Formalising Mathematics

Release 0.1

Kevin Buzzard

Mar 23, 2022

CONTENTS:

A handbook for mathematicians. Written by Kevin Buzzard, for his Formalising Mathematics course. Thanks as ever to Imperial College London.

Note: This document is currently under active development. It is being written during the period of January to March 2022 as part of an undergraduate course for mathematicians at Imperial College London. It's online even though it's currently incomplete, so that the students can access it. It will be stabilising by April 2022.

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`. Right now this document covers Lean 3; when `mathlib` has been ported to Lean 4 I will rewrite this document to cover Lean 4.

If you're new here, start at the *introduction*. The mathematics is all in Part A.

Here's a link to the repository for this course containing all the code, and here's a link to the YouTube playlist where I work through the problem sheets and discuss techniques.

Here is a pdf of this document.

CHAPTER

ONE

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 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.

1.2 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.

1.3 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.

We will be using Lean 3 this year, because right now Lean 4 is still a work in progress and in particular it does not yet have a powerful mathematics library. When Lean 4 is ready I will port this course over.

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.

1.5 How to read this document

This book comes in three parts: Part A, Part B and Part C. Part A is the mathematical meat 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.

CHAPTER

TWO

GETTING LEAN RUNNING ON YOUR COMPUTER

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 :=
begin
  apply h2,
  exact h1
end
```

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 between the `begin` and the `end` 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.

2.1 The best way: install Lean on your computer

The main advantage of this method is that, once you have it all working, Lean will be *quick*. It will start up instantly and it will run fast.

You will need to install Lean 3, and the Lean community tools. Instructions on how to get these things 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 them, you can install the Lean repository `formalising-mathematics-2022` associated with this course. Fire up your command line, navigate to the place where you want to install the repository, and type

```
leanproject get ImperialCollegeLondon/formalising-mathematics-2022
```

Then use VS Code's "open folder" open and open the `formalising-mathematics-2022` directory.

2.2 An alternative: Gitpod

You will need to set up an account in some way, but it is possible to access the course repository using Gitpod.

Right click here and "open link in new tab" to access the repository using Gitpod.

It takes longer to fire up, and I've had problems with some browsers, but I've got it working with Chrome. The disadvantages of this method: you are not able to save your work, you are not able to create your own projects, and I believe there is some time limit for the amount that you can use gitpod for free. However if you are just dabbling then this might be the solution for you.

CHAPTER

THREE

PART A: THE MATHEMATICS

The material in this course is divided into 11 parts, which correspond to the 11 weeks of the course which I am teaching. I would recommend that readers work through these parts in order. This is in contrast to the material in parts B and C, which can be read in any order.

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 →, ¬, ∧, ↔ and ∨ interact with propositions like `true` and `false`.

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-2022` Lean repository, find `section01` within the `src` directory of the repository, and try doing the problem sheets! If you need help getting started, you can read the below, or you can watch the video **TODO** make a video and link to it here.

3.1.1 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).

3.1.2 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 "goals accomplished 🎉" 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, between a `begin` and an `end`. 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.

3.2 Section 2 : An introduction to the real numbers

3.2.1 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+ε=N` because of rounding errors, so `(-N)+(N+ε)=0` but `((-N)+N)+ε=ε`.

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.

3.2.2 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.

3.2.3 Lean is not a computer

The last thing I would like to say about the real numbers in this introduction is that Lean 3 is *not designed to compute*. This might all change in Lean 4, 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` and right now you would have to do this manually by 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).

3.2.4 Working with reals

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

```
example : (2 : ℝ) + 2 = 4 :=
begin
  sorry
end
```

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 mathlib community tactic documentation .

3.3 Section 3 : An introduction to group theory

3.3.1 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 section 3 of the course repository we study groups (in sheets 1 and 2), subgroups (in sheet 3) and group homomorphisms (in sheet 4). The approach we take is the following. In sheets 1 and 2 we make some basic group theory from scratch. In sheet 3 we dump our development and go back to Lean's groups, but develop the theory of subgroups of a group from scratch. And finally in sheet 4 we dump our theory of subgroups and use Lean's subgroups, but develop the theory of group homomorphisms from scratch.

3.3.2 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.

3.3.3 What you need to know to do sheets 1 and 2

There will be a lot of `rw` involved, because we are constantly using the group axioms. Lean's simplifier `simp` is introduced in sheet 1; don't forget to experiment with it. It will do multiple rewrites in one go. As the sheet goes on, the simplifier gets smarter because we tag more and more lemmas with the `@[simp]` attribute.

Sheet 2 is perhaps mathematically tricky.

3.3.4 What you need to know to do sheet 3

Although you don't need it, the `convert` tactic can be useful at times. Otherwise it's just a case of keeping your head, and using `intro`, `apply` and `rw`.

The *ext* tactic can be used to prove that two subgroups are equal.

3.3.5 What you need to know to do sheet 4

The *ext* tactic can be used to prove that two functions are equal.

3.4 Section 4 : functions

Note to future Kevin: this needs to be moved to section 3; it's easier than groups. We want functions section 3, sets section 4, groups section 5.

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 we can write `f x` in Lean instead of `f(x)` and it works just as well. Brackets can be used – they are just optional.

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.

3.4.1 The problem sheets

For the first sheet you should be able to get away with what you already know. For the second one I introduce a method for making finite types and explain how to define functions from these finite types to other types. This is needed because in the first sheet we are proving theorems, but in the second sheet we are expected to give counterexamples.

3.5 Section 5 : Sets

Note to future Kevin: this needs to be moved to section 4, and functions to section 3, and groups to section 5.

Lean uses type theory, not set theory. Lean has sets, but they're always subsets of a type. For example, the real numbers 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 ℝ is a type, so 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. 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? 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`.

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.

3.6.1 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 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).

3.6.2 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.

3.6.3 Quotients

Here's the set-up. We have `X` and an equivalence relation ≈ on `X`. Equivalence relations are some kind of generalisation of equality, and we want a new object `Q` where ≈ 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 ≈ 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".

3.6.4 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`.

CHAPTER

FOUR

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.

4.1 Types and terms

4.1.1 Sets and their elements

In mathematics you have seen many examples of sets and their elements. For example the real numbers 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 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 set theory.

4.1.2 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.

4.1.3 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`.

4.1.4 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, otberwise 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.

4.1.5 `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.

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 `refl` care about. Finally, there is propositional equality; this is the "usual" kind of equality as understood by mathematicians.

4.2.1 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.

4.2.2 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.

4.2.3 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 `refl` work up to definitional equality. For example, the following proof works:

```
example (x : ℕ) : x + 0 = x :=
begin
  refl
end
```

which is perhaps not what you would expect if you have played the natural number game; I explicitly withhold this knowledge from the player (and occasionally get “bug reports” when people discover it). However the following does not work:

```
example (x : ℕ) : 0 + x = x :=
begin
  refl -- failed to unify 0 + x = x with ?m_2 = ?m_2
end
```

Similarly, this code works:

```
example (x y : ℕ) (h : x + 0 = y) : x = y :=
begin
  exact h,
end
```

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 :=
begin
  intro h,
  /-
  tactic state now

  P : Prop
  h : P
  ⊢ false
  -/
  sorry,
end
```

(although the goal is of course not provable).

4.2.4 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.

4.2.5 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.

4.3 The three kinds of types

This is some background on Lean's type theory.

4.3.1 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 11.4).]

[A technical footnote about universes. if you look in the source code of `mathlib` **TODO** add web link to github) 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 *eveyrthing* 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.]

4.3.2 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 λ. In maths it's quite common for λ to denote a real number, but λ 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. 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 := λ x, x^2+3
```

If you have understood the "mapsto" symbol ↦ in mathematics you might have seen this function defined as `x ↦ x^2+3`. Computer scientists use "prefix notation" for this construction rather than "infix notation", i.e. they put the symbol indicating that we're making a function *before* the input and output, instead of *between* the input and output.

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 `Π (i : I), Y i`. 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 `Π (i : 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.

4.3.3 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 λ 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.

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
| zero : nat
| succ : nat → nat
```

(Peano observed that these two construtions were enough to define all natural numbers) and then we set up the notation ℕ for `nat` immediately afterwards. 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 λ, for example `λ (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
| 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:

```
variables (P Q : Prop)

inductive and : Prop
| intro (hp : P) (hq : Q) : and

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

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 `Goals accomplished`; 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 3 or so.

4.3.4 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 : 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.

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 tactic` or some other import which imports group theory) then we get the following output:

```
mul_assoc : ∀ {G : Type u_1} [_inst_1 : 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 `_inst_1` 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 @mul_one_class.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.

4.4.1 `()` 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`.

4.4.2 `{}` 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:

```
∀ {G : Type} [_inst_1 : 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`. 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 those 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 `{}`.

4.4.3 `[]` 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`. You can see its type with `#check @add_comm`. Its type is this:

```
add_comm : ∀ {G : Type} [_inst_1 : add_comm_semigroup G] (a b : G), a + b = b + a
```

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

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 data.real.basic

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_1`, whose type is `add_comm_semigroup ℝ`; you can think of it as "a proof that the reals are an additive commutative semigroup" but it's actually more than just a proof – it's all the data of the addition and identity element as well; the functions and constants as well as the proofs. Where does Lean get this variable `_inst_1` 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 semigroup. So the system throws this package together and fills in the `_inst_1` 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 tactic

variables (G : Type) [add_comm_group 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 variables 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 semigroup, so the function add_comm x y runs successfully and outputs a proof that x + y = y + x.

4.4.4 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 all the systems by using the @ symbol in front of the function. In Lean 3 it's all or nothing – either the system is left alone, or the user supplies every single input (including the ones which the system could have done itself). Here are some examples. Note that _ means "get the system to do it anyway".

```
import tactic

variables (G : Type) [add_comm_group G] (x y : G)

/-
add_comm : ∀ {G : Type} [_inst_1 : add_comm_semigroup G] (a b : G), a + b = b + a
-/

-- override `{}` and `[]` inputs with `@`
#check @add_comm _ _ x y -- get the system to work out `G` and the group structure
#check @add_comm G _ x y -- user supplies `G`, Lean figures out group structure
#check @add_comm _ (by apply_instance) x y -- Lean figures out `G`, user supplies group structure
```

I say "the user supplies the group structure", on the last line, but I do it by just running the apply_instance tactic, which tells the typeclass inference system to do it anyway.

For fun examples of the system failing, search for things like _ _ _ _ _ in mathlib. In Lean 4 there will be a way of supplying only those inputs which Lean was having problems with. However when the system is failing, it can sometimes be an indication that the user's definitions are not optimal.

4.5 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 going through the example of group homomorphisms in some detail. The Lean code is in sheet 4 of section 3. A group homomorphism is a structure, so it will serve as an example of how structures work.

4.5.1 The mathematics

Let G and H be groups. A group homomorphism $f : G \to H$ is a function f from G to H satisfying $f(ab) = f(a)f(b)$ for all $a, b \in G$. So, pedantically speaking, a group homomorphism is a pair consisting of a function and a proof.

4.5.2 The Lean

In Lean the way to say "let G be a group" is `variable (G : Type) [group G]`. We'll talk more about this later when we get onto classes (`group G` is a class) but let's not worry about this now. Here is the definition of a group homomorphism from `G` to `H` in sheet 4 of section 3:

```
/-- `my_group_hom G H` is the type of group homomorphisms from `G` to `H`. -/
@[ext] structure my_group_hom (G H : Type) [group G] [group H] :=
(to_fun : G → H)
(map_mul' (a b : G) : to_fun (a * b) = to_fun a * to_fun b)
```

The first line, starting with `/--`, is the docstring for the definition, i.e., an explanation of the definition in human language. This means that whenever you see `my_group_hom` in some Lean code, hovering over `my_group_hom` will display a pop-up with this explanation.

The next line (`@[ext] structure...`) announces that we're going to make a new structure called `my_group_hom` which takes as input two groups `G` and `H`. The result of this command will be a new type `my_group_hom G H`, the type of group homomorphisms from `G` to `H`. Remember that a type is just a collection of stuff, like a set; here we are making the collection $\mathrm{Hom}(G, H)$ of all group homomorphisms from `G` to `H`; the way to talk about a group homomorphism $f : G \to H$ will be `f : my_group_hom G H` (although in the file in section 3 I actually make notation `f : G →** H` for this; I would have used `f : G →* H` but that notation is already taken by Lean's actual group homomorphisms).

The `@[ext]` tag on the structure means that it's tagged with the `@[ext]` attribute. Group homomorphisms, like most things in maths, are *extensional* objects: two group homomorphisms are equal iff they take the same value on every input. Tagging the structure with this `@[ext]` attribute automatically generates some lemmas `my_group_hom.ext` and `my_group_hom.ext_iff` expressing this idea. Extensionality for group homomorphisms is provable in Lean whether or not you tag the structure; the advantage of tagging the structure is simply that the system generates the lemmas automatically.

The remaining two lines in the definition are the *fields* of the structure; they are a list of the data which we have to give to Lean to make a group homomorphism. The data is unsurprising: we want a function (called `to_fun`) which defines a map from `G` to `H`, and then we want an axiom `map_mul'` which says that for all `a b : G`, `to_fun (a * b) = to_fun(a) * to_fun(b)` (and note the usual functional programming trick of leaving off the brackets). Why did we put a prime on `map_mul`? We'll come back to this later; the point is that we want `map_mul` to be something *definitionally equal* but syntactically more beautiful.

So that's it for the definition. A group homomorphism is then two pieces of data; the function, and the axiom that it preserves multiplication. We have made a new type `my_group_hom G H`. The next thing we need to know is its constructor and eliminators; in less computer-sciency terms, we need to know how to make a term of this type, and also how to get at the data (the function and the proof), given a term of this type.

Lean has a standard way of making *any* inductive type with one constructor; the pointy bracket constructor. If `φ : G → H` and `h : ∀ a b : G, φ (a * b) = φ a * φ b` then we can make the group homomorphism associated to this data simply as `⟨φ, h⟩ : my_group_hom G H`. But there are classier ways to make a term of a structure. If you write

```
def f : my_group_hom G H :=
_
```

in VS Code, and put your cursor near the _, then a light bulb will pop up. Clicking on the light bulb and selecting “generate a skeleton for the structure under construction” will give you

```
def f : my_group_hom G H :=
{ to_fun := _,
  map_mul' := _ }
```

and you can now fill in the underscores with the function and the proof.

So there are two ways of constructing terms of type `my_group_hom G H`; what is a way of eliminating them, that is, accessing the function and the proof from the group homomorphism? The way structures work is that the full names of the fields become new functions which are added to Lean. When the structure is created, Lean generates a function `my_group_hom.to_fun`, which takes as input `f : my_group_hom G H` and outputs the function `G → H`. It also generates a function `my_group_hom.map_mul'` which takes as input `f : my_group_hom G H` and spits out a proof that `my_group_hom.to_fun f` preserves multiplication.

Because `my_group_hom.to_fun` takes as input a term `f` of type `my_group_hom...`, this means that Lean’s *dot notation* applies, giving us a shorthand; instead of writing `my_group_hom.to_fun f` we can write `f.to_fun`. Similarly, we can write `f.map_mul' : ∀ (a b : G), f.to_fun (a * b) = f.to_fun a * f.to_fun b`.

But dot notation does not solve the actual problem which we face here. Lean pedantically distinguishes between the function `G → H` and the group homomorphism `f : my_group_hom G H`; a group homomorphism is a *pair* consisting of a function and a proof. That’s all very funny of course, but in practice mathematicians don’t *want* that pedantry; given a group homomorphism `f : my_group_hom G H` they want to talk about `f g` for `g : G`, and not `is_group_hom.to_fun f g` or even `f.to_fun g`. We finish by explaining how to make this work.

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; the function is usually omitted completely by mathematicians, and is typically denoted by some kind of up arrow in Lean. One of the coercion capabilities that Lean has is that you can set up a coercion which takes a term of some type (like `f : my_group_hom G H`) and coerces it to be a function with notation `⇑f : G → H`. Furthermore, you do not even need to type that ⇑ up-arrow (which you do with `\u=` by the way); once the coercion is defined, you can just write `f g` and Lean will make sense of this.

We finish by looking at all the Lean code used to define group homomorphisms, and explain what it all means.

```
/-- `my_group_hom G H` is the type of group homomorphisms from `G` to `H`. -/
@[ext] structure my_group_hom (G H : Type) [group G] [group H] :=
(to_fun : G → H)
(map_mul' (a b : G) : to_fun (a * b) = to_fun a * to_fun b)

namespace my_group_hom

-- throughout this sheet, `G` and `H` will be groups.
variables {G H : Type} [group G] [group H]

-- We use notation `G →** H` for the type of group homs from `G` to `H`.
notation G ` →** ` H := my_group_hom G H

-- A group hom is a function, and so we set up a "coercion"
-- so that a group hom can be regarded as a function (even
-- though it's actually a pair consisting of a function and an axiom)
instance : has_coe_to_fun (G →** H) (λ _, G → H) :=
{ coe := my_group_hom.to_fun }

-- Notice that we can now write `f (a * b)` even though `f` isn't actually a
-- function, it's a pair consisting of a function `f.to_fun` and an axiom `f.map_mul
↪'`.
```

(continues on next page)

(continued from previous page)

```
-- The below lemma is how we want to use it as mathematicians though.
lemma map_mul (f : G →** H) (a b : G) :
  f (a * b) = f a * f b :=
begin
  -- the point is that f.map_mul and f.map_mul' are *definitionally equal*
  -- but *syntactically different*, and the primed version looks uglier.
  -- The point of this lemma is that `f.map_mul` looks nicer.
  exact f.map_mul' a b
end

end my_group_hom -- close namespace
```

After the structure `my_group_hom G H` is defined, we move into the `my_group_hom` namespace. This means that any definitions we make here (for example `definition foo := ...` will actually be called `my_group_hom.foo`.

The first thing we do is to set up notation `G →** H` for `my_group_hom G H`. I would have used `G →* H` but that's already used for Lean's "official" group homomorphisms.

The next thing is to set up the coercion from `G →** H` to `G → H`. The coercion is the function `my_group_hom.to_fun` which takes a group homomorphism to its underlying function.

And now we magically have the ability to write things like `f a` instead of `f.to_fun a`! So we can define `map_mul` (without the `'`) to say `f (a * b) = f a * f b`, which is the expression we want to be using. Because this definition is made in the `my_group_hom` namespace, it means that if `f : G →** H` then `f.map_mul` is the proof of `∀ a b, f (a * b) = f a * f b`. Note that `f.map_mul` and `f.map_mul'` are definitionally equal, however they are syntactically different, and the unprimed version is the cleaner one.

4.6 Classes

4.6.1 Making groups; learning about classes

Here are some of the gory details. This is technical, and maybe you don't need to know it.

In sheet 1 of the groups section, we make groups from scratch, by writing down the axioms of a group and making a "class" called `mygroup G` (not to be confused with `group G`, which is the "official" definition of a group in Lean's maths library). So what is a class, and how do classes work?

A class is a structure which is tagged with the `class` attribute. A structure is a kind of inductive type with only one constructor. An inductive type is a way to make new types in Lean. So let's start at the beginning with an explanation of the inductive type involved.

If `(G : Type)` is a type, then the idea is that we want *one object* which is going to do all the "group" stuff on `G`, and turn `G` into a group. This one object will thus be a collection of *eight* things. It will be a multiplication on `G`, an identity element, an inverse function `G → G`, and also proofs of the five group axioms (associativity, left and right identity, left and right inverse). This one object is a term of type `mygroup G`, where `mygroup G` is defined as in sheet 1 of the groups section.

But now think about how mathematicians use groups. They say "a group is a set `G` equipped with a bunch of data satisfying some axioms" but then they don't say "let's call this data (and the proofs of the fact that the data satisfies the axioms) `X`". They say "let's not give this data a name, let's just say things like '`G` is a group' even though really `G` is a set". They're saying "let `G` be both the set and the set-with-all-the-group-structure".

In Lean this is how we do it too. Here `G` is a type, not a set, and then we have a term `_inst_1 : mygroup G` which is all the group data, but we are going to put this term into Lean's typeclass inference structure, by using square brackets `[_inst_1 : mygroup G]`, and in fact we are not even going to name the term, we're just going to be writing `[mygroup G]`. Then when we see `(g : G)` this just means that `g` is a term of type `G`, but when we see `g * g` something magic is happening; Lean sees that we are trying to use a multiplication on the type `G`, and multiplication is a class, so Lean looks up `G` in its typeclass system and asks if there are any terms there which give `G` a multiplication, and then `_inst_1` pops up and says "oh yeah, I'm a group structure on `G` so I contain a bunch of stuff, including a multiplication; use that".

You can see this secret storage of all the group data and proofs if you look at Lean's tactic state in the middle of a group theory proof. For example, if you click in the middle of the proof of `inv_mul_cancel_left` in groups sheet 1, you can see that the tactic state is

```
G : Type
_inst_1 : mygroup G
a b :  G
⊢ a⁻¹ * (a * b) = b
```

That `_inst_1` is something you never need to mention, the typeclass system is taking care of it. All theorems about groups you need will magically apply to `G` because the typeclass inference system knows about `_inst_1` and will use it whenever necessary (e.g. when you want to invert an element, or apply an axiom from group theory).

4.6.2 What's the difference between a class and a structure?

With classes, the idea is that a "magic" system which we don't have to think about (Lean's typeclass system) finds the terms for you. Here is an example where you do *not* want this to happen! Say we're proving theorems about subgroups of a group `G`. We have two subgroups, and maybe we want to intersect them and prove that this is a subgroup too. If we had made subgroups into a class, then it would be the job of the magic system to supply a term of type `subgroup G` from its list, and there will only be one possible term on the list (that's how the system works). So it's going to be really hard to talk about more than one subgroup of `G` at once; Lean will keep saying "here, I have the subgroup, use this". Groups can have more than one subgroup, so subgroups are a structure.

In contrast, when we say "let `G` be a group", then 99.9 percent of the time that group structure on `G` will be one fixed group structure, and it's never going to change in the middle of a mathematical argument. So groups are a perfect example of a class – we give the magic system `G` and we say "please find me a multiplication on `G`" and the system says "oh I have exactly one group structure on `G` right here, you can use the multiplication from that". A type will 99.9 percent of the time have exactly one group structure, if it has one at all, and so this is why `group` is a class.

4.7 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`.

4.8 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 data.real.sqrt

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.

So where did this coercion come from, what other types of coercions are there, and how do we make them?

4.8.1 Making coercions

Coercions are handled by Lean's type class inference system. There are three kinds. You can coerce a term into a term (this is how a natural number becomes a real number), you can coerce a term into a type (this is how a subgroup becomes a group; remember that subgroups are terms of type `subgroup G` and in particular they're terms; if we want to treat them as groups then they need to be promoted to types); and finally there is a special coercion which turns a term into a function.

4.8.2 Coercion from terms to terms

An example of this is `coe_subtype`. If `X : Type` is a type, and `p : X → Prop` is a predicate on `X`, then one can make a subtype of `X` consisting of the terms `x : X` such that `p x` is true. The notation for this subtype is `{x : X // p x}`. Remember that in type theory, this subtype is not a "subset" of `X`; a term of type `{x : X // p x}` is a *pair* consisting of a term `x : X` and a proof of `p x`. But given such a term, we might well want to treat it as a term of type `X` anyway. Here is some code indicating how this happens automatically:

```
variables (X : Type) (p : X → Prop) (s : {x : X // p x})

#check (s : X) -- ↑s : X
```

The reason this happens is because of the term `coe_subtype : has_coe {x : X // p x} X` which is defined in core Lean and which is tagged with the `@[instance]` attribute. The `has_coe` structure just has one field, namely the function from `{x : X // p x}` to `X`, which of course just returns the term of type `X` and throws away the proof. The type class system then takes care of everything after that; if Lean sees a term of type `{x : X // p x}` where it expects a term of type `X` then it magically applies this function and denotes the function itself with the up-arrow notation.

4.8.3 Coercions from terms to types

A subgroup of a group in Lean is a term, not a type. A subgroup `H` of `G` is a term of type `subgroup G`. In particular, `H` isn't a type. So the below code shouldn't work – but it does.

```
import tactic
import group_theory.subgroup.basic -- group theory

variables (G : Type) [group G] (H : subgroup G)
variable (x : H) -- shouldn't work because H isn't a type

#check x -- x : ↥H
```

Lean promotes the term `H` to a type, because of a term of type `has_coe_to_sort (subgroup G) Type` in the typeclass inference system. Coercions to types like this have a different notation than coercions from terms to terms; a different up-arrow `↥` is used. Type it with `\u-|`.

4.8.4 Coercions from terms to functions

A group homomorphism is a term of type `G →* H` in Lean, and to give a group homomorphism is to give a function plus some proofs. In particular, a group homomorphism is not actually a function, it is a tuple consisting of a function and some proofs. But of course we'd like to regard a group homomorphism as a function, and we can, because of the third type of coercion which Lean has, triggered by terms of type `has_coe_to_fun`.

```
import group_theory.subgroup.basic

variables (G H : Type) [group G] [group H] (f : G →* H) (x : G)

#check f x -- ⇑f x : H
```

Even though `f` isn't strictly speaking a function (it's a function plus some other stuff), Lean will still let us write `f x`, and it will interpret `f` as the function `⇑f : G → H`. Here `⇑`, which you can get with `\u=`, is the function from `G →* H` to `G → H` which the typeclass inference system produced for us, from the instance `monoid_hom.has_coe_to_fun : has_coe_to_fun (M →* N) (λ _, M → N)`.

CHAPTER

FIVE

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.

5.1 Tactic cheatsheet

If you think you know which part of the tactic state your "next move" should be (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`
`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`	`split`	`cases h with hP hQ`
`P ↔ Q`	`split`	`cases h` or `rw h`
`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`

5.2 Tactic documentation

5.2.1 apply

Warning: The `apply` tactic does *one very specific thing*, 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. 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".

Summary

If your local context is

```
h : P → Q
⊢ Q
```

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

Note: `apply h` will *not work* unless `h` is of the form `P → Q`. It will also *not work* unless the goal is equal to the conclusion 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, the `apply` tactic does this: 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 a hypothesis which is

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

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

2) `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`.

3) 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 _`.

5.2.2 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 `;` instead of `,` at the end of a tactic, to try and prove 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 `split; assumption`. The semicolon, used instead of a comma, means "do `split` and then do `assumption` on all goals produced by `split`".

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. Rob Lewis has made some excellent videos on tactic writing at **insert link here**. Note that there will be nothing about tactic writing in this course because the author doesn't have the first clue about it.

5.2.3 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 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 `h : P` and in the second you have a new hypothesis `h : ¬P`.
2) If you already have a hypothesis `h` then this can get a bit confusing, so you can also do `by_cases hP : P`; then your new hypotheses will be `hP : P` and `hP : ¬P`.

Details

For those of you interested in constructive mathematics, 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, however you won't learn any other tactics if you keep using it!

5.2.4 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`.

```
example (P : Prop) : ¬ ¬ P → P :=
begin
  intro hnnP, -- assume ¬ ¬ P
  by_contra hnP, -- goal is now false
  apply hnnP, -- goal is now ¬ P
  exact hnP,
end
```

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. However `exfalso` often terminates in under a millisecond, a speed which `by_contra` can only dream of.

5.2.5 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` will make hypothesis `h` vanish and will replace it with the two "components" which made the proof of `h` in the first place.

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 `h_left : P` and `h_right : Q`. These are not the best names for hypotheses; 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 need to keep a copy of `h` around, you could always try `have h2 := h, 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⟩`.

5.2.6 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 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.

5.2.7 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*.

5.2.8 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,
apply h,
exact hP
```

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.

5.2.9 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 sets are the same if they have the same elements, two subgroups 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 group_theory.subgroup.basic

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

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

```
import group_theory.subgroup.basic

example (G : Type) [group G] (K L : subgroup G)
  (h : ∀ a, a ∈ K ↔ a ∈ L) : K = L :=
begin
  -- ⊢ K = L
  ext g,
  -- ⊢ g ∈ K ↔ g ∈ L
  apply h, -- goals accomplished
end
```

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 in the second example above, `ext g` does the same thing as `apply subgroup.ext, intro g`. Furthermore, if you try `#print subgroup.ext`, you'll see a list at the very top of the output, starting with an `@`, and somewhere in that list is the word `ext`, meaning that `subgroup.ext` is tagged with the `@[ext]` attribute, 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.

5.2.10 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, exact 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,`. 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.

5.2.11 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, a + a = 2 * a
```

into

```
x : X
⊢ x + x = 2 * x
```

where `X` is the type of `x` and `a` (for example `X` could be the naturals or the reals, or some other type equipped with an addition and multiplication).

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).

5.2.12 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.

5.2.13 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.

5.2.14 linarith

Summary

The `linarith` tactic solves certain kind of linear equalities and inequalities. Unlike the `ring` tactic, `linarith` uses hypotheses in the tactic state.

You can feed extra proofs into `linarith` by adding them as a list like this: `linarith [h1, h2]`. We will see examples of this below.

Examples

To be written. Basically, if you have a bunch of hypotheses like `h1 : a < b`, `h2 : b <= c`, `h3 : c = d` and a goal of `a < d` then `linarith` will solve it; it knows a ton of results about how to put inequalities together. As another example if you have hypotheses `h1 : a < b` and `h2 : c < d` then it will prove a goal `c + a < b + d`.

5.2.15 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).

5.2.16 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 `A = B`, `A ≤ B`, `A < B` and `A ≠ B`, as long as `A` and `B` involve only numbers like `-37/5`.

Examples

(with `data.real.basic` imported to get the reals, and of course `tactic` imported to get the tactics)

```
example : (1 : ℝ) + 1 = 2 :=
begin
  norm_num,
end

example : (1 : ℝ) + 1 ≤ 3 :=
begin
  norm_num,
end

example : (1 : ℝ) + 1 < 4 :=
begin
  norm_num,
end

example : (1 : ℝ) + 1 ≠ 5 :=
begin
  norm_num,
end
```

I put a comma after `norm_num` so that I can check that the "goals accomplished" message is there.

`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 :=
begin
  norm_num,
end
```

5.2.17 nth_rewrite

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_rewrite`. For example `nth_rewrite 2 h` will change only the third `a` into `b`, and `nth_rewrite 0 h` will turn only the first one.

Examples

1) Faced with

```
h : x = y
⊢ x * x = a
```

the tactic `nth_rewrite 0 h` will turn the goal into `⊢ y * x = a` and `nth_rewrite 1 h` will turn it into `⊢ x * y = a`. Compare with `rw h` which will turn it into `⊢ y * y = a`.

2) `nth_rewrite` works on hypotheses too. If `h : x = y` is a hypothesis and `h2 : x * x = a` then `nth_rewrite 0 h at h2` will change `h2` to `y * x = a`.

3) Just like `rw`, `nth_rewrite` accepts `← h3` if you want to change the right hand side of `h3` to the left hand side.

Further notes

Variants of `nth_rewrite` are `nth_rewrite_lhs` and `nth_rewrite_rhs` which operate only on the left hand or right hand side of an = or ↔.

5.2.18 obtain

Summary

The `obtain` tactic can be used to do `have` and `cases` all in one go.

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.

Details

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.

5.2.19 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 : ∃ (H : 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 : 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.

Details

Note that `rcases` works up to *definitional equality*.

Variants of `intro` include the *intros* tactic (`intro` + `rcases`) and the *obtain* tactic (`have` + `rcases`).

5.2.20 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 : ℝ) (H : δ > 0), |f y - f x| < δ
```

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

Further notes

refine works up to definitional equality, as does most tactics.

5.2.21 refl

Summary

The refl tactic proves goals of the form ⊢ x = y where x and y are *definitionally equal*. More generally it proves goals of the form R x y if x and y are definitionally equal and R is a reflexive binary relation.

Examples

1) refl will prove ⊢ x = x.
2) refl will prove ⊢ P ↔ P.
3) refl will prove ⊢ ¬ P ↔ (P → false) because even though the two sides of the iff are not syntactically equal, they are definitionally equal.
4) refl 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 algorithmically 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 refl 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 refl, but refl 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 refl 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.

5.2.22 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.

5.2.23 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.

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) :=
begin
  ring
end
```

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, 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 :=
begin
  have h2 : x^4=(x^2)^2, -- introduce a new goal
  { ring }, -- proves ``h2``; note that the `{...}` brackets are good practice
            -- when there is more than one goal
  rw h2, -- goal now ``⊢ (x ^ 2) ^ 2 = y ^ 4``...
  rw h, -- ...so ``rw h`` now works, giving us ``⊢ (y ^ 2) ^ 2 = y ^ 4``
  ring,
end
```

Further notes

There's a variant of `ring` called `ring!` but I don't know a concrete example where one works and the other doesn't.

`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`.

`ring` might not work sometimes with variable exponents like `x^n`. The variant `ring_exp` is better at dealing with them.

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.

5.2.24 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, 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 `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
```

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`.

5.2.25 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 by ``Q``s. ``rw works up to syntactic equality, and only works with = and ``

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 refl 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 refl 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_rewrite 0 h (for the first one) or nth_rewrite 1 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 need to use the more powerful simp_rw tactic; simp_rw add_zero works and changes the goal to ∀ (x : ℕ), x = 0 + x.

5.2.26 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)) :=
begin
  simp
end
```

If you want to know which lemmas `simp` used, you can use the related tactic `squeeze_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 data.finset.basic
import data.real.basic

open_locale big_operators

open finset

example (n : ℕ) :
  □ i in range n, (i : ℝ) = n * (n - 1) / 2 :=
begin
  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
```

(continues on next page)

(continued from previous page)

```
    ring, -- a more appropriate tactic to finish the job
  }
end
```

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 are in the explanation of how to use simp is in the community documentation and in section 5.7 of Theorem Proving In Lean .

Further notes

The two related tactics simp? and squeeze_simp both attempt to figure out exactly which lemmas simp used. Sometimes they give different answers; it turns out that because simp is written in C++ rather than in Lean, reverse-engineering it is a subtle problem.

5.2.27 simpa

Summary

If you have a goal and a hypothesis h, and 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 tactic
import data.real.basic

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

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 example here simpa using h won't work because the simplifier doesn't know hxy (the simplifier doesn't use hypotheses in the tactic state).

```
import tactic

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

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`.

5.2.28 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, ...` then looking at your tactic state more carefully you might find that it actually says something like `∀ (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
{ norm_num }, -- `h37` now proved
specialize h h37
```

However you could just drop the proof in directly:

```
specialize h (begin norm_num end)
```

There is special notation `by` for tactic blocks with just one tactic in, so in fact the following variants would also work

```
have h37 : (37 : ℝ) > 0 := by norm_num, -- still only one goal
specialize h h37
```

or even

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

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 norm_num).`

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`.

5.2.29 split

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 `split` tactic will turn your goal into these multiple simpler goals.

Examples

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

Further notes

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

5.2.30 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 `refl` 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 `split` tactic also proves a `true` goal, although you would have to learn a bit about inductive types to understand why. Also `triv` solves a true gol in 3 millisends on a modern computer, whereas `split` takes 4 milliseconds.

5.2.31 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 tactic
import data.real.basic

example : ∃ (a b : ℝ), a + b = 37 :=
begin
```

(continues on next page)

(continued from previous page)

```
  use [5, 32],
  -- ⊢ 5 + 32 = 37
  norm_num,
end
```

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”.