2) If you have a hypothesis `h : ∀ x, f x = 37` saying that the function `f (x)` is a constant function with value 37, then `specialize h 10` will change `h` to `h : f 10 = 37`.

Note that `h` has now become a weaker statement; you have *lost* the hypothesis that `∀ x, f x = 37` after this application of the `specialize` tactic. If you want to keep it around, you could make a copy by running `have h2 := h` beforehand.

3) You can specialize more than one variable at once. For example if you have a hypothesis `h : ∀ (x y : ℕ), ...` then `specialize h 37 42` sets `x = 37` and `y = 42` in `h`.

4) If you have a hypothesis `h : ∀ ε > 0, ∃ δ > 0, ...`, it turns out that internally this is `∀ (ε : ℝ), ε > 0 → ...`. If you also have variables `ε : ℝ` and `hε : ε > 0` then you can do `specialize h ε hε` in which case `h` will change to `∃ δ > 0, ...`.

5) If again you have a hypothesis `h : ∀ ε > 0, ∃ δ > 0, ...` but you would like to use `h` in the case where `ε = 37` then you can `specialize h 37` and this will change `h` to `h : 37 > 0 → (∃ δ...`. Now obviously `37 > 0` but `h` still wants a proof of this as its next input. You could make this proof in a couple of ways. For example (assuming that `37` is the real number `37 : ℝ`, which it probably is if you're doing epsilon-delta arguments) this would work:

```
have h37 : (37 : ℝ) > 0 -- two goals now
· positivity
specialize h h37
```

However you could just drop the proof in directly:

```
specialize h (by positivity)
```

or

```
specialize h (by norm_num)
```

or

```
specialize h (by linarith)
```

In fact going back to the original hypothesis `h : ∀ ε > 0, ∃ δ > 0, ...`, and remembering that Lean actually stores this as `h : ∀ (ε : ℝ), ε > 0 → ∃ δ ...`, you could specialize to `ε = 37` in one line with `specialize h 37 (by positivity)`.

Further notes

You don't always have to `specialize`. For example if you have `h : ∀ ε > 0, ∃ N, some_fact` where `some_fact` is some proposition which depends on `ε` and `N`, and you also have `ε : ℝ` and `hε : ε > 0` in your tactic state, you can just get straight to `N` and the fact with `cases' h ε hε with N hN` or `obtain ⟨N, hN⟩ : some_fact := h ε hε`.

triv

Summary

The `triv` tactic proves `⊢ True`.

It also proves goals of the form `x = x` and more generally of the form `x = y` when `x` and `y` are definitionally equal, but traditionally people use `rfl` to do that.

Examples

If your goal is

```
⊢ True
```

then it's pretty triv, so try `triv`.

Details

Note that `True` here is the true proposition. If you know a proof in your head that the goal is true, that's not good enough. If your goal is `⊢ P` and you can tell that `P` is true (e.g. because you can deduce it from the hypotheses), `triv` won't work; `triv` only works when the goal is actually definitionally equal to `True`.

Further notes

The `constructor` tactic also proves a `True` goal, although you would have to learn a bit about inductive types to understand why.

use

Summary

The `use` tactic can be used to make progress with ∃ goals; if the goal is to show that there exists a number `n` with some property, then `use 37` will reduce the goal to showing that `37` has the property.

Examples

1) Faced with the goal

```
⊢ ∃ (n : ℝ), n + 37 = 42
```

progress can be made with `use 5`. Note that `use` is a tactic which can leave you with an impossible goal; `use 6` would be an example of this, where a goal which was solvable becomes unsolvable.

2) You can give `use` a list of things, if the goal is claiming the existence of more than one thing. For example

```
import Mathlib.Tactic

example : ∃ (a b : ℝ), a + b = 37 := by
use 5, 32
-- ⊢ 5 + 32 = 37
norm_num
```

Further notes

The *refine* tactic can do what `use` does; for example instead of `use 5, 32` in the above example, one can try `refine ⟨5, 32, ?_⟩`. The ?underscore means "I'll do the proof later".