Introduction to Formal Mathematics with Lean 4

Xingyu Zhong

December, 2025

Table of Contents

Preface

This book, or repository, contains the lecture notes and supplementary materials for the hobby course *Introduction to Formal Mathematics with Lean 4*, offered in Fall 2025 at Beijing Institute of Technology. It is designed as a companion to the course *Abstract Algebra* offered in the same semester, mainly intended for students with a background in undergraduate-level pure mathematics.

Goals

The main goals of this course are

- to get used to think formally
- to migrate from set theory to dependent type theory
- to practice basic skills to translate statements and proofs into Lean 4
- to familiarize with the Mathlib 4 library
- to know how to find existing theorems, how the community works
- to acquire enough common senses to read the bibles (MiL, TPiL, Mathlib 4 Doc, etc.) by yourself for future formalization projects

Approach

The style of this course is inspired by Kevin Buzzard's 2024 course on formalising mathematics in the Lean theorem prover and many other courses, that is: dozens of Lean files packed with well-organized examples and exercises should get you started. In fact, almost all chapters of this book is just a single Lean file with comments, converted into other formats later on. It is best to download the course repository and read it alongside a Lean 4 environment, so that you can try out the code examples and exercises interactively.

We try to organize the materials in a way that balances theory and practice. The main feature of this book is that we introduce Lean 4 and formalized mathematics by "illustrating the theory", that is: we accept the Mathlib definitions "as is", but reprove the major consequences that constitute the theory. Once a result is proved, we immediately relate it back to its Mathlib version (via a definitional equality, mostly), and use the Mathlib version in the subsequent developments. Hopefully, this may help you familiarize with the Mathlib API more quickly, while at the same time understand how the theory is builded up mathematically, without being lost in the huge Mathlib codebase.

We try to keep the materials well-organized, but due to the extra-curricular nature of the course, the lecture notes are often prepared in a hurry, so please forgive the messiness here and there. Due to the scope of the course, the limited time and my personal inability, several important topics, such as inductive types and inductive proofs, discrete mathematics, type classes and coercions, are only briefly touched upon or omitted. The readers are encouraged to explore these topics by themselves using the references provided at the end of this preface. We might fill in these gaps in the next versions of this book.

Structure

The book is divided into four parts[1]. The first part is purely introductory, advertising that formal mathematics and the Lean language is interesting and increasingly important nowadays. In the second part, we illustrate how first-order logic is built up in Lean's dependent type theory. Alongside the way, the readers are familiarized with Lean 4's syntax, semantics, and basic proof techniques. In the third part, we advance from `Prop` to `Type`, introducing numbers, functions, equalities and inequalities. This cultimates in Chapter 7, where we do some mathematical analysis in Lean 4. The last part is devoted to algebraic structures, where we introduce the Mathlib philosophy of organizing algebraic structures, morphisms, substructures, and quotients, especially in the case of groups. We prove the first isomorphism theorem for groups as a grand finale.

Instructions

We maintain an online version of the lecture notes with embedded Lean code if you prefer to read it in your browser. A printed PDF version is also provided as a souvenir, though no special effort has been made to resolve the hyperlinks.

For installation, first make sure you have installed Git, VSCode and Lean 4 extension for VSCode correctly. Refer to the installation guide `https://lean-lang.org/install/` if you haven't done so. Then execute the following commands in your terminal:

```
git clone git@github.com:sun123zxy/2025fall-lean4-teach.git # download the repository
cd 2025fall-lean4-teach
lake update # download Mathlib source files
lake exe cache get # download compiled Mathlib artifacts
```

To update the repository, make sure you have discarded any local changes (otherwise you may need to merge manually). Then execute the following commands in your terminal:

```
git pull
```

Compiling the book

Both the online and the PDF version of this book are prepared by SunQuarTeX, a publishing system based on Quarto and LaTeX. Refer to `https://github.com/sun123zxy/sunquartex` for more information.

Portals

- Course repository: `https://github.com/sun123zxy/2025fall-lean4-teach`
- Online lecture notes: `https://sun123zxy.github.io/2025fall-lean4-teach/`
- Online compiler: `https://live.lean-lang.org`
- Community (Lean Zulip): `https://leanprover.zulipchat.com/`
- Lean 4 tactics cheatsheet: `https://leanprover-community.github.io/papers/lean-tactics.pdf`

[1] each entitled with a valid Lean 4 keyword

References

We recommend the following resources for further study of Lean and formalized mathematics.

Introductory videos and articles:

- CAV2024: https://leodemoura.github.io/files/CAV2024.pdf
- Terence Tao at IMO 2024: AI and Mathematics: https://www.youtube.com/watch?v=e049IoFBnLA
- Lean 的前世今生: https://zhuanlan.zhihu.com/p/183902909
- Natural Number Game: https://adam.math.hhu.de/#/g/leanprover-community/nng4
- Computational Trilogy - nLab: https://ncatlab.org/nlab/show/computational+trilogy

Bibles for further study:

- Mathematics in Lean (MIL): https://leanprover-community.github.io/mathematics_in_lean/

  A comprehensive tutorial for mathematicians to get started with Lean and the mathlib library. Focuses on building up mathematical structures.

- Theorem Proving in Lean 4: https://leanprover.github.io/theorem_proving_in_lean4/

  Strong emphasis on logic and dependent type theory. Excellent for both mathematicians and computer scientists.

- Lean Language Manual: https://lean-lang.org/doc/reference/latest/

  Comprehensive, precise description of Lean: a reference work in which Lean users can look up detailed information, rather than a tutorial intended for new users.

- Type Theory - nLab: https://ncatlab.org/nlab/show/type+theory

  If you want to understand the theoretical foundations of Lean, this is a good place to start.

- Other bibles: https://lakesare.brick.do/all-lean-books-and-where-to-find-them-x2nYwjM3AwBQ

Courses and lecture notes:

- Kevin Buzzard's 2024 course on formalising mathematics in the Lean theorem prover: https://github.com/ImperialCollegeLondon/formalising-mathematics-2024

Acknowledgements

The main motivation for this course is to promote Lean 4 and formalized mathematics to students majoring in mathematics at Beijing Institute of Technology, as well as to provide a structured exposition of Lean 4 and formalized mathematics with the author's personal flavor. Prior to this course, the author was introduced to Lean and formal mathematics in the summer of 2024 via an invited talk by Jia Li (李嘉) on Project Numina at Beijing Institute of Technology. One year later, he had the chance to participate in the 2025 algebra and formalization summer school held in Renmin University of China and Beijing International Center for Mathematical Research. The author wishes to express gratitude to the organizers and participants of these events for the inspiring experience, especially Professor Riccardo Brasca for his wonderful lectures, Wanyi He (何琬仪), Jiedong Jiang (姜杰东), and Yicheng Tao (陶亦成) for their tutorials and kind help on Lean 4. The teaching style of this course was deeply inspired by Kevin Buzzard's 2024 course on formalising mathematics in the Lean theorem prover. The author would like to thank Professor Yangyu Fan (范洋宇), for kindly allowing this hobby course to be offered alongside his *Abstract Algebra* course in Fall 2025 at Beijing Institute of Technology. The author is grateful to Junchao Huang (黄俊超), for providing the after-class exercises, and his valuable suggestions from a perspective of AI4Math. Finally, the author would like to thank all the fellow students who kindly scheduled their time to participate in this purely extra-curricular course. Yihan Zhuo (卓逸涵) deserve special thanks for his full attendance and active participation throughout the course.

Xingyu Zhong (钟星宇)
`https://scholar.sun123zxy.top`

December 2025, Beijing

Part I

Prelude

Introduction to Formal Mathematics with Lean 4

For newcomers to formalization methods

What is formalization

Natural language vs. formal language

- ambiguity in natural language
  - implicit assumptions
  - skipping details: "It's clear that we have…"
  - "viewed as" arguments: $V^{**} = V$, $(A \times B) \times C = A \times (B \times C)$ [2]
  - abuses of notation: $3 \in \mathbb{Z}/5\mathbb{Z}$, $\mathbb{C} \subseteq \mathbb{C}[x]$
- precision in formal language
  - computer programs are formal languages

Mathematical proofs vs. Computer programs[3]

Logic	Programming
proposition	type
proof	term
proposition is true	type has a term
proposition is false	type doesn't have a term
logical constant `TRUE`	unit type
logical constant `FALSE`	empty type
implication $\rightarrow$	function type
conjunction $\wedge$	product type $\prod$
disjunction $\vee$	sum type $\sum$
universal quantification $\forall$	dependent product type $\prod$
existential quantification $\exists$	dependent sum type $\sum$

Table 1: Curry–Howard correspondence

[2]Knowledgable audience may recognize them as examples of natural isomorphisms in category theory.
[3]see also Computational Trilogy, with category theory as the third vertex

Set theory vs. Type theory

- Mathematicians choose axiomatic set theory (with first-order logic) as the foundation of mathematics.
  - naive set theory fits human's intuition well
- Type theory is an alternative foundation that is equally expressive, but more suitable for computer formalization.

Set Theory	Type Theory
everything is a set	everything has a type
$3 \in \mathbb{R}$ is a proposition	$(3 : \mathbb{R})$ is a typing judgment
$\mathbb{Q} \subseteq \mathbb{R}$ is an inclusion	$\mathbb{Q} \to \mathbb{R}$ is a type conversion

What is Lean 4

- A modern functional programming language designed for theorem proving

  "Lean is based on a version of dependent type theory known as the *Calculus of Constructions*, with a countable hierarchy of non-cumulative universes and inductive types." — Theorem Proving in Lean 4

Lean's dependent type theory

- Dependent type theory is a powerful extension of type theory where
  - types may depend on terms "given before" them
  - first-order logic can be implemented in dependent type theory
- functions, inductive types and quotient types[4] are the basic methods to construct new types.

Set Theory	Lean's dependent type theory
$\forall x \in \mathbb{R},\, x^2 \geq 0$	has type $(x : \mathbb{R}) \to (x^2 \geq 0)$
$(n \in \mathbb{N}) \mapsto (1, 0, \ldots, 0) \in \mathbb{R}^n$	has type $(n : \mathbb{N}) \to \mathbb{R}^n$
$\{0, 1\} = 2$ is a set equality	make no sense
cardinality is an equivalence class	is a quotient type
Russell's paradox	Girard's paradox

An example Lean 4 code

- FLT
- TendsTo

[4]Though seemingly redundant, there are reasons for making quotient types as a fundamental constructing method. funext thesis

Why formalize

The rise of AI

AI excels in Python. Why not Lean?

- Automated theorem proving
  - especially those "abstract nonsense"
  - full-auto (create a proof without human interaction)
  - semi-auto (suggest tactics)
    - `exact?`, Github Copilot, …
- Natural language to formal language
  - automatically transplanting textbooks and papers into Lean
  - full-auto (translate without human interaction)
  - bolt-action (search for existing theorems)
    - LeanSearch, LeanExplore, …
  - Converse? Already happening!
- Proposing conjectures
  - on which facts should we care about

Rigor matters

- It's the foundation of mathematics
- Imprecise natural language often leads to misunderstandings and glitches
  - Especially when proofs get longer and longer
- formalization fully confirms the correctness of a theorem
  - things that are too "technical" (boring) or simply impossible to verify by oneself
    - e.g. classification of finite simple groups
    - e.g. "technical"

      "I spent much of 2019 obsessed with the proof of this theorem, almost getting crazy over it. In the end, we were able to get an argument pinned down on paper, but I think nobody else has dared to look at the details of this, and so I still have some small lingering doubts." — Peter Scholze

"Mathematical engineering"

- manipulating tons of theorems and proofs with mature software engineering techniques
- referencing existing theorems as dependencies
- collaborative work across the globe

  The beauty of the system: you do not have to understand the whole proof of FLT in order to contribute. The blueprint breaks down the proof into many many small lemmas, and if you can formalise a proof of just one of those lemmas then I am eagerly awaiting your pull request. — Kevin Buzzard on the FLT Project

Formalization as learning

- proofs with infinite detail
  - intuitive textbooks, rigorous formalization
- makes us understand things better
  - Global: How to build natural numbers from scratch?
    - natural number game
    - A journey to the world of numbers, by Riccardo Brasca
  - Local: reducing the cognitive load
    - (With good organization at the beginning) you can focus on small parts of the proof at a time

Why now formalize

Formalization becomes more accessible

Mathematician-friendly languages, interfaces, tools and community emerges:

- Lean 4 with VSCode extension, modern interactive theorem prover made for mathematicians
- formalization becomes more and more fashionable
- big names works on formalization:
  - Kevin Buzzard works on formalizing FLT
  - Peter Scholze's work on condensed mathematics has been formalized
  - Terrence Tao gave a talk on formalization in IMO 2024 and wrote a Lean 4 companion of his book "Analysis I" recently
- computer scientists and volunteering mathematicians run Lean 4 community collaboratively

Mathlib 4 is expanding explosively

By the time of 2025/09/16, Mathlib 4 has[5]

Lines of code	Definitions	Theorems	Contributors
1950000	115438	232204	653

- undergraduate may contribute: some low-hanging fruits

[5]Statistics fetched from Mathlib statistics

How to formalize

The goal

The goal, at the end of this course, is

- to get used to think formally
- to migrate from set theory to dependent type theory
- to practice basic skills to translate statements and proofs into Lean 4
- to know how to find existing theorems, how the community works
- to acquire enough common senses to read the bibles (MiL, TPiL, Mathlib 4 Doc, etc.) by yourself for future formalization projects
- (optimistically) to set up a Lean 4 formalization club at BIT!

How will we learn

As mathematicians, we learn Lean 4 to formalizing mathematics. We learn by practice.

- Dozens of Lean files packed with well-organized examples and exercises suffice to get you started, suitable for both guided study and self study. Most lectures will be given in this style.

- This style of teaching is inspired by Kevin Buzzard's 2024 course on formalising mathematics in the Lean theorem prover and many other courses.

What we won't cover

Due to the limited time, my personal inability and the design of this course, we might not be able to cover:

- a deep discussion into dependent type theory or Lean as a programming language itself

  - Read TPiL for a Lean 4 tutorial that emphasizes on type theory.
  - Read FPiL for a Lean 4 tutorial that focuses more on functional programming.
  - Refer to Lean Language Manual for precise specifications.

- systematic exposition of how a particular branch of mathematics is formalized in Mathlib 4

  - Read MiL for this purpose.

- how to organize a massive formalization project from scratch, i.e. project management

  - somewhat subtle, might can only be learned by reading Mathlib codes and practical experience

Disclaimer

- Formalization is tedious in its nature, Lean is no exception

- Type conversions can be an extra burden (exclusive for type-theory-based systems)

- Knowledge needs to be re-learned before being referenced

- Different people may formalize the same thing in different ways

If these do not scare you away...

Welcome aboard. Have fun formalizing mathematics!

Resources

- course repository
- online documentation

Chapter 1

At the Very Beginning

You may skip the materials tagged with [IGNORE] for the first runthrough. They could be not well-explained, or too advanced for now.

Materials tagged with [EXR] are recommended for you to try before looking at the solution.

Materials tagged with [TODO] means that I'm still working on it, or I'm not sure about the content yet. Feel free to give your suggestions!

1.1 A first glance

Have a look at the sample Lean code below. Can you understand what it means, without any prior knowledge of Lean?

```
import Mathlib

theorem FLT (n : ℕ) (hn : n > 2) (a b c : ℕ) :
    a ≠ 0 → b ≠ 0 → c ≠ 0 → a^n + b^n ≠ c^n := by
  sorry

def TendsTo (a : ℕ → ℝ) (t : ℝ) : Prop :=
  ∀ ε > 0, ∃ n₀ : ℕ, ∀ n, n₀ ≤ n → |a n - t| < ε

example : TendsTo (fun _ ↦ 998244353) 998244353 := by
  unfold TendsTo
  intro ε hε
  use 19260817
  intro n hn
  simp [hε]

theorem tendsTo_add {a b : ℕ → ℝ} {A : ℝ} {B : ℝ} (ha : TendsTo a A) (hb : TendsTo b B) :
    TendsTo (fun n => a n + b n) (A + B) := by
  sorry

theorem tendsTo_sandwich {a b c : ℕ → ℝ} {L : ℝ} (ha : TendsTo a L) (hc : TendsTo c L)
    (hab : ∀ n, a n ≤ b n) (hbc : ∀ n, b n ≤ c n) : TendsTo b L := by
  sorry
```

1.2 At the very beginning...

There are some basic notions you should be familiar with: `:` and `:=`.

`3 : ℕ` means that `3` is a term of type `ℕ`.

By the Curry–Howard correspondence, `hp : p` means that `hp` is a proof of the proposition `p`.

```
#check 3
#check ℕ

#check ∀ x : ℝ, 0 ≤ x ^ 2
#check sq_nonneg
#check (sq_nonneg : ∀ x : ℝ, 0 ≤ x ^ 2)
```

`:=` is used to define terms.

```
def myThree : ℕ := 3

#check myThree
```

`theorem` is just a definition in the `Prop` universe By the Curry–Howard correspondence, for `theorem`, behind `:`, the theorem statement follows; behind `:=`, a proof should be given.

```
theorem thm_sq_nonneg : ∀ x : ℝ, 0 ≤ x ^ 2 := sq_nonneg

-- `example` is just an anonymous theorem
example : ∀ x : ℝ, 0 ≤ x ^ 2 := thm_sq_nonneg
```

We shall work out the basic logic in Lean's dependent type theory.

In this part, we cover:

- Implication
  - Syntax for defining functions / theorems
- Tactic Mode

[IGNORE] You may notice along the way that except `→`, all other logical connectives are defined as *inductive types*. And they have their own *self-evident introduction rules* and *elimination rules*. If possible, we might discuss inductive types later in this course. These logical connectives serve as good examples.

Part II

Prop

Chapter 2

Logic (Part I)

2.1 Implication →

Implication `→` is the most fundamental way of constructing new types in Lean's dependent type theory. It's one of the first-class citizens in Lean.

In the universe of `Prop`, for propositions `p` and `q`, the implication `p → q` means "if `p` then `q`".

```
section

variable (p q r : Prop) -- this introduces global variables within this section

#check p
#check q
#check p → q
```

`→` is right-associative. In general, hover the mouse over the operators to see how they associate. so `p → q → r` means `p → (q → r)`. You may notice that this is logically equivalent to `p ∧ q → r`. This relationship is known as *currification*. We shall discuss this later.

modus ponens

```
theorem mp : p → (p → q) → q := by sorry -- `sorry` is a placeholder for unfinished proofs
```

By the Curry–Howard correspondence, `p → q` is also understood as a function that takes a proof of `p` and produces a proof of `q`.

We introduce an important syntax to define functions / theorems: When we define a theorem `theorem name (h1 : p1) ... (hn : pn) : q := ...`, we are actually defining a function `name` of type `(h1 : p1) → ... → (hn : pn) → q`. Programmingly, `h1`, …, `hn` are the parameters of the function and `q` is the return type.

The significance of this syntax, compared to `theorem name : p1 → ... → pn → q := ...`, is that now `h1`, …, `hn`, proofs of `p1`, …, `pn`, are now introduced as hypotheses into the context, available for you along the way to prove `q`.

this proves a theorem of type `p → p`

```
example (hp : p) : p := hp
```

modus ponens, with a proof

```
example (hp : p) (hpq : p → q) : q := hpq hp
```

A function can also be defined inline, using `fun` (lambda syntax): `fun (h1 : p1) ... (hn : pn) ↦ (hq : q)` defines a function of type `(h1 : p1) → ... → (hn : pn) → q`

Some of the type specifications may be omitted, as Lean can infer them.

```
example : p → p := fun (hp : p) ↦ (hp : p)
example : p → p := fun (hp : p) ↦ hp
example : p → (p → q) → q := fun (hp : p) (hpq : p → q) ↦ hpq hp
example : p → (p → q) → q := fun hp hpq ↦ hpq hp
```

2.2 Tactic mode

Construct proofs using explicit terms is called *term-style proof*. This can be tedious for complicated proofs.

Fortunately, Lean provides the *tactic mode* to help us construct proofs interactively.

`by` activates the tactic mode.

The tactic mode captures the way mathematicians actually think: There is a goal `q` to prove, and we have several hypotheses `h1 : p1`, …, `hn : pn` in the context to use. We apply tactics to change the goal and the context until the goal is solved. This produces a proof of `p1 → ... → pn → q`.

```
example (hp : p) : p := by exact hp
```

tactic: `exact` If the goal is `p` and we have `hp : p`, then `exact hp` solves the goal.

`exact?` may help to close some trivial goals

```
example (hp : p) (hpq : p → q) : q := by exact?
```

tactic: `intro` Sometimes a hypothesis is hidden in the goal in the form of an implication. If the goal is `p → q`, then `intro hp` changes the goal to `q` and adds the hypothesis `hp : p` into the context.

modus ponens, with a hidden hypothesis

```
example (hp : p) : (p → q) → q := by
  intro hpq
  exact hpq hp
```

```
example (hq : q) : p → q := by
  intro _  -- use `_` as a placeholder if the introduced hypothesis is not needed
  exact hq
```

modus ponens, with two hidden hypothesis

```
example : p → (p → q) → q := by
  intro hp hpq -- you can `intro` multiple hypotheses at once
  exact hpq hp
```

[EXR] transitivity of →

```
example : (p → q) → (q → r) → (p → r) := by
  intro hpq hqr hp
  exact hqr (hpq hp)
```

tactic: `apply` If `q` is the goal and we have `hpq : p → q`, then `apply hpq` changes the goal to `p`.

modus ponens

```
example (hp : p) (hpq : p → q) : q := by
  apply hpq
  exact hp
```

[EXR] transitivity of →

```
example (hpq : p → q) (hqr : q → r) : p → r := by
  intro hp
  apply hqr
  apply hpq
  exact hp
```

[IGNORE] Above tactics are minimal and sufficient for simple proofs. When proofs went more complicated, you may want more tactics that suit your needs. Remember your favorite tactics and use them accordingly.

tactic: `specialize` If we have `hpq : p → q` and `hp : p`, then `specialize hpq hp` reassigns `hpq` to `hpq hp`, a proof of `q`.

```
example (hp : p) (hpq : p → q) : q := by
  specialize hpq hp
  exact hpq

example (hpq : p → q) (hqr : q → r) : p → r := by
  intro hp
```

```
  specialize hpq hp
  specialize hqr hpq
  exact hqr
```

tactic: `have` `have` helps you to state and prove a lemma in the middle of a proof. `have h : p := hp` adds the hypothesis `h : p` into the context, where `hp` is a proof of `p` that you provide.
`haveI` is similar to `have`, but it adds the hypothesis as `this`.

```
example (hpq : p → q) (hqr : q → r) : p → r := by
  intro hp
  have hq : q := hpq hp
  have hr : r := by -- combine with `by` is also possible
    apply hqr
    exact hq
  exact hr
```

tactic: `suffices` Say our goal is `q`, `suffices hp : p from hq` changes the goal to `p`, as long as you can provide a proof `hq` of `q` from a proof `hp` of `p`. You may also switch to the tactic mode by `suffices hp : p by ...`

```
example (hpq : p → q) (hqr : q → r) : p → r := by
  intro hp
  suffices hq : q from hqr hq
  exact hpq hp

example (hpq : p → q) (hqr : q → r) : p → r := by
  intro hp
  suffices hq : q by
    apply hqr
    exact hq
  exact hpq hp
```

`show` (it is not a tactic!) Sometimes you want to clarify what exactly you are giving a proof for. `show p from h` make sure that `h` is interpreted as a proof of `p`. `show p by ...` switches to the tactic mode to construct a proof of `p`.

```
example (hpq : p → q) (hqr : q → r) : p → r := by
  intro hp
  exact hqr (show q by apply hpq; exact hp)

end
```

Chapter 3

Logic (Part II)

- `And` and `Or`
- `Forall` and `Exists`

3.1 `And` and `Or`

In Lean's dependent type theory, `∧` and `∨` serve as the *direct product* and the *direct sum* in the universe of `Prop`.

Eagle-eyed readers may notice that `∧` and `∨` act similarly to Cartesian product and disjoint union in set theory.

They are also constructed as inductive types.

```
import Mathlib

section

variable (p q r : Prop)
```

3.1.1 `And` (`∧`)

Introducing `And`

The only constructor of `And` is `And.intro`, which takes a proof of `p` and a proof of `q` to produce a proof of `p ∧ q`.

It is self-evident. Regard this as the *universal property of the direct product* if you like.

```
#check And.intro

example (hp : p) (hq : q) : p ∧ q := And.intro hp hq
```

`And.intro hp hq` can be abbreviated as `⟨hp, hq⟩`, called the *anonymous constructor*.

```
example (hp : p) (hq : q) : p ∧ q := ⟨hp, hq⟩
```

introducing nested `And`

```
example (hp : p) (hq : q) (hr : r) : p ∧ q ∧ r := by
  exact ⟨hp, hq, hr⟩ -- equivalent to `⟨hp, ⟨hq, hr⟩⟩`
```

`constructor` tactic applies `And.intro` to split the goal `p ∧ q` into subgoals `p` and `q`. You may also use the anonymous constructor notation `⟨hp, hq⟩` to mean `And.intro hp hq`.

use · to focus on the first goal in your goal list.

```
example (hp : p) (hq : q) : p ∧ q := by
  constructor
  · exact hp
  · exact hq
```

`on_goal` tactic can be used to focus on a specific goal.

```
example (hp : p) (hq : q) : p ∧ q := by
  constructor
  on_goal 2 => exact hq
  exact hp
```

`all_goals` tactic can be used to simultaneously perform tactics on all goals.

```
example (hp : p) : p ∧ p := by
  constructor
  all_goals exact hp
```

`assumption` tactic tries to close goals using existing hypotheses in the context. Can be useful when there are many goals.

```
example (hp : p) (hq : q) : p ∧ q := by
  constructor
  all_goals assumption
```

`split_ands` tactic is like `constructor` but works for nested `And`s.

```
example (hp : p) (hq : q) (hr : r) : p ∧ q ∧ r := by
  split_ands
  · exact hp
  · exact hq
  · exact hr
```

[EXR] →–∨ distribution. Universal property of the direct product.

```
example (hrp : r → p) (hrq : r → q) : r → p ∧ q := by
  intro hr
  exact ⟨hrp hr, hrq hr⟩
```

Eliminating `And`

`And.left` and `And.right` are among the elimination rules of `And`, which extract the proofs of `p` and `q`.

```
#check And.left
#check And.right
example (hpq : p ∧ q) : p := hpq.left
example (hpqr : p ∧ q ∧ r) : r := hpqr.right.right
```

`rcases hpq with ⟨hp, hq⟩` is a tactic that breaks down the hypothesis `hpq : p ∧ q` into `hp : p` and `hq : q`. Equivalently you can use `have ⟨hp, hq⟩ := hpq`.

```
example (hpq : p ∧ q) : p := by
  rcases hpq with ⟨hp, _⟩
  exact hp
```

implicit break-down in `intro`

```
example : p ∧ q → p := by
  intro ⟨hp, _⟩
  exact hp
```

nested `And` elimination

```
example (hpqr : p ∧ q ∧ r) : r := by
  rcases hpqr with ⟨_, _, hr⟩
  exact hr
```

[EXR] `And` is symmetric

```
example : p ∧ q → q ∧ p := by
  intro ⟨hp, hq⟩
  exact ⟨hq, hp⟩
#check And.comm -- above has a name
```

[EXR] `→`–`∨` distribution, in another direction.

```
example (hrpq : r → p ∧ q) : (r → p) ∧ (r → q) := by
  constructor
  · intro hr
    exact (hrpq hr).left
  · intro hr
    exact (hrpq hr).right
```

Currification

The actual universal elimination rule of `And` is the so-called *decurrification*: From `(p → q → r)` we may deduce `(p ∧ q → r)`. This is actually a logical equivalence.

Intuitively, requiring both `p` and `q` to deduce `r` is nothing but requiring `p` to deduce that `q` is sufficient to deduce `r`.

[IGNORE] Decurrification is also self-evidently true in Lean's dependent type theory.

Currification is heavily used in functional programming for its convenience, Lean is no exception.

You are no stranger to decurrification even if you are not a functional programmer: The *universal property of the tensor product of modules* says exactly the same.

$$\mathrm{Hom}(M \otimes N, P) \cong \mathrm{Hom}(M, \mathrm{Hom}(N, P))$$

[EXR] currification

```
example (h : p ∧ q → r) : (p → q → r) := by
  intro hp hq
  exact h ⟨hp, hq⟩
```

[EXR] decurrification

```
example (h : p → q → r) : (p ∧ q → r) := by
  intro hpq
  exact h hpq.left hpq.right

example (h : p → q → r) : (p ∧ q → r) := by
  intro ⟨hp, hq⟩ -- `intro` is smart enough to destructure `And`
  exact h hp hq

example (h : p → q → r) : (p ∧ q → r) := by
  intro ⟨hp, hq⟩
  apply h -- `apply` is smart enough to auto-decurrify and generate two subgoals
  · exact hp
  · exact hq
```

[IGNORE] decurrification actually originates from `And.rec`, which is self-evident

```
#check And.rec
theorem decurrify (h : p → q → r) : (p ∧ q → r) := And.rec h
```

`And.left` is actually a consequence of decurrification

```
example : p ∧ q → p := by
  apply decurrify
  intro hp _
  exact hp
```

3.1.2 `Iff` (↔)

It's high time to introduce `Iff` here.

`Iff` (↔) contains two side of implications: `Iff.mp` and `Iff.mpr`.

Though it is defined as a distinct inductive type, `Iff` may be seen as a bundled version of `(p → q) ∧ (q → p)`. you may, somehow, even use it like a `(p → q) ∧ (q → p)`. The only major difference is the name of the two components.

```
#check Iff.intro
#check Iff.mp
#check Iff.mpr

example : (p ↔ q) ↔ (p → q) ∧ (q → p) := by
  constructor
  · intro h
    exact ⟨h.mp, h.mpr⟩
  · intro ⟨hpq, hqp⟩
    exact ⟨hpq, hqp⟩
```

3.1.3 `Or` (∨)

Introducing `Or`

`Or` has two constructors, `Or.inl` and `Or.inr`. Either a proof of `p` or a proof of `q` produces a proof of `p ∨ q`.

```
#check Or.inl
#check Or.inr

example (hp : p) : p ∨ q := Or.inl hp
```

`left` (resp. `right`) tactic reduce `Or` goals to `p` (resp. `q`)

```
example (hq : q) : p ∨ q := by
  right
  exact hq
```

Eliminating `Or`

To prove `r` from `p ∨ q`, it suffices to prove both `p → r` and `q → r`. This is the elimination rule of `Or`, or the *universal property* of the direct sum.

```
#check Or.elim
#check Or.rec -- [IGNORE]

example (hpr : p → r) (hqr : q → r) : (p ∨ q → r) := fun hpq ↦ (Or.elim hpq hpr hqr)
example (hpr : p → r) (hqr : q → r) : (p ∨ q → r) := (Or.elim · hpr hqr) -- note the use of `·`
example (hpr : p → r) (hqr : q → r) (hpq : p ∨ q) : r := by
  apply Or.elim hpq
  · exact hpr
  · exact hqr
```

`match`-style syntax is designed to make use of `Or.elim` to destructure `Or` to cases. [IGNORE] You may just skim through this syntax for now.

```
example (hpr : p → r) (hqr : q → r) : (p ∨ q → r) := fun
  | Or.inl hp => hpr hp
  | Or.inr hq => hqr hq
example (hpr : p → r) (hqr : q → r) (hpq : p ∨ q) : r :=
  match hpq with
  | Or.inl hp => hpr hp
  | Or.inr hq => hqr hq
example (hpr : p → r) (hqr : q → r) (hpq : p ∨ q) : r := by
  match hpq with
  | Or.inl hp => exact hpr hp
  | Or.inr hq => exact hqr hq
example (hpr : p → r) (hqr : q → r) (hpq : p ∨ q) : r := by
  cases hpq with
  | inl hp => exact hpr hp
  | inr hq => exact hqr hq
```

`rcases` may also serve as a tactic version of `match`, which is much more convenient.

```
example (hpr : p → r) (hqr : q → r) (hpq : p ∨ q) : r := by
  rcases hpq with (hp | hq) -- `rcases` can also destructure `Or`
  · exact hpr hp
  · exact hqr hq
example (hpr : p → r) (hqr : q → r) : p ∨ q → r := by
  rintro (hp | hq) -- `rintro` is a combination of `intro` and `rcases`
  · exact hpr hp
  · exact hqr hq
```

[EXR] distributive laws

```
example : p ∧ (q ∨ r) ↔ (p ∧ q) ∨ (p ∧ r) := by sorry
example : p ∨ (q ∧ r) ↔ (p ∨ q) ∧ (p ∨ r) := by sorry

end
```

3.2 Forall and Exists

3.2.1 Forall (∀)

As you may have already noticed, `∀` is just an alternative way of writing `→`. Say `p` is a predicate on a type `X`, i.e. of type `X → Prop`, then `∀ x : X, p x` is exactly the same as `(x : X) → p x`.

Though `→` is primitive in Lean's dependent type theory, we may still (perhaps awkwardly) state the introduction and elimination rules of `∀`:

- Introduction: `fun (x : X) ↦ (h x : p x)` produces a proof of `∀ x : X, p x`.
- Elimination: Given a proof `h` of `∀ x : X, p x`, we can obtain a proof of `p a` for any specific `a : X`. It is exactly `h a`.

```
section

variable {X : Type} (p q : X → Prop) (r s : Prop) (a b : X)

#check ∀ x : X, p x
#check ∀ x, p x -- Lean is smart enough to infer the type of `x`

example : (∀ x : X, p x) → p a := by
  intro h
  exact h a
```

[IGNORE] Writing `∀` emphasizes that the arrow `→` is of dependent type, and the domain `X` is a type, not a proposition. But they are just purely psychological, as the following examples show.

```
example : (hrs : r → s) → (∀ _ : r, s) := by
  intro hrs
  exact hrs
```

3.2.2 Exists (∃)

`∃` is a bit more complicated.

Slogan: `∀` is a dependent `→`, `∃` is a dependent `×` (or `∧` in `Prop` universe)

```
#check ∃ x : X, p x
#check ∃ x, p x -- Lean is smart enough to infer the type of `x`
```

Introducting `Exists`

`∃ x : X, p x` means that we have the following data:

- an element `a : X`;
- a proof `h : p a`.

So a pair `(a, h)` would suffice to construct a proof of `∃ x : X, p x`. This is the defining introduction rule of `Exists` as an inductive type.

```
#check Exists.intro
example (a : X) (h : p a) : ∃ x, p x := Exists.intro a h
```

As like `And`, you may use the anonymous constructor notation `⟨a, h⟩` to mean `Exists.intro a h`.

```
example (a : X) (h : p a) : ∃ x, p x := ⟨a, h⟩
```

In tactic mode, `use a` make use of `Exists.intro a` to reduce the goal `∃ x : X, p x` to `p a`.

```
example (a : X) (h : p a) : ∃ x, p x := by use a

-- [EXR]
example (x y z : ℕ) (hxy : x < y) (hyz : y < z) : ∃ w, x < w ∧ w < z :=
  ⟨y, ⟨hxy, hyz⟩⟩
```

Note that in the defining pair `(a, h)`, `h` is a proof of `p a`, whose type depends on `a`. Thus psychologically, you may view `∃ x : X, p x` as a dependent pair type `(x : X) × (p x)`.

Have writing `Exists` as a dependent pair type reminded you of the currification process?

Eliminating `Exists`

To construct the implication `(∃ x : X, p x) → q`, it suffices to have a proof of `(∀ x : X, p x → q)`, i.e. `(x : X) → p x → q`. `Exists.elim` does exactly above.

```
#check Exists.elim

example : (∀ x, p x → r) → ((∃ x, p x) → r) := by
  intro hf he
  exact Exists.elim he hf
```

In tactic mode, `rcases h with ⟨a, ha⟩` make use of this elimination rule to break down a hypothesis `h : ∃ x : X, p x` into a witness `a : X` and a proof `ha : p a`.

```
example : (∀ x, p x → r) → ((∃ x, p x) → r) := by
  intro hf he
  rcases he with ⟨a, hpa⟩
```

```
  exact hf a hpa

example : (∀ x, p x → r) → ((∃ x, p x) → r) := by
  intro h ⟨a, hpa⟩ -- you may also `rcases` implicitly
  exact h a hpa
```

[EXR] reverse direction is also true

```
example :  ((∃ x, p x) → r) → (∀ x, p x → r) := by
  intro h a hpa
  apply h
  use a

-- [EXR]
example : (∃ x, r ∧ p x) → r ∧ (∃ x, r ∧ p x) := by
  intro ⟨a, ⟨hr, hpa⟩⟩
  exact ⟨hr, ⟨a, ⟨hr, hpa⟩⟩⟩

-- [EXR]
example : (∃ x, p x ∨ q x) ↔ (∃ x, p x) ∨ (∃ x, q x) := by
  constructor
  · rintro ⟨a, (hpa | hqa)⟩
    · left; use a
    · right; use a
  · rintro (⟨a, hpa⟩ | ⟨a, hqa⟩)
    · use a; left; exact hpa
    · use a; right; exact hqa

end
```

[IGNORE] A cosmological remark

The pair `(a, h)` actually do not have type `(x : X) × (p x)`. The latter notation is actually for the *dependent pair type* (or `Sigma` type), which lives in `Type*` universe.

But `Exists` should live in `Prop`, and in `Prop` universe we admit *proof-irrelevance*, i.e. we do not save data. So `Exists` forget the exact witness `a` once it is proved.

This "forgetfulness" is revealed by the fact that there is no elimination rule `Exists.fst` to extract the witness `a` from a proof of `∃ x : X, p x`, as long as `X` lives in the `Type*` universe. (Note that `Exists.elim` can only produce propositions in `Prop`)

But if `X` lives in `Prop` universe, then we do have `Exists.fst`:

```
section

#check Exists.fst
```

Wait, wait, we never worked with `X : Prop` before. Say `p : r → Prop` and `r s : Prop`, what does `∃ hr : r, p hr` mean? It means that `r` and `p hr` are both true? [TODO] I don't know how to explain this properly so far.

```
variable (r : Prop) (p : r → Prop)
#check ∃ hr : r, p hr

-- Prove `Exists.fst` and `Exists.snd` by `Exists.elim`
example (he : ∃ hr : r, p hr) : r ∧ p he.fst := by
  apply Exists.elim he
  intro hr hpr
  exact ⟨hr, hpr⟩

end
```

3.3 [IGNORE] A cosmological remark, continued

Same construction, different universes. Other examples are also shown below.

```
#print And -- `×` in `Prop`
#print Prod -- `×` in `Type*`

-- Forall `∀`: dependent `∏` in `Prop`
-- dependent function type: dependent `∏` in `Type*`

#print Or -- `⊕` in `Prop`
#print Sum -- `⊕` in `Type*`

#print Exists -- dependent `∑` in `Prop`
#print Sigma -- dependent `Σ` in `Type*`

#print Nonempty -- a proof of non-emptiness living in `Prop`
#print Inhabited -- an designated element living in `Sort*`
```

Chapter 4

Logic (Part III)

1. `True`, `False` and `Not`
2. classical logic tactics, e.g. proof by contradiction
3. negation-pushing techniques
4. the difference between classical and intuitionistic logic
5. `Decidable`

3 is recommended for those who wants to have some exercises. For lazy ones, you may only remember the tactics introduced there.

4, 5 are optional and left for logical lunatics.

```
import Mathlib
```

4.1 `True`, `False` and `Not`

In Lean's dependent type theory, `True` and `False` are propositions serving as the *terminal and initial objects* in the universe of `Prop`.

Eagle-eyed readers may notice that `True` and `False` act similarly to singleton sets and empty sets in set theory.

They are constructed as *inductive types*.

```
section

variable (p q : Prop)
```

4.1.1 `True` (`⊤`)

`True` has a single constructor `True.intro`, which produces the unique proof of `True`. `True` is self-evidently true by `True.intro`.

```
#check True.intro
```

`True` as the terminal object

```
example : p → True := by
  intro _
  exact True.intro
```

The following examples shows that `True → p` is logically equivalent to `p`.

```
example (hp : p) : True → p := by
  intro _
  exact hp
```

[IGNORE] Above is actually the elimination law of `True`.

```
example (hp : p) : True → p := True.rec hp

example (htp : True → p) : p := htp True.intro
```

`trivial` is a tactic that solves goals of type `True` using `True.intro`, though it's power does not stop here.

```
example (htp : True → p) : p := by
  apply htp
  trivial
```

4.1.2 `False` (⊥)

`False` has no constructors, meaning that there is no way to construct a proof of `False`. This means that `False` is always false.

`False.elim` is the eliminator of `False`, serve as the "principle of explosion", which allows us to derive anything from a falsehood. `False.elim` is self-evidently true in Lean's dependent type theory.

```
#check False.elim
#check False.rec -- [IGNORE] `False.elim` is actually defined as `False.rec`
```

eliminating `False`

```
example (hf : False) : p := False.elim hf
```

`exfalso` is a tactic that applys `False.elim` to the current goal, changing it to `False`.

```
example (hf : False) : p := by
  exfalso
  exact hf
```

`contradiction` is a tactic that proves the current goal by finding a trivial contradiction in the context.

```
example (hf : False) : p := by
  contradiction

-- [EXR]
example (h : 1 + 1 = 3) : RiemannHypothesis := by
  contradiction
```

On how to actually obtain a proof of `False` from a trivially false hypothesis via term-style proof [TODO], see here

[IGNORE] Experienced audiences may question why `False.elim` lands in `Sort*` universe instead of `Prop`. This is because `False` is a *subsingleton*. See the manual to understand how the universe of a recursor is determined.

```

end
```

4.2 Not (¬)

In Lean's dependent type theory, negation `¬p` is realized as `p → False`

You may understand `¬p` as "if `p` then absurd", indicating that `p` cannot be true.

```

section

variable (p q : Prop)

#print Not

example (hp : p) (hnp : ¬p) : False := hnp hp
#check absurd -- above has a name
```

[EXR] contraposition

```
example : (p → q) → (¬q → ¬p) := by
  intro hpq hnq hp
  exact hnq (hpq hp)
```

`contrapose!` is a tactic that does exactly this. We shall discuss this later.

[EXR]

```
example : ¬True → False := by
  intro h
  exact h True.intro
```

[EXR]

```
example : ¬False := by
  intro h
  exact h
```

[EXR] double negation introduction

```
example : p → ¬¬p := by
  intro hp hnp
  exact hnp hp
```

Double negation elimination is not valid in intuitionistic logic. You'll need *proof by contradiction* `Classical.byContradiction` to prove it. The tactic `by_contra` is created for this purpose. If the goal is `p`, then `by_contra hnp` changes the goal to `False`, and adds the hypothesis `hnp : ¬p` into the context.

```
#check Classical.byContradiction
```

double negation elimination

```
example : ¬¬p → p := by
  intro hnnp
  by_contra hnp
  exact hnnp hnp
```

You can use the following command to check what axioms are used in the proof

```
#print axioms Classical.not_not -- above has a name
```

For logical lunatics:

In Lean, `Classical.byContradiction` is proved by the fact that all propositions are `Decidable` in classical logic, which is a result of - the *axiom of choice* `Classical.choice` - the *law of excluded middle* `Classical.em`, which is a result of - the axiom of choice `Classical.choice` - *function extensionality* `funext`, which is a result of - the quotient axiom `Quot.sound` - *propositional extensionality* `propext`

You can always trace back like this in Lean, by ctrl-clicking the names. This is a reason why Lean is awesome for learning logic and mathematics.

[EXR] another side of contraposition

```
example : (¬q → ¬p) → (p → q) := by
  intro hnqnp hp
  by_contra hnq
  exact hnqnp hnq hp

end
```

[IGNORE] In fact above is equivalent to double negation elimination. This one use the `have` tactic, which allows us to state and prove a lemma in the middle of a proof.

```
example (hctp : (p q : Prop) → (¬q → ¬p) → (p → q)) : (p : Prop) → (¬¬p → p) := by
  intro p hnnp
  have h : (¬p → ¬True) := by
    intro hnp _
    exact hnnp hnp
  apply hctp True p h
  trivial
```

4.3 Pushing negations

Some negation can be pushed within intuitionistic logic. Some cannot.

4.3.1 Negation with ∧ and ∨

```
section

variable (p q r : Prop)
```

Classical logic: case analysis

```
example (hpq : p → q) (hnpq : ¬p → q) : q := Or.elim (Classical.em p) hpq hnpq
#check Classical.byCases -- above has a name
```

We have a corresponding tactic: `by_cases`

```
example (hpq : p → q) (hnpq : ¬p → q) : q := by
  by_cases hp : p
  · exact hpq hp
  · exact hnpq hp
```

Proof by cases would help us to obtain an equivalent characterization of `Or`.

```
example : (p ∨ q) ↔ (¬p → q) := by
  constructor
  · rintro (hp | hq)
    · intro hnp
      exfalso
      exact hnp hp
    · intro _
      exact hq
  · intro hnpq  -- the direction of constructing `Or` needs classical logic
    by_cases h?p : p
    · left; exact h?p
    · right; exact hnpq h?p
```

Note that this vividly illustrates the difference between classical logic and intuitionistic logic.

In intuitionistic logic, `Or` means slightly stronger than in classical logic: by `p ∨ q` we mean that we know explicitly which one of `p` and `q` is true. We cannot do implications like `¬p → q` implying `p ∨ q`, because we don't know exactly which one of `p` and `¬p` is true, and the introduction rules of `Or` are asking us to provide it explicitly. This is a reason why intuitionistic logic is considered to be computable.

We also have an equivalent characterization of `And`. This is also done in classical logic.

```
example : (p ∧ q) ↔ ¬(p → ¬q) := by
  constructor
  · intro ⟨hp, hnq⟩ hpnq
    exact hpnq hp hnq
  · intro hnpnq -- the direction of constructing `And` needs classical logic
    contrapose hnpnq
    rw [Classical.not_not]
    intro hp hq
    exact hnpnq ⟨hp, hq⟩
```

[EXR] `→`–`∨` distribution

```
example : (r → p ∨ q) ↔ ((r → p) ∨ (r → q)) := by
  constructor
  · intro hrpq -- this direction needs classical logic
    by_cases h?r : r
    · rcases hrpq h?r with (hp | hq)
      · left; intro _; exact hp
      · right; intro _; exact hq
    · left
      intro hr
      exfalso; exact h?r hr
  · rintro (hrp | hrq)
    · intro hr
      left; exact hrp hr
    · intro hr
      right; exact hrq hr
```

```
#check imp_or -- above has a name
```

[EXR] De Morgan's laws

```
example : ¬(p ∨ q) ↔ ¬p ∧ ¬q := by
  constructor
  · intro hnq
    constructor
    · intro hp
      apply hnq
      left; exact hp
    · intro hq
      apply hnq
      right
      exact hq
  · rintro ⟨hnp, hnq⟩ (hp | hq)
    · exact hnp hp
    · exact hnq hq
#check not_or -- above has a name
```

[EXR] De Morgan's laws

```
example : ¬(p ∧ q) ↔ ¬p ∨ ¬q := by
  constructor
  · intro hnpq -- this direction needs classical logic
    by_cases h?p : p
    · right
      intro hq
      apply hnpq
      exact ⟨h?p, hq⟩
    · left
      exact h?p
  · rintro (hnp | hnq) ⟨hp, hq⟩
    · exact hnp hp
    · exact hnq hq
#check not_and -- above has a name
```

Introducing `push_neg` tactic: automatically proves all the above. It works in classical logic where *negation normal forms* exist.

`by_contra!`, `contrapose!` are `push_neg`-enhanced version of their non-`!` counterparts.

For more exercises, see Propositions and Proofs - TPiL4

```


end
```

4.4 [IGNORE] `Decidable`

It's high time to introduce `Decidable` here for the first time.

Mathematicians are often aware of intuitionistic logic. They know classical logic is equipped with `Classical.em: p ∨ ¬p` for any proposition `p`. Though rarely do they know the concept of `Decidable`, which more often appears in the theory of computation.

For short, `Decidable p` means exactly the same as `p ∨ ¬p` in intuitionistic logic. It means that we know explicitly (or computationally) which one of `p` and `¬p` is true.

Though formally in Lean, `Decidable` is defined as a distinct inductive type, it is very similar to `Or` in that you may, somehow, even use it like a `p ∨ ¬p`. But there are major differences. They are:

- [IGNORE] `Decidable` lives in `Type` universe, instead of `Prop` universe.

  In Lean's dependent type theory, things in `Prop` universe are allowed to be non-constructive. This is because in `Prop` universe, proofs are *proof-irrelevant*: Lean forgets the exact proof of a proposition once it is proved. So when we have an `Or`, we actually have no idea which one of the two sides is true. Lean is designed so, probably because most of the mathematics is non-constructive.

  On the other hand, things in `Type` universe are required to be constructive, unless you have used `Classical.choice` (In such situation, Lean will require you to tag it as `noncomputable`).

  `Decidable` is designed to be constructive, because it is used to decide whether a proposition is true or false by computation. So `Decidable` must live in `Type` universe: To save whether `p` or `¬p` is true.

  In short, `Prop` is non-constructive and proof-irrelevant, while `Type` is constructive and saves data. This makes `Decidable` stronger than a pure proof of `p ∨ ¬p : Prop`.

- [IGNORE] It is tagged as a typeclass.

  This allows Lean to automatically find a proof of `Decidable p` so that you don't have to prove it yourself.

  So at many places `Decidable p` is implicitly deduced.

- The constructors of `Decidable` has different names: `isTrue` and `isFalse`

To wrap up, we have `Decidable` because:

- To mean exactly the same as `p ∨ ¬p` in intuitionistic logic, to make it computable.

- To allow you to just assume `p ∨ ¬p` for only some propositions, which is more flexible than a classical logic overkill.

```
section

variable (p q : Prop)

#print Decidable
#check Decidable.isTrue
#check Decidable.isFalse
```

`Decidable` enables computational reasoning to see if a proposition is true or false

```
#eval True
#eval True → False
#eval False → (1 + 1 = 3)
#synth Decidable (False → (1 + 1 = 3))
```

Manually proving `Decidable` to ensures a computable proof

```
instance : Decidable (p → p ∨ q) := by
  apply Decidable.isTrue -- explicit use of constructor
  intro hp
  left
  exact hp
#synth Decidable (p → p ∨ q)
#eval (p q : Prop) → (p → (p ∨ q))
```

`Decidable` enables partial classical logic

```
#check Classical.byContradiction -- we have done this before
```

proof by contradiction in intuitionistic logic with decidable hypothesis

```
example [dp : Decidable p] : (¬p → False) → p := by
  intro hnpn
  rcases dp with (hnp | hp)
  · exfalso; exact hnpn hnp
  · exact hp
#check Decidable.byContradiction -- above has a name

end
```

Part III

Type

Chapter 5

Type and Equality

In the previous chapter, we have seen that propositions are types in the `Prop` universe. In this chapter, we shall move up to the `Type*` universe, and see how the most fundamental notion in mathematics, equality, works there.

- Numbers
- Universe hierarchy
- Equality `Eq` (`=`)
  - Arithmetic in `CommRing`
- Defining terms and functions
  - Definitional equality vs propositional equality

```
import Mathlib
```

5.1 Numbers

Lean and Mathlib have many built-in types for numbers, including

```
#check ℕ
#check ℤ
#check ℚ
#check ℝ
#check ℂ
```

There are some built-in ways to represent numbers. Lean interprets their types accordingly, like every programming language does.

```
#check 3
#check 3.14
#check (22 / 7 : ℚ)
#check Real.pi
#check Complex.log (-1) / Complex.I
```

Do note that numbers in different types work differently. Sometimes you need to explicitly specify the type you want.

```
#eval 22 / 7
#eval (22 : ℚ) / 7
#eval (22 / 7 : ℚ)
```

You may not `#eval (22 : ℝ) / 7` because ℝ is not computable. It's defined using Cauchy sequences of rational numbers. For `Float` computation you may use `Float` type.

```
#eval (22 : Float) / 7
```

Strange as it may seem, this type checks.

```
#check (Real.sqrt 2) ^ 2 = (5 / 2 : ℕ)
```

Note how the type of a number is interpreted and *implicitly coerced*.

Coercions are automatic conversions between types. It somewhat allows us to abuse notations like mathematicians always do. Detailing coercions would be another ocean of knowledge. We shall stop here for now.

5.2 Universe hierarchy

If everything has a type, what is the type of a type?

```
#check 3
#check Nat
#check Type
#check Type 1
#check Type 2

#check 1 + 2 = 3
#check Prop
```

Lean has a hierarchy of universes:

```
   ...
    ↓
Type 3            = Sort 4
    ↓
Type 2            = Sort 3
    ↓
Type 1            = Sort 2
    ↓
 Type   = Type 0  = Sort 1
    ↓
 Prop   =  Sort   = Sort 0
```

- `Prop` is the universe of logical propositions.
- `Type` is the universe of most of the mathematical objects.

At most times, you don't need to care about universe levels above `Type`. But do recall the critical difference between `Prop` and `Type`:

Terms in `Prop`, i.e. proofs, are *proof-irrelevant*, i.e. all proofs of the same proposition are considered equal, while terms in `Type` are distinguishable in general. This allows classical reasoning in `Prop`, and computation in `Type`.

You may explore more on this in the previous logic chapters.

5.2.1 Remark

Two questions arise naturally here:

- Why `Prop` is separated from `Type`?

  This is answered by the need of proof irrelevance.

- Why `Prop` is at the bottom of the hierarchy?

  We come up with two explanations ([TODO] discussions are welcome!):

  - `Prop` is often compared to `Bool : Type`. This analogy validates the `Prop : Type` convention.

    `Bool` has two values `true` and `false`, representing truth values, acting as a switch. `Prop` may be viewed as a non-computatble version of `Bool`, switching by whether a proposition is true or false. e.g. In Mathlib, a subset of `α` is defined as a predicate `α → Prop`, a relation on `α` is defined as `α → α → Prop`, etc. But all of these are non-computable. e.g. you cannot define a computable function by this switch.
  - On determining universe levels of predicates and functions.

    For a function `α → β` where `α : Type u` and `β : Type v`, its should live in `Type (max u v)` naturally. But recall `∀` and `∃` quantifiers from logic. They eat `α → Prop` functions to produce propositions, living in `Prop`. This means that `Prop` should be larger than any `Type u` to accommodate such functions. As a convention, we put `Prop` at the bottom of the hierarchy to reflect this. The true universe level of a function is `imax u v` if it maps from `Sort u` to `Sort v`, where `imax` is the regular `max` except that `imax u 0 = imax 0 u = 0` for any `u`.

5.3 Equality `Eq` (=)

Equality is a fundamental notation in mathematics, but also a major victim of *abuse of notation*. Though trained experts can usually tell from context what kind of equality is meant, things still become hopelessly confusing from time to time.

In set theory, by axiom of extensionality, two sets are equal if and only if they have the same elements.

In Lean's type theory, we distinguish between different equalities:

- *Definitional equality*
- *Propositional equality* (`Eq`, i.e. `=`)
- *Heterogeneous equality* (We shall not touch this)

We shall now show the basic usage of `=` in Lean, mostly in tactic mode. We detail a little on the difference between definitional and propositional equality afterwards. The real, full discussion of equality is only accessible with enough knowledge of inductive types.

```
section
```

`Eq` takes two terms of the same type (up to definitional equality), and produces a proposition in `Prop`. For terms `a b : α`, the proposition `a = b` means that `a` and `b` are equal. Do note that types of `a` and `b` must be the same, i.e. definitionally equal.

```
#check Eq
#check 1 + 1 = 3
-- #check 1 + 1 = Nat -- this won't compile. Eq requires both sides to have the same type.
```

5.3.1 Handling equality

```
variable (a b c : ℚ)
```

The most basic way to show an equality is by tactic `rfl`: LHS is definitionally equal to RHS.

```
example : a = a := rfl
```

Note that `rfl` works for not only literally-the-same terms, but also definitionally equal terms. We'll detail definitional equality afterwards.

`rw` is a tactic that rewrites a goal by a given equality.

```
example (f : ℚ → ℚ) (hab : a = b) (hbc : b = c) : f a = f c := by
  rw [hab, hbc]
```

you may also apply the equality in the reverse direction

```
example (f : ℚ → ℚ) (hab : b = a) (hbc : b = c) : f a = f c := by
  rw [← hab, hbc]
```

You may also use `symm` tactic to swap an equality

```
#help tactic symm
example (f : ℚ → ℚ) (hab : b = a) (hbc : b = c) : f a = f c := by
  symm at hab
  rw [hab, hbc]
```

or swap at the goal

```
example (f : ℚ → ℚ) (hab : b = a) : f a = f b := by
  symm
  rw [hab]
```

You may also rewrite at a hypothesis.

```
example (hab : a = b) (hbc : b = c) : a = c := by
  rw [hbc] at hab
  exact hab
```

`congr` tactic reduces the goal `f a = f b` to `a = b`.

```
#help tactic congr
example (f : ℚ → ℚ) (hab : a = b) (hbc : b = c) : f a = f c := by
  congr
  rw [hab, hbc]
```

5.3.2 Working in `CommRing`

Let's do some basic rewrites in commutative rings, e.g. `ℚ`.

Commutativity and associativity

```
#check add_comm
example : a + b = b + a := by rw [add_comm]

#check add_assoc
example : (a + b) + c = a + (b + c) := by rw [add_assoc]

#check mul_comm
example : a * b = b * a := by rw [mul_comm]

#check mul_assoc
example : (a * b) * c = a * (b * c) := by rw [mul_assoc]
```

Sometimes you need to specify the arguments to narrow down possible targets for `rw`.

```
example : (a + b) + c = (b + a) + c := by
  rw [add_comm a b]
```

[EXR] You may chain multiple rewrites in one `rw`.

```
example : (a + b) + c = a + (c + b) := by
  rw [add_assoc, add_comm b c]

-- [EXR]
example : a + b + c = c + a + b := by
  rw [add_comm, add_assoc]

#check mul_add
example : (a + b) * c = c * a + c * b := by
  rw [mul_comm, mul_add]

-- [EXR]
example : (a + b) * (c + b) = a * c + a * b + b * c + b * b := by
  rw [add_mul, mul_add, mul_add, ← add_assoc]
```

Zero and one

```
#check add_zero
example : a + 0 = a := by rw [add_zero]
#check zero_add
example : 0 + a = a := by rw [zero_add]

#check mul_one
example : a * 1 = a := by rw [mul_one]
#check one_mul
example : 1 * a = a := by rw [one_mul]

-- [EXR]
example : 1 * a + (0 + b) * 1 = a + b := by
   rw [one_mul, zero_add, mul_one]
```

[EXR] uniqueness of zero

```
example (o : ℚ) (h : ∀ x : ℚ, x + o = x) : o = 0 := by
  specialize h 0
  rw [zero_add] at h
  exact h
```

Subtraction

transposition

```
#check add_sub_assoc
#check sub_self
#check add_zero
```

```
example (h : c = a + b) : c - b = a := by
  rw [h, add_sub_assoc, sub_self, add_zero]
```

Automation

Had enough of these tedious rewrites? Automation makes your life easier.

`simp (at h)` tactic eliminates `0` and `1` automatically. `simp?` shows you what lemmas `simp` used.

```
#help tactic simp
example : c + a * (b + 0) = a * b + c := by
  simp
  rw [add_comm]
```

`ring` tactic is even stronger: it reduces LHS and RHS to a canonical form (it exists in any commutative ring) to solve equalities automatically. `ring_nf (at h)` reduces the expression `h` to its canonical form.

```
#help tactic ring -- check out the documentation
example : (a + 1) * (b + 2) = a * b + 2 * a + b + 2 := by
  ring
```

`apply_fun at h` tactic applies a function to both sides of an equality hypothesis `h`. Combined with `simp` and `ring`, it make transpotions easier.

```
#help tactic apply_fun
example (h : a + c = b + c) : a = b := by
  apply_fun (fun x ↦ x - c) at h
  simp at h
  exact h
```

[EXR] transposition again

```
example (h : c = a + b) : c - b = a := by
  apply_fun (fun x ↦ x - b) at h
  simp at h
  exact h
```

A remark on type classes

Wondering how Lean knows that commutativity, associativity, distributivity, etc. hold for `ℚ`? Wondering how Lean knows `a * 1 = a` and has relevant lemmas for that? This is because Lean knows that `ℚ` is an commutative ring. This is because in Mathlib, `ℚ` has been registered as an instance of the typeclass `CommRing`. So that once you `import Mathlib`, Lean automatically knows about the `CommRing` structure of `ℚ`. We might learn about typeclasses later in this course.

```
#synth CommRing ℚ -- Checkout the `CommRing` instance that Mathlib provides for `ℚ`
```

5.3.3 funext and propext

There are several ways to show a (propositional) equality other than `rfl` and `rw`.

Functional extensionality `funext` states that two functions are equal if they give equal outputs for every input.

It's a theorem in Lean's type theory, derived from the quotient axiom `Quot.sound`.

```
#check funext
example (f g : ℚ → ℚ) (h : ∀ x : ℚ, f x = g x) : f = g := funext h
```

It has a tactic version `ext` / `funext` as well

```
#help tactic funext
example (f g : ℚ → ℚ) (h : ∀ x : ℚ, f x = g x) : f = g := by
  funext x
  exact h x
```

Propositional extensionality `propext` states that two propositions are equal if they are logically equivalent. It's admitted as an axiom in Lean.

```
#check propext
example (P Q : Prop) (h : P ↔ Q) : P = Q := propext h
```

It has a tactic version `ext` as well

```
#help tactic ext
example (P Q : Prop) (h : P ↔ Q) : P = Q := by
  ext
  exact h
```

This allows you to `rw` an `iff` (↔) like an equality (`=`).

```
example (P Q : Prop) (h : P ↔ Q) : P = Q := by
  rw [h]
```

5.4 Definitions, and definitional equality

We now come back to detail a little on the exact power of `rfl`, i.e. what is the meaning of definitional equality.

First, we show how to define terms and functions in Lean.

5.4.1 Global definitions

Recall that you may use `def` to define your own terms.

```
def myNumber : ℚ := 998244353
#check myNumber
```

`def` can also define functions.

```
#check fun (x : ℚ) ↦ x * x
def square (x : ℚ) : ℚ := x * x
def square' : ℚ → ℚ := fun x ↦ x * x
#print square
#print square'
```

Be open minded: you may even use tactic mode to define terms!

```
def square'' : ℚ → ℚ := by
  intro x
  exact x * x
#print square''

def square_myNumber : ℚ := by
  apply square
  exact myNumber
#print square_myNumber
```

5.4.2 Local definitions

You may also define local terms and functions using `let`. It may be used in both term mode and tactic mode.

```
#help tactic let

example : ℚ := by
  let a : ℚ := 3
  let b : ℚ := 4
  exact square (a + b)

example : ℚ :=
  let a : ℚ := 3
  let b : ℚ := 4
  square (a + b)

example : let a := 4; let b := 4; a = b := rfl
```

Sometimes you want an alias for a complex term. `set` tactic is a variant of `let` that automatically replaces all occurrences of the defined term.

```
#help tactic set

example (a b c : ℕ) : 0 = a + b - (a + b) := by
  set d := a + b
  simp
```

It's crucial to distinguish between `let` and `have`: `let` saves the term of the definition for later use, but `have` is "opaque": it won't let you unfold the definition later. Thus naturally, `let` is often used for `Type*`s, and `have` is used for `Prop`s.

```
example : 3 = 3 := by
  let a := 3
  let b := 3
  have h : a = b := rfl
  exact h

example : 3 = 3 := by
  have a := 3
  have b := 3
  -- have h : a = b := rfl
  sorry -- above won't compile
```

[TODO] Explain why it works here.

```
example : have a := 3; have b := 3; a = b := rfl
```

5.4.3 Unfolding definitions

To manually unfold a definition in the tactic mode, you may use the `rw (at h)` tactic or the `unfold (at h)` tactic.

```
#help tactic unfold
example : square myNumber = 998244353 * 998244353 := by
  rw [square]
  unfold myNumber
  rfl
```

For local (non-`have`) definitions, you may use `unfold` as well. Though sadly `rw` does not work for local definitions for now.

```
example (a b : ℕ): (a + b) - (a + b) = 0 := by
  set d := a + b
  unfold d
  simp
```

Luckily, `have`, `let` and `set` all allows you to obtain a propositional equality when defining. (Technically this is not an unfolding, though.)

```
example (a b : ℕ) : (a + b) - (a + b) = 0 := by
  let (eq := h1) d1 := a + b
  have (eq := h2) d2 := a + b
  set d3 := a + b with h3
  simp
```

5.4.4 Definitional equality vs propositional equality

[IGNORE] Skip this if you find it confusing for the first time. You can recall this when we deal with quotient types.

Definitional equality means that two terms are the same by definition (i.e. they reduce to the same form).

- `def`, `theorem`-like commands
- Applications of functions

are examples of definitional equalities.

It is a meta-level concept, it cannot be stated as a proposition.

`rfl`

As the sole constructor of propositional equality, `rfl` proves a definitional equality.

```
#check rfl
```

Note that `myNumber` is definitionally equal to `998244353`.

```
example : myNumber = 998244353 := rfl
```

`rfl` can even solve simple evaluations, because both sides reduce to `8` by the (inductive) definition of arithmetic operations over `ℕ`.

```
example : 5 + 3 = 2 * 2 * 2 := rfl
```

`rfl` also has a tactic version. This tactic works for logical equivalences (`↔`) as well, as `Iff.rfl` does.

```
#help tactic rfl

example : True ↔ True := by rfl
```

`dsimp` is a weaker version of `simp`, which only applies (obvious) definitional equalities to simplify an expression.

```
#help tactic dsimp

example (a b c : ℕ) : 0 + a = a - (a + 0) + a := by
  dsimp
  simp
```

These are some non-examples for definitional equality. They are only propositionally equal, by `propext` and logical equivalence.

```
-- example (p : Prop) : True ↔ (p → True) := by rfl
-- example True ↔ ¬ False := by rfl
```

Type checking

Type checking is determined up to definitional equality.

In fact, it's the sole responsibility of Lean's compiler to check definitional equalities.

An failure of definitional equality results in a type error. That is, it is regarded as invalid Lean code.

```
def myType := ℚ
```

This won't compile, because Lean do not know a coercion of `ℕ → myType`.

```
-- def myTypeNumber := (998244353 : myType)
```

This passes the type check. because we manually build a bridge here: Lean knows the coercion `ℕ → ℚ` and that `myType` is definitionally equal to `ℚ`.

```
def myTypeNumber : myType := (998244353 : ℚ)
#check myTypeNumber
```

This also passes the type check for the same reason.

```
#check myTypeNumber = myNumber
```

The type of `myNumber : ℚ` and `myTypeNumber : myType` are definitionally equal, thus the equality passes the type check. Their values are also definitionally equal, so you can prove their equality by `rfl`.

```
example : myTypeNumber = myNumber := rfl
```

`abbrev` defines an abbreviation, which is like a `def`, but always expands when processed. This is useful for type synonyms.

```
abbrev myAbbrev := ℚ
def myAbbrevNumber : myAbbrev := 998244353
#check myAbbrevNumber
```

Propositional equality

Propositional equality is

- defined as the inductive type `Eq` (notation `=`),
- constructed by the constructor `rfl` (reflexivity, i.e. `a = a`), with `propext` and `Quot.sound` as extra axioms (`funext` is an corollary of `Quot.sound`),
- eliminated by the `rw` tactic (in practice).

Propositional equality is not a meta-level concept. It's a proposition in `Prop` that may be proved or disproved.

Propositional equality on types does not get the types check. For example, this won't compile.

```
-- example (α : Type) (h : α = ℕ) (a : α) : a = (998244353 : ℕ) := by sorry

end
```

Chapter 6

Inequality

- Inequality `PartialOrder`
- `abs`, `min` and `max`
- The Art of Capturing Premises (TAOCP)
- Wheelchair tactics

```
import Mathlib
```

6.1 Inequality

6.1.1 Basics

Inequality is determined by a partial order `PartialOrder`. A partial order is a relation with reflexivity, antisymmetry, and transitivity. In Lean, a relation means `α → α → Prop` for some type `α`, capturing the fact that each `a ≤ b` gives a proposition.

```
section

variable (a b c d : ℚ)
```

`PartialOrder` makes `LE`(`≤`) and `LT`(`<`) available in the context.

```
#check PartialOrder

#check a ≤ b
#check a < b
#check b ≥ a
#check b > a

#check le_refl
#check le_antisymm
#check le_trans

#check lt_irrefl
```

```
#check lt_asymm
#check lt_trans
```

`<` is determined by `≤`

```
#check lt_iff_le_not_ge
```

`≥`, `>` are just aliases of `≤`, `<`

```
example : (a < b) = (b > a) := by rfl
example : (a ≤ b) = (b ≥ a) := by rfl

example : a < b ↔ a ≤ b ∧ a ≠ b := by
  rw [lt_iff_le_not_ge]
  constructor
  · intro ⟨hab, hnba⟩
    constructor
    · exact hab
    · intro h
      rw [h] at hnba
      apply hnba
      exact le_refl b
  · intro ⟨hab, hnab⟩
    constructor
    · exact hab
    · intro hba
      apply hnab
      exact le_antisymm hab hba
#check lt_of_le_of_ne -- this have a related theorem
```

A linearly ordered commutative ring is a commutative ring with a total order s.t addition and multiplication are strictly monotone, e.g. `ℚ`.

In Lean this reads `[CommRing R] [LinearOrder R] [IsStrictOrderedRing R]`.

We will work with `ℚ` as an example afterwards.

[TODO] For some reason, `LinearOrder ℚ` is constructed using classical logic. Don't be surprised if `#print axioms ...` shows some classical axioms.

6.1.2 Pure partial order reasoning

`norm_num` tactic solves numerical equalities and inequalities automatically.

```
#help tactic norm_num
example : (22 / 7 : ℚ) < 4 := by norm_num

-- [EXR]
example (hab : a ≤ b) (hba : b ≤ a) : a = b := by
  apply le_antisymm
  · exact hab
```

```
  · exact hba
```

`grw` rewrites like `rw`, but works for inequalities.

```
#help tactic grw
example (hab : a ≤ b) (hbc : b < c) : a < c := by
  grw [hab]
  exact hbc
example (hab : a ≤ b) (hbc : b < c) : a < c := by
  grw [← hab] at hbc
  exact hbc
#check lt_of_le_of_lt -- this have a name
```

`calc` is a term / tactic for proving inequalities by chaining.

```
#help tactic calc
example (hab : a ≤ b) (hbc : b < c) : a < c := by
  calc
    a ≤ b := hab
    _ < c := hbc
```

6.1.3 Linear order reasoning

A linear order is a partial order with `le_total`: either `a ≤ b` or `b ≤ a`.

```
#check le_total
```

[EXR] Use this to prove the trichotomy of `<` and `=`.

```
example : a < b ∨ a = b ∨ a > b := by
  rcases le_total a b with (hle | h)
  · by_cases heq : a = b
    · right; left; exact heq
    · left
      apply lt_of_le_of_ne
      · exact hle
      · exact heq
  · -- do it similarly
    sorry
#check eq_or_lt_of_le -- this have a name
```

6.1.4 Monotonicity of `+`

It's important to recognize that the (strict) monotonicity of `+` is a nontrivial theorem. That is a part of the meaning of `IsStrictOrderedRing`.

```
#synth IsStrictOrderedRing ℚ

#check add_le_add_left
#check add_le_add_right

#check add_lt_add_left
#check add_lt_add_right
```

Luckily, `grw` recognizes these theorems and applies them automatically.
transposition of `≤`

```
example (h : a + b ≤ c) : a ≤ c - b := by
  grw [← h]
  simp
```

monotonicity of `+`

```
example : a + c ≤ b + c ↔ a ≤ b := by
  constructor
  · intro h
    calc
      a = (a + c) - c := by simp
      _ ≤ (b + c) - c := by grw [h]
      _ = b := by simp
  · intro h
    grw [h]
#check add_le_add_iff_right -- this have a name
```

strict monotonicity of `+`

```
example : a < b ↔ a + c < b + c := by
  constructor
  · contrapose!
    intro h
    rw [add_le_add_iff_right] at h
    exact h
  · contrapose!
    intro h
    grw [h]
#check add_lt_add_iff_right -- this have a name

-- [EXR]
example (h : a + b < c + d) : a - d < c - b := by
  sorry
```

6.1.5 Automation

Tired of these? Use automation!

linarith

`linarith` is a powerful tactic that solves linear inequalities automatically. It uses hypotheses in the context and basic properties of linear orders to deduce the goal.

`linarith only [h1, h2, ...]` use only hypotheses `h1`, `h2`, … to solve the goal.

```
#help tactic linarith

example : a < b ↔ a - c < b - c := by
  constructor
  all_goals
    intro
    linarith

example (h : a + b < c + d) : a - d < c - b := by
  linarith

example (h : a > 0) : (2 / 3) * a > 0 := by
  linarith

example (h : (-5 / 3) * a > 0) : 4 * a < 0 := by
  linarith
```

Note the limitations of `linarith`.

It only works for linear inequalities, not polynomial ones.

```
example : a ^ 2 ≥ 0 := by sorry -- linarith fails here
```

though some of polynomial inequalities can be solved by `nlinarith`

```
#help tactic nlinarith
example : a ^ 2 ≥ 0 := by nlinarith
#check sq_nonneg -- this have a name
```

It solve all inequalities in a dense linear order.

It does solve some inequalities in discrete linear orders like ℤ, but no guarantee for all of them.

```
example (n m : ℤ) (h : n < m) : n + 1 ≤ m := by linarith
example (n m : ℤ) (h : n < m) : n + (1/2 : ℚ) ≤ m := by sorry -- linarith fails here
```

It won't recognize inequalities involving `min`, `max`, `abs`, etc. It won't recognize some basic `simp` transformations, either.

```
example (h : a * (min 1 2) > 0) : (id a) ≥ 0 := by
  simp at *
  linarith -- direct `linarith` will fail
```

[EXR] admits a dense linear order

```
example (hab : a < b) : ∃ c : ℚ, a < c ∧ c < b := by
  use (a + b) / 2
  constructor
  all_goals linarith
```

simp

`add_lt_add_iff_right`-like theorems are registered for `simp`, so sometimes `simp` can reduce things like:

```
example (h : a + b < c + b) : a < c := by
  simp at h
  exact h
```

apply_fun

Sometimes you would like to `apply_fun` at an inequality. This requires you to manually show the monotonicity of the function.

```
example (h : a + b < c + d) : a - d < c - b := by
  apply_fun (· - b - d) at h
  · ring_nf at *
    exact h
  · unfold StrictMono
    simp
```

[EXR] Mimick the above example.

```
example (h : a + c ≤ b) : a ≤ b - c := by
  apply_fun (· - c) at h
  · ring_nf at *
    exact h
  · unfold Monotone
    simp
```

6.1.6 Monotonicity of *

[TODO] It's not needed in the course so far, so we skip it for now.

```
end
```

6.2 abs, min, max and TAOCP

A mature formalizer finds their theorems by themselves. The art of capturing premises includes, but not limited to:

- `exact?`
- name guessing
- natural language search engine: LeanSearch, LeanExplore, etc.
- mathlib documentation
- AI copilot completion

```
section

variable (a b c : ℚ)

#check abs
```

[EXR] Find all the below by yourself

```


example : |a| ≥ 0 := by exact abs_nonneg a
example : |-a| = |a| := by exact abs_neg a
example : |a * b| = |a| * |b| := by exact abs_mul a b
example : |a + b| ≤ |a| + |b| := by exact abs_add_le a b
example : |a| - |b| ≤ |a - b| := by exact abs_sub_abs_le_abs_sub a b
example : |a| ≤ b ↔ -b ≤ a ∧ a ≤ b := by exact abs_le
example : |a| ≥ b ↔ a ≤ -b ∨ b ≤ a := by exact le_abs'

example (h : a ≥ 0) : |a| = a := by exact abs_of_nonneg h
example (h : a ≤ 0) : |a| = -a := by exact abs_of_nonpos h
example (h : b ≥ 0) : |a| = b ↔ a = b ∨ a = -b := by exact abs_eq h
```

A mindless way to prove these linear inequalities involving `abs` is to eliminate all `abs` by casing on the sign of the arguments, then use `linarith`.

```
example : |a - c| ≤ |a - b| + |b - c| := by
  all_goals rcases le_total 0 (a - b) with h1 | h1
  all_goals
    try rw [abs_of_nonneg h1]
    try rw [abs_of_nonpos h1]
  all_goals rcases le_total 0 (b - c) with h2 | h2
  all_goals
    try rw [abs_of_nonneg h2]
    try rw [abs_of_nonpos h2]
  all_goals rcases le_total 0 (a - c) with h3 | h3
  all_goals
    try rw [abs_of_nonneg h3]
    try rw [abs_of_nonpos h3]
```

```
  all_goals linarith
```

combine brute-force method with theorem-finding

```
example : |(|a| - |b|)| ≤ |a - b| := by
  rcases le_total 0 (|a| - |b|) with h1 | h1
  all_goals
    try rw [abs_of_nonneg h1]
    try rw [abs_of_nonpos h1]
  · apply abs_sub_abs_le_abs_sub
  · simp only [neg_sub] -- use `simp only` to supress unwanted lemmas
    grw [abs_sub_abs_le_abs_sub]
    rw [← abs_neg]
    simp

-- [EXR]
example : |(|a| - |b|)| ≤ |a + b| := by
  rcases le_total 0 (|a| - |b|) with h1 | h1
  all_goals
    try rw [abs_of_nonneg h1]
    try rw [abs_of_nonpos h1]
  · haveI := abs_sub_abs_le_abs_sub a (-b)
    simp at *
    exact this
  · haveI := abs_sub_abs_le_abs_sub b (-a)
    simp at *
    grw [this]
    ring_nf
    simp
```

[EXR] Get familiar with `min`, `max` and solve the following by yourself.

```
example : min a b ≤ a := by exact min_le_left a b
example : min a b + max a b = a + b := by exact min_add_max a b

end
```

6.3 Wheelchair tactics

You have seen some all-in-one tactics like `simp`, `ring` and `linarith`. There are even more powerful tactics that save your effort. Do try them when you feel tired of trivial steps.

```
#help tactic simp
#help tactic dsimp
#help tactic simp_rw
#help tactic field_simp
```

```
#help tactic group
#help tactic abel
#help tactic ring
#help tactic module

#help tactic linarith
#help tactic nlinarith

#help tactic omega
#help tactic aesop
#help tactic grind
#help tactic tauto
```

A tactic cheatsheet is available at lean-tactics.pdf

Chapter 7

Mathematical Analysis

Finally some real math in Lean! In this file we show how to define limits of sequences and continuity of functions in Lean. Of course it is just a toy version, far from the real Mathlib definitions. Nevertheless, that should be enough for you to get a taste of formalizing something that is not completely trivial.

Since we haven't touch division quite much yet, you may find it's difficult to deal with multiplication and division. `field_simp` tactic may help you a lot in such cases. It won't break things up as `simp` does. Anyway, don't worry too much about it for now.

```
import Mathlib

def TendsTo (a : ℕ → ℝ) (t : ℝ) : Prop :=
  ∀ ε > 0, ∃ n₀ : ℕ, ∀ n, n₀ ≤ n → |a n - t| < ε
```

[EXR] The limit of the constant sequence with value `c` is `c`.

```
theorem tendsTo_const (c : ℝ) : TendsTo (fun _ ↦ c) c := by
  unfold TendsTo
  intro ε hε
  use 1
  intro n hn
  simp [hε]
```

`-` commutes with `tendsTo`

```
theorem tendsTo_neg {a : ℕ → ℝ} {t : ℝ} (ha : TendsTo a t) : TendsTo (fun n ↦ -a n) (-t) := by
  unfold TendsTo
  intro ε hε
  specialize ha ε hε
  rcases ha with ⟨n₀, hn₀⟩
  use n₀
  intro n hn
  specialize hn₀ n hn
  simp
  -- what theorems should I use?
```

```
  rw [← abs_neg, add_comm]
  simp
  -- what theorems should I use?
  rw [← sub_eq_add_neg]
  exact hn₀
```

`+` commutes with `tendsTo`

```
theorem tendsTo_add {a b : ℕ → ℝ} {A : ℝ} {B : ℝ} (ha : TendsTo a A) (hb : TendsTo b B) :
    TendsTo (fun n => a n + b n) (A + B) := by
  intro ε hε
  specialize ha (ε / 2) (by linarith only [hε])
  specialize hb (ε / 2) (by linarith only [hε])
  rcases ha with ⟨n₀, ha⟩
  rcases hb with ⟨m₀, hb⟩
  use max n₀ m₀
  intro n hn
  -- what theorems should I use?
  rw [max_le_iff] at hn
  specialize ha n (by linarith only [hn.left])
  specialize hb n (by linarith only [hn.right])
  simp
  -- common tactic: eliminate abs to make use of `linarith`
  -- what theorems should I use?
  rw [abs_lt] at ha hb ⊢
  constructor
  · linarith only [ha, hb]
  · linarith only [ha, hb]
```

[EXR] `-` commutes with `tendsTo`

```
theorem tendsTo_sub {a b : ℕ → ℝ} {A B : ℝ} (ha : TendsTo a A) (hb : TendsTo b B) :
    TendsTo (fun n => a n - b n) (A - B) := by
  haveI := tendsTo_add ha (tendsTo_neg hb)
  -- [TODO] `congr` closes the goal directly here. Find out why.
  ring_nf at this
  exact this
```

`≤` version of `TendsTo` is equivalent to the usual `TendsTo`.

```
def TendsTo_le (a : ℕ → ℝ) (t : ℝ) : Prop :=
  ∀ ε > 0, ∃ n₀ : ℕ, ∀ n, n₀ ≤ n → |a n - t| ≤ ε

-- [EXR]
theorem tendsTo_le_iff_TendsTo {a : ℕ → ℝ} {t : ℝ} : TendsTo_le a t ↔ TendsTo a t := by
  constructor
  · intro h ε hε
```

```
    rcases h (ε / 2) (by linarith [hε]) with ⟨n₀, hn₀⟩
    use n₀
    intro n hn; specialize hn₀ n hn
    linarith only [hn₀, hε]
  · intro h ε hε
    rcases h ε hε with ⟨n₀, hn₀⟩
    use n₀
    intro n hn; specialize hn₀ n hn
    linarith only [hn₀, hε]
```

a weaker version of `TendsTo` where we require `ε < l`. When `l > 0`, this is equivalent to `TendsTo`.

```
def TendsTo_εlt (a : ℕ → ℝ) (t : ℝ) (l : ℝ) : Prop :=
  ∀ ε > 0, ε < l → ∃ n₀ : ℕ, ∀ n, n₀ ≤ n → |a n - t| < ε

theorem tendsTo_εlt_iff_TendsTo {a : ℕ → ℝ} {t : ℝ} {l : ℝ} (l_gt_zero : l > 0) :
    TendsTo_εlt a t l ↔ TendsTo a t := by
  constructor
  · intro h ε hε
    specialize h (min ε (l / 2))
                 (by apply lt_min; all_goals linarith)
                 (by apply min_lt_of_right_lt; linarith only [l_gt_zero])
    rcases h with ⟨n₀, hn₀⟩; use n₀
    intro n hn; specialize hn₀ n hn
    rw [lt_min_iff] at hn₀
    exact hn₀.left
  · exact fun h ε hε _ ↦ h ε hε
```

`*` commutes with `tendsTo`. [TODO] I failed to finish the proof swiftly. You are welcome to optimize it!

```
theorem tendsTo_mul {a b : ℕ → ℝ} {A B : ℝ} (ha : TendsTo a A) (hb : TendsTo b B) :
    TendsTo (fun n ↦ a n * b n) (A * B) := by
  rw [← tendsTo_εlt_iff_TendsTo (show 1 > 0 by linarith)]
  intro ε hε hεlt1; simp
  specialize ha (ε / (3 * (|B| + 1))) (by
    apply div_pos hε
    linarith only [abs_nonneg B])
  rcases ha with ⟨n₁, ha⟩
  specialize hb (ε / (3 * (|A| + 1))) (by
    apply div_pos hε
    linarith only [abs_nonneg A])
  rcases hb with ⟨n₂, hb⟩
  use max n₁ n₂
  intro n hn
  rw [max_le_iff] at hn
  specialize ha n hn.left
  specialize hb n hn.right
```

```
  rw [show a n * b n - A * B = (a n - A) * (b n - B) + A * (b n - B) + B * (a n - A) by ring]
  repeat grw [abs_add]
  repeat grw [abs_mul]
  grw [ha, hb]
  -- sometimes you have no choice but add some manual steps
  have h1 : |A| * (ε / (3 * (|A| + 1))) < ε / 3 := by
    field_simp
    rw [div_lt_iff₀]
    · ring_nf
      linarith only [hε]
    · linarith only [abs_nonneg A]
  have h2 : |B| * (ε / (3 * (|B| + 1))) < ε / 3 := by
    field_simp
    rw [div_lt_iff₀]
    · ring_nf
      linarith only [hε]
    · linarith only [abs_nonneg B]
  have h3 : ε / (3 * (|B| + 1)) * (ε / (3 * (|A| + 1))) < ε / 3 := by
    field_simp
    rw [div_lt_iff₀]
    · repeat grw [← abs_nonneg]
      ring_nf
      calc
        _ = ε * ε := by ring
        _ ≤ 1 * ε := by grw [← hεlt1]
        _ = ε     := by ring
        _ < ε * 3 := by linarith only [hε]
    · repeat grw [← abs_nonneg]
      ring_nf
      linarith only
  linarith only [h1, h2, h3]
```

squeeze theorem for sequences

```
theorem tendsTo_sandwich {a b c : ℕ → ℝ} {L : ℝ} (ha : TendsTo a L) (hc : TendsTo c L)
    (hab : ∀ n, a n ≤ b n) (hbc : ∀ n, b n ≤ c n) : TendsTo b L := by
  unfold TendsTo
  intro ε hε
  specialize ha ε hε
  specialize hc ε hε
  rcases ha with ⟨n₀, hn₀⟩
  rcases hc with ⟨m₀, hm₀⟩
  use max n₀ m₀
  intro n hn
  rw [max_le_iff] at hn
  specialize hab n
  specialize hn₀ n (by linarith only [hn.left])
  specialize hm₀ n (by linarith only [hn.right])
  specialize hbc n
  rw [abs_lt] at hn₀ hm₀ ⊢
```

```
  constructor
  · linarith only [hn₀, hm₀, hbc, hab]
  · linarith only [hn₀, hm₀, hbc, hab]
```

constant sequence tends to zero iff condition

```
theorem tendsTo_zero_iff_lt_ε {x : ℝ} : TendsTo (fun _ ↦ x) 0 ↔ (∀ ε > 0, |x| < ε) := by
  constructor
  · intro h ε hε
    specialize h ε hε
    rcases h with ⟨n₀, hn₀⟩
    specialize hn₀ n₀ (by linarith only)
    simp at hn₀; exact hn₀
  · intro h
    intro ε hε
    specialize h ε hε
    use 0
    intro n hn
    simp; exact h
```

[EXR] zero sequence tends to x iff condition

```
theorem zero_tendsTo_iff_lt_ε {x : ℝ} : TendsTo (fun _ ↦ 0) x ↔ (∀ ε > 0, |x| < ε) := by
  constructor
  · intro h
    unfold TendsTo at h; simp at h
    intro ε hε
    specialize h ε hε
    rcases h with ⟨n₀, hn₀⟩
    specialize hn₀ n₀ (by linarith only)
    exact hn₀
  · intro h
    intro ε hε
    use 0
    intro n hn
    simp
    exact h ε hε
```

uniqueness of limits

```
theorem tendsTo_unique (a : ℕ → ℝ) (s t : ℝ) (hs : TendsTo a s) (ht : TendsTo a t) : s = t := by
  by_contra! hneq
  have hstp : 0 < |t - s| := by
    rw [abs_pos]
    contrapose! hneq
    apply_fun fun x ↦ x + s at hneq
    simp at hneq
```

```
    symm
    exact hneq
  have hst := tendsTo_sub hs ht
  simp at hst
  rw [zero_tendsTo_iff_lt_ε] at hst
  specialize hst |t - s| hstp
  rw [abs_sub_comm] at hst
  linarith only [hst]

def contAt (f : ℝ → ℝ) (x₀ : ℝ) : Prop :=
  ∀ ε > 0, ∃ δ > 0, ∀ x, |x - x₀| < δ → |f x - f x₀| < ε

def cont (f : ℝ → ℝ) : Prop := ∀ x₀ : ℝ, contAt f x₀
```

continuity of function composition

```
def contAt_comp {f g : ℝ → ℝ} {x₀ : ℝ} (hf : contAt f (g x₀)) (hg : contAt g x₀) :
    contAt (f ∘ g) x₀ := by
  intro ε hε
  rcases hf ε hε with ⟨δf, hδf, hf⟩
  rcases hg δf hδf with ⟨δg, hδg, hg⟩
  use δg, hδg
  intro x hx
  simp only [Function.comp_apply]
  specialize hg x hx
  specialize hf (g x) hg
  exact hf
```

[EXR] continuity of function composition

```
def cont_comp {f g : ℝ → ℝ} (hf : cont f) (hg : cont g) : cont (f ∘ g) := by
  intro x
  exact contAt_comp (hf (g x)) (hg x)
```

[EXR] continuity implies sequential continuity

```
def tendsTo_of_contAt {f : ℝ → ℝ} {x₀ : ℝ} (hf : contAt f x₀)
    {a : ℕ → ℝ} (ha : TendsTo a x₀) : TendsTo (f ∘ a) (f x₀) := by
  intro ε hε
  rcases hf ε hε with ⟨δ, hδ, hδf⟩
  specialize ha δ hδ
  rcases ha with ⟨n₀, hn₀⟩
  use n₀
  intro n hn
  specialize hn₀ n hn
  specialize hδf (a n) hn₀
  exact hδf
```

The uniform limit of a sequence of continuous functions is continuous.

```
def uconv (f : ℕ → ℝ → ℝ) (f₀ : ℝ → ℝ) : Prop :=
  ∀ ε > 0, ∃ N : ℕ, ∀ n ≥ N, ∀ x : ℝ, |f n x - f₀ x| < ε

theorem cont_of_cont_of_uconv
    (f : ℕ → ℝ → ℝ) (f_cont : ∀ n : ℕ, cont (f n))
    (f₀ : ℝ → ℝ) (h_uconv : uconv f f₀) : cont f₀ := by
  intro x₀ ε hε
  rcases h_uconv (ε / 3) (by linarith only [hε]) with ⟨N, hN⟩
  specialize hN N (by linarith)
  rcases f_cont N x₀ (ε / 3) (by linarith only [hε]) with ⟨δ, hδ, hδf⟩
  use δ, hδ
  intro x hx
  specialize hδf x hx
  have hNx := hN x
  have hNx₀ := hN x₀
  -- brute force `linarith` argument
  rw [abs_lt] at hNx hNx₀ hδf ⊢
  constructor
  all_goals linarith only [hNx, hNx₀, hδf]
```

The sequential definition of function continuity is equivalent to the epsilon-delta definition.

```
def contAt_seq (f : ℝ → ℝ) (x₀ : ℝ) : Prop :=
  ∀ a : ℕ → ℝ, TendsTo a x₀ → TendsTo (f ∘ a) (f x₀)
```

[TODO] I failed to solve it swiftly. You are welcome to optimize it!

```
theorem contAt_iff_seq (f : ℝ → ℝ) (x₀ : ℝ) :
    contAt f x₀ ↔ contAt_seq f x₀ := by
  constructor
  · intro hf a ha
    exact tendsTo_of_contAt hf ha
  · contrapose
    intro hnfcont hnfseq
    unfold contAt at hnfcont
    push_neg at hnfcont
    -- construct a sequence `a n` tending to `x₀`
    let a (n : ℕ) : ℝ := 1 / (n + 1)
    have a_gt_zero (n : ℕ) : a n > 0 := by simp [a]; linarith only
    have a_TendsTo_zero : TendsTo a 0 := by
      intro ε hε
      use Nat.ceil (1 / ε) -- ceiling function
      intro n hn
      rw [Nat.ceil_le] at hn
      simp
      rw [abs_of_pos (a_gt_zero n)]
      unfold a
```

```
    rw [div_lt_comm₀ (by linarith only) hε]
    linarith only [hn]
  -- construct a diverging sequence `f x` with `x` tending to `x₀`
  -- this requires us to extract `Type*` objects from an existence to form a function
  -- may meet universe issues if done naively
  -- we use `Classical.indefiniteDescription` here to extract such objects classically
  rcases hnfcont with ⟨ε, hε, hnf⟩
  let x_subtype (n : ℕ) := Classical.indefiniteDescription _ <| hnf (a n) (a_gt_zero n)
  let x (n : ℕ) : ℝ := (x_subtype n).val
  have x_lt_a (n : ℕ) : |x n - x₀| < a n := by
    unfold x
    exact (x_subtype n).property.left
  have fx_diverge (n : ℕ) : |f (x n) - f x₀| ≥ ε := by
    unfold x
    exact (x_subtype n).property.right

  have x_tendsTo_x₀ : TendsTo x x₀ := by
    suffices TendsTo (fun n ↦ x n - x₀) 0 by
      have h_add := tendsTo_add this (tendsTo_const x₀)
      simp at h_add; exact h_add
    refine tendsTo_sandwich (?_ : TendsTo (fun n ↦ -a n) 0) (?_ : TendsTo (fun n ↦ a n) 0) ?_ ?_
     · haveI := tendsTo_neg a_TendsTo_zero
      simp at this; exact this
     · exact a_TendsTo_zero
    all_goals
      intro n
      haveI := x_lt_a n
      rw [abs_lt] at this
      linarith only [this]
  -- but it is said that all such sequences converge
  haveI := hnfseq x x_tendsTo_x₀
  rcases this ε hε with ⟨n₀, hn₀⟩
  specialize hn₀ n₀ (by linarith only); simp at hn₀
  specialize fx_diverge n₀
  linarith only [hn₀, fx_diverge]
```

Part IV

Structure

Chapter 8

Groups and Homomorphisms

The ultimate goal of the following several lectures is to state and prove the First Isomorphism Theorem for groups.

You might notice that starting from now, every lecture gets intolerablely lengthy. Unfortunately, this is the nature of formal mathematics. One has to endure this pain to reach the harmony of full formalization.

We organize the materials with respect to the philosophy of "illustrate the theory" (see the Preface if you don't know what this means), Be advised that you can always ctrl+click on any name to see its actual definition in Mathlib.

In this lecture, we illustrate how Mathlib develops the theory of everyday algebraic structures, starting from semigroups, monoids, groups, and their morphisms.

For a more complete treatment (especially on the philosophy behind API design), read MiL chapter 7 and 9.

```
import Mathlib
```

8.1 Semigroups

8.1.1 Objects

A `Semigroup` is a type with an associative binary operation `*`.

```
section

#check Semigroup
variable (G : Type*) [Semigroup G] (a b c : G)
example : a * (b * c) = (a * b) * c := by rw [mul_assoc]
```

An `AddSemigroup` is exactly the same as `Semigroup`, only with additive `+` notation.

```
variable (A : Type*) [AddSemigroup A] (a b c : A)
example : a + (b + c) = (a + b) + c := by rw [add_assoc]
```

Note that using the notation of `+` does not necessarily mean that the operation is commutative. To this end, we have `CommSemigroup` and `AddCommSemigroup`.

```
#check CommSemigroup
#check mul_comm

#check AddCommSemigroup
#check add_comm

end
```

8.1.2 Morphisms

A `MulHom` is a morphism between two semigroups that preserves the multiplication. The notation for this is `G₁ →ₙ* G₂`.

It's a bundle of:

- a function `f : G₁ → G₂`
- a proof that `f` preserves multiplication.

Strictly speaking, this definition does not require the `*` operation to be associative on `G₁` or `G₂`.

```
section

#check MulHom
variable {G₁ G₂ G₃ : Type*} [Semigroup G₁] [Semigroup G₂] [Semigroup G₃]
         (f : G₁ →ₙ* G₂) (g : G₂ →ₙ* G₃) (a b : G₁)
example : f (a * b) = f a * f b := by rw [map_mul]
```

Additive version of `MulHom` is `AddHom` (`→ₙ+`).

```
#check AddHom
```

You might already notice that a `MulHom` can be used just like a function. This is because Mathlib has instantiated the `FunLike` type class to `MulHom`, which provides the function coercion.

```
#synth FunLike (MulHom G₁ G₂) G₁ G₂
```

Creating a new `MulHom` requires providing all the data needed.

```
#check MulHom.mk
example : ℕ →ₙ+ ℕ := ⟨(· * 2), by intros; ring⟩
```

Composition of `MulHom`s as functions preserves multiplication.

```
example : G₁ →ₙ* G₃ := ⟨g ∘ f, by intros; dsimp; rw [map_mul, map_mul]⟩
```

To avoid manually constructing `MulHom` every time when composing, We may use `MulHom.comp g f`. The dot convention `g.comp f` is used here for convenience.

```
example : (g.comp f) (a * b) = (g.comp f) a * (g.comp f) b := by simp
```

A `MulHom` is determined by the underlying function.

```
#check MulHom.ext
example (f₁ : G₁ →ₙ* G₂) (f₂ : G₁ →ₙ* G₂) (h : f₁.toFun = f₂.toFun) : f₁ = f₂ := by
  ext x
  change f₁.toFun x = f₂.toFun x
  rw [h]
```

Above shows the bundled definition of `MulHom`, how to create it, and how to compose them. The same philosophy is adopted for other morphism-like structures in Mathlib, such as `MonoidHom`.

```
end
```

8.2 Monoids

8.2.1 Objects

A `Monoid` is a `Semigroup` with an identity element `1` s.t. `a * 1 = a` and `1 * a = a`.

```
section

#check Monoid
variable (G : Type*) [Monoid G] (a b c : G)
example : a * 1 = a := by rw [mul_one]
example : 1 * a = a := by rw [one_mul]
```

[EXR] characterization of the identity element

```
example (h : ∀ x : G, x * a = x) : a = 1 := by
  specialize h 1
  rw [one_mul] at h
  exact h
```

`Monoid` additionaly enables power notation `a ^ n` for natural number `n`.

```
#check Monoid.npow
example : a ^ 0 = 1 := by rw [pow_zero]
example (n : ℕ) : a ^ (n + 1) = a ^ n * a := by rw [pow_succ]
```

We are not prepared to prove this until we talk about induction.

```
#check one_pow
```

`AddMonoid` is the additive version of `Monoid`.

```
variable (A : Type*) [AddMonoid A] (a b c : A)
example : a + 0 = a := by rw [add_zero]
example : 0 + a = a := by rw [zero_add]
example : 0 • a = 0 := by rw [zero_smul]
example (n : ℕ) : (n + 1) • a = n • a + a := by rw [succ_nsmul]
```

For commutative monoids, we have `CommMonoid` and `AddCommMonoid`.

```
#check CommMonoid
#check AddCommMonoid

end
```

8.2.2 Morphisms

A `MonoidHom` is a morphism between two monoids that preserves the multiplication and the identity. The notation for this is `G →* H`.

```
section

#check MonoidHom
variable {G₁ G₂ G₃ : Type*} [Monoid G₁] [Monoid G₂] [Monoid G₃]
         (f : G₁ →* G₂) (g : G₂ →* G₃) (a b : G₁)
example : f (a * b) = f a * f b := by rw [map_mul]
example : f 1 = 1 := by rw [map_one]
```

Additive version of `MonoidHom` is `AddMonoidHom` (`→+`).

```
#check AddMonoidHom
```

`MonoidHom` need additional data to `MulHom`: preservation of `1`.

```
#check MonoidHom.mk
example : ℕ →+ ℕ := ⟨⟨( · * 2), by simp⟩, by intros; ring⟩

end
```

8.3 Groups

8.3.1 Objects

In Mathlib, a `Group` is defined to be a `Monoid` where every element `a` has an left inverse `a⁻¹` s.t. `a⁻¹ * a = 1`.

```
section

#check Group
variable (G : Type*) [Group G] (a b c : G)
#check a⁻¹
example : a⁻¹ * a = 1 := by rw [inv_mul_cancel]
```

The following exercises lead to a proof of: In a group, a left inverse is also a right inverse. This recovers the usual definition of a group.

[EXR] left multiplication is injective

```
example (h : a * b = a * c) : b = c := by
  apply_fun (a⁻¹ * · ) at h
  rw [← mul_assoc, ← mul_assoc, inv_mul_cancel, one_mul, one_mul] at h
  exact h
#check mul_left_cancel -- corresponding Mathlib theorem
```

[EXR] a left inverse actually also a right inverse

```
example : a * a⁻¹ = 1 := by
  apply_fun (a⁻¹ * · )
  · dsimp
    rw [← mul_assoc, inv_mul_cancel, one_mul, mul_one]
  · apply mul_left_cancel
#check mul_inv_cancel -- corresponding Mathlib theorem
```

The following proves that `G` is a `DivisionMonoid`. You don't need to know what this means for now.

```
#synth DivisionMonoid G
```

[EXR] characterization of a right inverse

```
example (h : a * b = 1) : b = a⁻¹ := by
  -- if you does not want to use `apply_fun`
  rw [← one_mul b, ← inv_mul_cancel a, mul_assoc, h, mul_one]
#check eq_inv_of_mul_eq_one_right -- corresponding Mathlib theorem
```

[EXR] characterization of a left inverse

```
example (h : a * b = 1) : a = b⁻¹ := by
  apply_fun (· * b⁻¹) at h
  rw [mul_assoc, mul_inv_cancel, mul_one, one_mul] at h
  exact h
#check eq_inv_of_mul_eq_one_left -- corresponding Mathlib theorem
```

[EXR] involutivity of the inverse

```
example : (a⁻¹)⁻¹ = a := by
  symm; apply eq_inv_of_mul_eq_one_right
  exact inv_mul_cancel a
#check inv_inv -- corresponding Mathlib theorem
```

[EXR] inverse of a product

```
example : (a * b)⁻¹ = b⁻¹ * a⁻¹ := by
  apply inv_eq_of_mul_eq_one_left
  rw [← mul_assoc, mul_assoc b⁻¹, inv_mul_cancel, mul_one, inv_mul_cancel]
#check mul_inv_rev -- corresponding Mathlib theorem
```

some other injectivity
[EXR] inverse is injective

```
example (h : a⁻¹ = b⁻¹) : a = b := by
  apply_fun (· ⁻¹) at h
  rw [inv_inv a, inv_inv b] at h
  exact h
#check inv_injective -- corresponding Mathlib theorem
```

[EXR] right multiplication is injective

```
example (h : b * a = c * a) : b = c := by
  apply_fun (· * a⁻¹) at h
  rw [mul_assoc, mul_assoc, mul_inv_cancel, mul_one, mul_one] at h
  exact h
#check mul_right_cancel -- corresponding Mathlib theorem
```

wheelchair tactic for groups

```
#help tactic group
example : (a ^ 3 * b⁻¹)⁻¹ = b * a⁻¹ * (a ^ 2)⁻¹  := by
  group
```

[TODO] `DivInvMonoid` enables `zpow` notation for integer powers `a ^ n` where `n : ℤ`.

[IGNORE] It extends `npow` for monoids. See the library note [forgetful inheritance] for the philosophy of this definition.

Additive and commutative versions of groups are as usual.

```
#check AddGroup
#check CommGroup
#check AddCommGroup

end
```

8.3.2 Morphisms

`MonoidHom` It is also used for group homomorphisms.

```
section

variable {G₁ G₂ G₃ : Type*} [Group G₁] [Group G₂] [Group G₃]
         (f : G₁ →* G₂) (g : G₂ →* G₃) (a b : G₁)
```

[EXR] Monoid homomorphisms preserve inverses

```
example : f (a⁻¹) = (f a)⁻¹ := by
  apply eq_inv_of_mul_eq_one_right
  rw [← map_mul, mul_inv_cancel, map_one]
#check map_inv -- corresponding Mathlib theorem
```

[EXR] `MonoidHom` requires one to show preservation of `1`. But this is redundant for group homomorphisms.

```
example (φ : G₁ →ₙ* G₂) : φ 1 = 1 := by
  haveI : φ 1 * φ 1 = φ 1 * 1 := by rw [← map_mul, mul_one, mul_one]
  exact mul_left_cancel this
```

Hence in the case of groups, Mathlib provides a constructor `MonoidHom.mk'` that only requires the preservation of multiplication to build a `MonoidHom`.

```
#check MonoidHom.mk'

end
```

Chapter 9

Substructures and Subgroups

Diffrent people formalize things differently. This is especially true when it comes to substructures and quotients.

In this file, we show how to use the Mathlib's API for substructures, from subsets to subgroups. It's a sophisticated hierarchy that I'm still trying to fully understand myself. For the philosophy behind this design, see MiL chapter 8.

```
import Mathlib
```

9.1 Subsets

9.1.1 Objects

In the previous lectures, we regarded types as sets intuitively. This is not flexible when one wishes to restrict to only a fraction of the elements of a type. It's also hard to implement unions and intersections of sets this way. Mathlib provides a dedicated type `Set α`, consists of all the subsets of a type `α`.

```
section

variable (α : Type*) (s t u : Set α) (a : α)

#print Set
```

You can see that `Set α` is defined as `α → Prop`.

This means a subset `s : Set α` tells you, for each `a : α`, whether `a` belongs to `s` or not. This proposition is denoted by `a ∈ s`, `Set.mem s a`, or `s.Mem a`.

Note that you are not supposed to write `s a` directly. The function definition of `Set α` should be regarded as an implementation detail.

```
#check a ∈ s
#check s.Mem a
```

A subset can be constructed using the set-builder notation `{x : α | p x}` or `setOf p`, where `p : α → Prop` is a predicate on `α`.

```
#check setOf (fun x ↦ x = a)
example : setOf (fun x ↦ x = a) = {x : α | x = a} := by rfl
example : setOf (fun x ↦ x = a) = {a} := by rfl
example : setOf (fun x ↦ x = a) = Set.singleton a := by rfl

example : Set ℕ := {n | n > 514}
example (n : ℕ) : n ∈ {x | x > 514} ↔ n > 514 := by rfl
```

the empty subset

```
#check (∅ : Set α)
example : ∅ = {x : α | False} := by rfl
example : a ∈ (∅ : Set α) ↔ False := by rfl
#check Set.mem_empty_iff_false -- corresponding `simp` lemma
```

the universal subset

```
#check (Set.univ : Set α)
example : Set.univ = {x : α | True} := by rfl
example : a ∈ Set.univ := by trivial
#check Set.mem_univ -- corresponding `simp` lemma
```

the complement of a subset

```
#check Set.compl s
#check sᶜ
example : sᶜ = {x | x ∉ s} := by rfl
example : a ∈ sᶜ ↔ a ∉ s := by rfl
#check Set.mem_compl -- corresponding `simp` lemma
```

Subset relation. Somehow you may use any proof of `s ⊆ t` like a function. It eats a proof of `a ∈ s` and produces a proof of `a ∈ t`.

```
#check Set.Subset s t
#check s ⊆ t
example : s ⊆ t ↔ ∀ x : α, x ∈ s → x ∈ t := by rfl
example (ha : a ∈ s) (hst : s ⊆ t) : a ∈ t := hst ha
#check Set.mem_of_subset_of_mem -- corresponding Mathlib lemma
```

intersection of two subsets

```
#check Set.inter s t
#check s ∩ t
example : a ∈ s ∩ t ↔ a ∈ s ∧ a ∈ t := by rfl
#check Set.mem_inter_iff -- corresponding `simp` lemma
```

union of two subsets

```
#check Set.union s t
#check s ∪ t
example : a ∈ s ∪ t ↔ a ∈ s ∨ a ∈ t := by rfl
#check Set.mem_union -- corresponding `simp` lemma
```

`ext` tactic reduces subset equality to element membership. Fundamentally this is implimented using function & propositional extensionality.

```
#check Set.ext
example : s ∩ t = t ∩ s := by ext x; simp [and_comm]
```

[EXR] De Morgan's law for sets

```
example : s ∩ (t ∪ u) = (s ∩ t) ∪ (s ∩ u) := by
  ext x
  constructor
  · intro ⟨h₁, h₂⟩
    rcases h₂ with (h₂ | h₂)
    · left
      exact ⟨h₁, h₂⟩
    · right
      exact ⟨h₁, h₂⟩
  · tauto_set

#help tactic tauto_set -- Wheelchair tactic for set equations

end
```

9.1.2 Morphisms

Functions between types induce functions between subsets.

```
section

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

range of a function `Set.range f`

```
#check Set.range f
example : Set.range f = {y | ∃ x, f x = y} := by rfl
example : Set.range f = {f x | x : α} := by rfl -- set-builder notation for range
example : b ∈ Set.range f ↔ ∃ x, f x = b := by rfl
#check Set.mem_range -- corresponding `simp` lemma
```

image of a subset `Set.image f s`

```
#check f '' s
#check Set.image f s
example : f '' s = {y | ∃ x ∈ s, f x = y} := by rfl
example : f '' s = {f x | x ∈ s} := by rfl -- set-builder notation for image
example : b ∈ f '' s ↔ ∃ x ∈ s, f x = b := by rfl
#check Set.mem_image -- corresponding `simp` lemma
```

Preimage of a subset `Set.preimage f t`

```
#check f ⁻¹' t
#check Set.preimage f t
example : f ⁻¹' t = {x | f x ∈ t} := by rfl
example : a ∈ f ⁻¹' t ↔ f a ∈ t := by rfl
#check Set.mem_preimage -- corresponding `simp` lemma
```

Note the following is not a definitional equality. The last step invokes `propext`, which destroys definitional equality. It has some unfortunate consequences in later discussions, when additional structure is involved.

```
example : f '' Set.univ = Set.range f := by
  ext x
  rw [Set.mem_range, Set.mem_image]
  simp only [Set.mem_univ, true_and]
#check Set.image_univ -- corresponding `simp` lemma
```

We teach an syntax sugar of `rcases` here: Say we are given `h : y ∈ Set.range f`, it is by definition `h : ∃ x, f x = y`. `rcases h with ⟨x, rfl⟩` will create a new variable `x : α`, and replace all occurrences of `y` in the context with `f x`.

Feel free to combine it with `rintro`.

```
example : Set.range f ⊆ t ↔ ∀ x, f x ∈ t := by
  constructor
  · intro h x
    apply h
    use x
  · intro h y hy
    rcases hy with ⟨x, rfl⟩
    exact h x
#check Set.range_subset_iff -- corresponding Mathlib theorem
```

[EXR] The so-called Galois connection between image and preimage.

```
example : f '' s ⊆ t ↔ s ⊆ f ⁻¹' t := by
  constructor
```

```
  · intro h x hx
    simp only [Set.mem_preimage]
    apply h
    simp only [Set.mem_image]
    use x
  · rintro h y ⟨x, hxs, rfl⟩
    specialize h hxs
    simp only [Set.mem_preimage] at h
    exact h
#check Set.image_subset_iff -- corresponding `simp` lemma

end
```

9.1.3 Exercises on functions and subsets

As an (a little bit advanced) exercise, prove that `f` has both inverses iff it is bijective. You might need the axiom of choice to construct such inverses. Familarize yourself with the following definitions if you haven't seen them before.

```
section

#check Function.comp
#check Function.Injective
#check Function.Surjective
#check Function.Bijective

#check Function.LeftInverse
#check Function.RightInverse

#check Classical.choose

#check Equiv
```

[EXR] your goal

```
#check Equiv.ofBijective
```

For those seeking a more challenging exercise, try proving the Bernstein–Schroeder theorem. See MiL chapter 3 for an answer.

```
#check Function.Embedding.schroeder_bernstein

end
```

9.1.4 Subset as a type

At most of the time, we prefer to write `a ∈ s`-like expressions, to indicate that `a : α` belongs to the subset `s : Set α`. We prefer this way because this does not create an extra psychological hierarchy in types: (recall that `Set α := α → Prop`, which is in the same universe as `α`)

```
Type u    α    Set α
          |      |
Term      a      s
```

However, there are situations where we want to treat `a` as an element of `s` directly: for example, when we wish to obtain a surjection from a function to its range.

Lean provides a way to do this: directly write `a : s` to say `a` is an element of the subset `s`.

```
section

variable {α : Type*} (s : Set α)

variable (a : s)
#check a
```

Note that `a` is now of type `↑s`, not `α`. This means that `a` is actually a bundled structure consisting of

- an element of type `α`, denoted by `a.val` or `↑a`
- a proof of `a.val ∈ s`, denoted by `a.property`

```
#check a.val
#check a.property
```

In tactic mode, `rcases` may be used to destructure `a : s` into its components.

```
example : a.val ∈ s := by
  rcases a with ⟨v, p⟩
  exact p
```

Given an `v : α` and a proof `p : v ∈ s`, we can construct an element of type `s` using `⟨v, p⟩`.

```
example (v : α) (p : v ∈ s) : ∃ a : s, a.val = v := by use ⟨v, p⟩
```

Mechanically, when Lean sees `a : s`, it automatically coerces `s` to a subtype of `α`, defined as `{x : α // x ∈ s}`. That is the coercion sign you see in the result `a : ↑s` of the type check. And hence the `a.val` and `a.property` is acutally `Subtype.val a` and `Subtype.property a`.

Though psychologically we have `a : s` and `s : Set α`, the actual hierarchy remains flat:

```
Type u      α        Set α        ↑s
            |          |          |
Term      a.val        s          a
```

```
#check Subtype
#check {x : α // x ∈ s}
example : {x : α // x ∈ s} = ↑s := by rfl
```

It's important to recognize that `Subtype.val : ↑s → α` is injective.

Note that `ext` is a general tactic to reduce an equality of structures into equalities of their components. You can use `rcases` to do this manually if you wish.

```
example (a₁ a₂ : s) (h : a₁.val = a₂.val) : a₁ = a₂ := by ext; exact h
#check Subtype.val_injective

end
```

9.1.5 Functions restricted to subsets

```
section

variable {α β : Type*} (f : α → β) (s : Set α) (s' : Set α) (t : Set β)
```

Given a function `f : α → β`, we can restrict its domain to a subset `s : Set α`.

```
#check Set.restrict
```

the universal property of `Set.restrict`

```
example : s.restrict f = f ∘ Subtype.val := by rfl
#check Set.restrict_apply -- corresponding `simp` lemma
```

the range of a restricted function

```
example : Set.range (s.restrict f) = f '' s := by
  ext y
  constructor
  · rintro ⟨x, rfl⟩
    dsimp
    use x.val, x.property
  · rintro ⟨x, hx, rfl⟩
    use ⟨x, hx⟩
    dsimp
#check Set.range_restrict -- corresponding `simp` lemma
```

Given a function `f : α → β`, we can also restrict its codomain to a subset `t : Set β`, once we know that `Set.range f ⊆ t`.

```
#check Set.range_subset_iff -- recall what we proved earlier
#check Set.codRestrict
example (h : ∀ x, f x ∈ t) : t.codRestrict f h = fun x ↦ ⟨f x, h x⟩ := by rfl
```

the universal property of `Set.codRestrict`

```
example (h : ∀ x, f x ∈ t) : Subtype.val ∘ (t.codRestrict f h) = f := by
  funext x
  rfl
#check Set.val_codRestrict_apply -- corresponding `simp` lemma
```

restriction on range

```
#check Set.rangeFactorization
example : Set.rangeFactorization f = (Set.range f).codRestrict f (by simp) := by rfl
#check Set.rangeFactorization_coe -- universal property of range restriction

end
```

9.2 Subsemigroups

9.2.1 Objects

A `Subsemigroup G` is a subset of a `Semigroup G` that is closed under the multiplication.

It's actually a bundled structure consisting of a subset and a proof of closure. To use it like a subset, Mathlib registers `Subsemigroup G` as an instance of `SetLike G`. It provides coercion from `Subsemigroup G` to `Set G`, so for `H : Subsemigroup G`, you can use `a ∈ H` to mean `a` belongs to the underlying subset of `H`.

```
section

variable (G : Type*) [Semigroup G]
variable (H₁ H₂ : Subsemigroup G) (a b : G)
example (ha : a ∈ H₁) (hb : b ∈ H₁) : a * b ∈ H₁ := mul_mem ha hb
```

the whole semigroup as a subgroup

```
#check (⊤ : Subsemigroup G)
example : (⊤ : Subsemigroup G) = ⟨Set.univ, by simp⟩ := by rfl
#synth Top (Subsemigroup G)
```

the empty subset as a subgroup

```
#check (⊥ : Subsemigroup G)
example : (⊥ : Subsemigroup G) = ⟨∅, by simp⟩ := by rfl
#synth Bot (Subsemigroup G)
```

The partial order structure on subsemigroups is inherited from subset relation of subsets.

```
example : H₁ ≤ H₂ ↔ H₁.carrier ⊆ H₂.carrier := by rfl
```

intersection of two subsemigroups

```
#check H₁ ⊓ H₂
#synth Min (Subsemigroup G)

example : H₁ ⊓ H₂ = ⟨H₁ ∩ H₂, by
    intro a b ha hb
    rcases ha with ⟨ha₁, ha₂⟩
    rcases hb with ⟨hb₁, hb₂⟩
    constructor
    all_goals apply mul_mem
    all_goals assumption
    ⟩ :=
  rfl
```

product of two subsemigroups.

```
#check H₁ ⊔ H₂
#synth Max (Subsemigroup G)
```

Definition of `H₁ ⊔ H₂` is more involved, relying the lattice structure of `Subsemigroup G`. it is defined as the infimum of all subsemigroups containing both `H₁` and `H₂`, where the infimum is given by intersection.

```
#synth CompleteLattice (Subsemigroup G)
```

This is characterized by the following properties.

```
#synth SemilatticeSup (Subsemigroup G)
example : H₁ ≤ H₁ ⊔ H₂ := by apply le_sup_left
example : H₂ ≤ H₁ ⊔ H₂ := by apply le_sup_right
example (K : Subsemigroup G) (h1 : H₁ ≤ K) (h2 : H₂ ≤ K) : H₁ ⊔ H₂ ≤ K :=
  sup_le h1 h2

end
```

9.2.2 Morphisms

Let's see how `MulHom` interacts with `Subsemigroup`.

```
section

variable {G₁ G₂ : Type*} [Semigroup G₁] [Semigroup G₂]
         (f : G₁ →ₙ* G₂) (H₁ : Subsemigroup G₁) (H₂ : Subsemigroup G₂)
```

The image of a subsemigroup under a `MulHom` is also a subsemigroup.

```
#check Subsemigroup.map f H₁
#check H₁.map f

example : Subsemigroup.map f H₁ = ⟨f '' H₁, by
    rintro x y ⟨a, ha, rfl⟩ ⟨b, hb, rfl⟩
    use a * b, H₁.mul_mem ha hb
    rw [map_mul]
    ⟩ := by rfl
```

The preimage of a subsemigroup under a `MulHom` is also a subsemigroup.

```
#check Subsemigroup.comap
#check H₂.comap f

example : Subsemigroup.comap f H₂ = ⟨f ⁻¹' H₂, by
    intro x y hx hy
    simp only [Set.mem_preimage] at hx hy ⊢
    rw [map_mul]
    exact mul_mem hx hy
    ⟩ := by rfl
```

To define the range of a `f : G₁ →ₙ* G₂`, a common idea is to adopt `(⊤ : Subsemigroup G₁).map f`. Unfortunately, this makes the underlying set being `f '' (univ : Set G₁)`, which is not definitionally equal to `Set.range f`. It will also cause `x ∈ ⊤` conditions in later proofs, redundant and annoying.

Hencee Mathlib define the range with some refinement: They manually replace the underlying set of `(⊤ : Subsemigroup G₁).map f` with `Set.range f`.

See Note range copy pattern for an official explanation.

```
#check MulHom.srange
```

the desired definitional equality

```
example : MulHom.srange f = ⟨Set.range f, by
    rintro x y ⟨a, rfl⟩ ⟨b, rfl⟩
```

```
    use a * b
    rw [map_mul]
    ⟩ := by rfl

example (x : G₂) : x ∈ MulHom.srange f ↔ x ∈ Set.range f := by rfl
#check MulHom.mem_srange -- corresponding Mathlib theorem

end
```

9.2.3 `Subsemigroup` as a type

Sometimes we treat subset as a type directly. The same applies to subsemigroups.

```
section

variable {G : Type*} [Semigroup G] (H : Subsemigroup G)

variable (a : H)
#check a
example : ↥H = {x : G // x ∈ H} := by rfl -- Hence the meaning of `a : H` is coerced
#check a.val
#check a.property

end
```

9.2.4 `MulHom` restricted to subsemigroups

Upgraded version of `Set.restrict` and `Set.codRestrict` for `MulHom`.

```
section

variable {G₁ G₂ : Type*} [Semigroup G₁] [Semigroup G₂]
         (f : G₁ →ₙ* G₂) (H₁ : Subsemigroup G₁) (H₂ : Subsemigroup G₂)

#check MulHom.restrict
#check MulHom.restrict_apply -- universal property of restriction
#check MulHom.codRestrict
#check MulHom.codRestrict_apply_coe -- universal property of codomain restriction

#check MulHom.srangeRestrict -- restriction on range

end
```

9.3 Submonoids

A `Submonoid M` is a subsemigroup of a `Monoid M` that contains the identity element.

```
section

variable (G : Type*) [Monoid G]
variable (H₁ H₂ : Submonoid G) (a b : G)

example : a ∈ H₁ → b ∈ H₁ → a * b ∈ H₁ := by apply mul_mem
example : (1 : G) ∈ H₁ := by apply one_mem
```

The whole monoid as a submonoid. Note the use of `with`, to extend the underlying `Subsemigroup` with the proof of containing `1`.

```
#check (⊤ : Submonoid G)
example : (⊤ : Submonoid G) = {(⊤ : Subsemigroup G) with
    one_mem' := by
      change 1 ∈ (⊤ : Set G)
      apply Set.mem_univ
    } := by rfl
#synth Top (Submonoid G)
```

The trivial submonoid consisting of only the identity element. Note the difference from `⊥ : Subsemigroup G`, which is the empty set.

```
#check (⊥ : Submonoid G)
example : (⊥ : Submonoid G) = {
    carrier := {1}
    one_mem' := by rfl
    mul_mem' := by
      rintro x y hx hy
      simp only [Set.mem_singleton_iff] at hx hy ⊢
      rw [hx, hy, one_mul]
    } := by rfl
#synth Bot (Submonoid G)
```

We don't repeat tedious lattice structure part, which is similar to those for `Subsemigroup`.

```
#synth CompleteLattice (Submonoid G)

end
```

9.3.1 Morphisms

`MonoidHom` interacts with `Submonoid` similarly to `MulHom` and `Subsemigroup`.

```
section

variable {G₁ G₂ : Type*} [Monoid G₁] [Monoid G₂]
```

```
        (f : G₁ →* G₂) (H₁ : Submonoid G₁) (H₂ : Submonoid G₂)
```

We still have image and preimage of submonoids, which can be built on top of those for subsemigroups, with extra care to verify the identity element membership.

```
#check Submonoid.map
example : Submonoid.map f H₁ = { Subsemigroup.map f.toMulHom H₁.toSubsemigroup with
      one_mem' := by
        simp
        use 1, H₁.one_mem
        rw [map_one]
    } := by rfl

#check Submonoid.comap
example : Submonoid.comap f H₂ = { Subsemigroup.comap f.toMulHom H₂.toSubsemigroup with
      one_mem' := by simp
    } := by rfl
```

Range is also specially handled as `MulHom.range`.

```
#check MonoidHom.mrange
example (x : G₂) : x ∈ MonoidHom.mrange f ↔ x ∈ Set.range f := by rfl
#check MonoidHom.mem_mrange -- corresponding Mathlib theorem
```

With the presence of identity element, we can define the kernel of a `MonoidHom`.

```
#check MonoidHom.mker
example : MonoidHom.mker f = (⊥ : Submonoid G₂).comap f := by rfl
```

[EXR] manual definition of `mker`

```
example : MonoidHom.mker f = {
      carrier := {x | f x = 1}
      one_mem' := by rw [Set.mem_setOf, map_one]
      mul_mem' := by
        rintro x y hx hy
        simp only [Set.mem_setOf] at hx hy ⊢
        rw [map_mul, hx, hy, one_mul]
    } := by rfl

example (x : G₁) : x ∈ MonoidHom.mker f ↔ f x = 1 := by rfl
#check MonoidHom.mem_mker -- corresponding Mathlib theorem

end
```

9.3.2 Submonoid as a type, and `MonoidHom` restriction

Tedious upgrade again. Note that `MonoidHom.mker` and `MonoidHom.mrange` steps in.

```
section

#check MonoidHom.restrict
#check MonoidHom.restrict_apply -- universal property of restriction

#check MonoidHom.codRestrict
#check MonoidHom.injective_codRestrict -- restriction on codomain preserves injectivity

#check MonoidHom.mrangeRestrict
#check MonoidHom.coe_mrangeRestrict -- universal property of range restriction
#check MonoidHom.mrangeRestrict_mker -- restriction on range preserves the kernel
end
```

9.3.3 Exercise

As an exercise, let's define addition on `AddSubmonoid A` with the intrinsic definition, and show that it coincides with the supremum.

```
section

variable {A : Type*} [AddCommMonoid A]

instance : Add (AddSubmonoid A) := ⟨fun B₁ B₂ ↦ {
  carrier := {x | ∃ b₁ ∈ B₁, ∃ b₂ ∈ B₂, x = b₁ + b₂}
  zero_mem' := ⟨0, B₁.zero_mem, 0, B₂.zero_mem, by rw [add_zero]⟩
  add_mem' := by
    rintro x y ⟨b₁, hb₁, b₂, hb₂, rfl⟩ ⟨c₁, hc₁, c₂, hc₂, rfl⟩
    use b₁ + c₁, B₁.add_mem hb₁ hc₁
    use b₂ + c₂, B₂.add_mem hb₂ hc₂
    abel
}⟩

example (B₁ B₂ : AddSubmonoid A) : B₁ ⊔ B₂ = B₁ + B₂ := by
  apply le_antisymm
  · apply sup_le
    · intro x hx
      use x, hx, 0, B₂.zero_mem
      rw [add_zero]
    · intro x hx
      use 0, B₁.zero_mem, x, hx
      rw [zero_add]
  · intro x hx
    rcases hx with ⟨b₁, hb₁, b₂, hb₂, rfl⟩
    haveI : B₁ ≤ B₁ ⊔ B₂ := le_sup_left
    replace hb₁ : b₁ ∈ B₁ ⊔ B₂ := this hb₁
    haveI : B₂ ≤ B₁ ⊔ B₂ := le_sup_right
```

```
    replace hb₂ : b₂ ∈ B₁ ⊔ B₂ := this hb₂
    exact add_mem hb₁ hb₂

end
```

9.4 Subgroups

9.4.1 Objects

There's nothing new about `Subgroup G` of a `Group G` compared to `Subsemigroup` and `Submonoid`. It just adds the closure under taking inverses.

```
section

variable (G : Type*) [Group G]
variable (H₁ H₂ : Subgroup G) (a b : G)
example : a ∈ H₁ → b ∈ H₁ → a * b ∈ H₁ := by apply mul_mem
example : (1 : G) ∈ H₁ := by apply one_mem
example : a ∈ H₁ → a⁻¹ ∈ H₁ := by apply inv_mem
```

We skip the lattice structure again.

```
end
```

9.4.2 Morphisms

`MonoidHom` works for `Subgroup` as well.

```
section

variable {G₁ G₂ : Type*} [Group G₁] [Group G₂]
        (f : G₁ →* G₂) (H₁ : Subgroup G₁) (H₂ : Subgroup G₂)
```

Image and preimage of subgroups, upgraded to show closure under inverses.

```
#check Subgroup.map
#check Subgroup.comap
```

For groups, `mker` and `mrange` has been upgraded to `ker` and `range` respectively.

```
#check MonoidHom.ker
#check MonoidHom.range

example : MonoidHom.ker f = (⊥ : Subgroup G₂).comap f := by rfl
```

```
example : MonoidHom.ker f = {MonoidHom.mker f with
      inv_mem' := by simp
    } := by rfl
```

[EXR] injectivity characterization via kernel

```
example : MonoidHom.ker f = ⊥ ↔ Function.Injective f := by
  constructor
  · intro h
    intro x y hxy
    apply_fun (· * (f y)⁻¹) at hxy
    simp only [mul_inv_cancel] at hxy
    rw [← map_inv, ← map_mul, ← MonoidHom.mem_ker, h, Subgroup.mem_bot] at hxy
    apply_fun (· * y) at hxy
    simp at hxy; exact hxy
  · intro h
    ext x
    simp only [MonoidHom.mem_ker, Subgroup.mem_bot]
    constructor
    · intro hx
      rw [← map_one f] at hx
      exact h hx
    · intro hx
      rw [hx]
      exact map_one f
#check MonoidHom.ker_eq_bot_iff -- corresponding Mathlib theorem

end
```

9.4.3 Subgroup as a type, and `MonoidHom` restriction

Similar to those for `Submonoid`.

```
section

#check MonoidHom.restrict
#check MonoidHom.restrict_apply -- universal property of restriction

#check MonoidHom.codRestrict
#check MonoidHom.ker_codRestrict -- restriction on codomain preserves the kernel
#check MonoidHom.injective_codRestrict -- restriction on codomain preserves injectivity

#check MonoidHom.rangeRestrict
#check MonoidHom.coe_rangeRestrict -- universal property of range restriction
#check MonoidHom.ker_rangeRestrict -- restriction on range preserves the kernel
#check MonoidHom.rangeRestrict_injective_iff -- restriction on range preserves injectivity

end
```

9.4.4 Normal Subgroups

For later discussions on quotient groups, we introduce normal subgroups here.

```
section

#check Subgroup.Normal
```

`Subgroup.Normal` is a bundled structure consisting of a proof of normality.

```
example {G : Type*} [Group G] (H : Subgroup G) :
    H.Normal ↔ ∀ h ∈ H, ∀ g : G, g * h * g⁻¹ ∈ H := by
  constructor
  · intro ⟨h⟩
    exact h
  · intro h
    exact ⟨h⟩
```

The kernel of a group homomorphism is a normal subgroup.

```
example {G₁ G₂ : Type*} [Group G₁] [Group G₂]
    (f : G₁ →* G₂) : (f.ker).Normal := by
  constructor
  intro x hx y
  rw [MonoidHom.mem_ker]
  rw [map_mul, map_mul, hx, map_inv, mul_one, mul_inv_cancel]
```

Actually, Mathlib contains an instance for kernels, so that Lean automatically recognizes the normality of kernels.

```
#check MonoidHom.normal_ker
example {G₁ G₂ : Type*} [Group G₁] [Group G₂]
    (f : G₁ →* G₂) : (f.ker).Normal := inferInstance

end
```

[TODO]

- Indexed infimum and supremum of substructures
- `xxxClass` for substructures as canonical maps

Chapter 10

Quotients and Quotient Groups

After building up the theory of groups, homorphisms, and subgroups, we are now ready to define quotient groups.

In fact, quotients are so fundamental that Lean makes them a primitive way of constructing new types. [IGNORE] Other reasons including the foundation of `funext`, see The Lean Language Manual.

We shall first illustrate the general quotient construction in Lean, and then specialize it to quotient groups.

At the end of the journey, we show the first isomorphism theorem for groups as promised.

```
import Mathlib
```

10.1 Quotient types

We still build up the theory from types.

10.1.1 `Equivalence`, `Setoid`, quotient types

First, we need to define equivalence relations on a type `α`. As you can guess, a binary relation `r` is just a `α → α → Prop`.

```
section

variable {α : Type*} {r : α → α → Prop} (a b c : α)
```

`Equivalence r` is a bundled structure that packages the three properties:

- reflexivity of `r`
- symmetry of `r`
- transitivity of `r`

```
variable (r_equiv : Equivalence r)

example : r a a := by exact r_equiv.refl _
example : r a b → r b a := by exact r_equiv.symm
example : r a b → r b c → r a c := by exact r_equiv.trans
```

```
end
```

As a running example, we can define an equivalence relation on `ℕ × ℕ`, which ultimately gives us the construction of integers from natural numbers.

```
section

-- tag as `simp` lemma for auto decomposition
@[simp] def NN_r (z w : ℕ × ℕ) : Prop := z.1 + w.2 = z.2 + w.1
def NN_equiv : Equivalence NN_r where
  refl := by intro _; rw [NN_r, add_comm]
  symm := by
    intro ⟨z1, z2⟩ ⟨w1, w2⟩ h
    simp at *
    linarith only [h]
  trans := by
    intro ⟨z1, z2⟩ ⟨w1, w2⟩ ⟨u1, u2⟩ h1 h2
    simp at *
    linarith only [h1, h2]
```

In mathmatics we often denote equivalence relations by `~`. In Lean, we can also use `≈` as notation for equivalence relations, once we register the `Setoid` instance for `α`.

The `Setoid` typeclass is a bundle of

- an equivalence relation `r` on `α`
- the proof that `r` is an equivalence relation.

```
#check Setoid

instance NN_setoid : Setoid (ℕ × ℕ) where
  r := NN_r
  iseqv := NN_equiv

-- tag as `simp` lemma for auto decomposition
@[simp] lemma NN0_r_of_equiv {z w : ℕ × ℕ} : z ≈ w ↔ NN_r z w := by rfl

example : (1, 2) ≈ (2, 3) := by simp
example (n m p : ℕ) : (n, m) ≈ (n + p, m + p) := by
  simp; ring
```

From a `Setoid α` instance `s`, we can define the quotient type `Quotient s`. The elements of `Quotient s`, are mathematically viewed as equivalence classes of `α`.

```
#check Quotient
```

For example, the following defines the type `Z` of integers as the quotient of `ℕ × ℕ` by the equivalence relation `NN_r`.

```
def Z := Quotient NN_setoid
```

10.1.2 The universal properties

Quotient types behave similarly to inductive types. But there is a key difference: Inductive types have their internal data directly accessible (via pattern matching and its elimination rules), while the internal data of a quotient type is hidden behind the equivalence relation.

The introduction rule of a Quotient type is given by `Quotient.mk`, essentially a map from `α → Quotient s`.

```
#check Quotient.mk

example : Z := Quotient.mk NN_setoid (3, 5)
example : Z := Quotient.mk' (3, 5) -- this version detects the `Setoid` instance in the context
example : Z := ⟦(3, 5)⟧ -- anonymous "constructor" for `Quotient`
```

Two elements of an inductive type are equal, iff they are constructed from the same data using the same constructor.

This is different in the case of quotient types, since the same equivalence class can be represented by different elements of `α`.

So quotient types need an additional "soundness" axiom to clarify when two quotient elements, coming from two possibly different representatives in `α`, are equal. It is an axiom in Lean, hence the equality is not definitional.

```
#check Quotient.sound

example : (⟦(3, 5)⟧ : Z) = ⟦(4, 6)⟧ := by
  apply Quotient.sound
  simp
```

The following two self-evident rules are the elimination rules for quotient types in `Sort*` and `Prop` respectively.

`Quotient.lift` is essentially the universal property of quotients: It allows us to define functions out of a quotient type `Quotient s` by defining them on representatives in `α`, once we verify that the function respects the equivalence relation `s.r`.

[IGNORE] Note that `Quotient.lift` does not allow a motive depending on the input, since the well-definedness needs to be verified in the same type. Quotient types do have a dependent recursor, called `Quotient.rec`. With a more complicated type signature, it is more "tricky" to use and not often needed in practice.

```
#check Quotient.lift
```

The universal property, true by definition

```
#check Quotient.lift_mk
```

As an example, we define the negation function on integers.

```
def Z_neg : Z → Z := by
  apply Quotient.lift (fun ⟨z1, z2⟩ ↦ ⟦(z2, z1)⟧ : ℕ × ℕ → Z)
  intro ⟨z1, z2⟩ ⟨w1, w2⟩ h
  dsimp
  apply Quotient.sound
  simp at *
  linarith only [h]
```

Activate the `-` notation for `Z`.

```
instance : Neg Z := ⟨Z_neg⟩
```

Define the multiplication on `Z`. It uses `Quotient.lift₂`, which is for binary operations on quotient types. Challenge: Define Quotient.lift by yourself!

```
#check Quotient.lift₂
def Z_mul : Z → Z → Z := by
  apply Quotient.lift₂
    (fun ⟨z1, z2⟩ ⟨w1, w2⟩ ↦ ⟦(z1 * w1 + z2 * w2, z1 * w2 + z2 * w1)⟧ : ℕ × ℕ → ℕ × ℕ → Z)
  intro ⟨z1, z2⟩ ⟨w1, w2⟩ ⟨u1, u2⟩ ⟨v1, v2⟩ h1 h2
  dsimp
  apply Quotient.sound
  simp at *
  nlinarith only [h1, h2]
```

Activate the `*` notation for `Z`.

```
instance : Mul Z := ⟨Z_mul⟩
```

`Quotient.ind` is also an elimination rule for quotient types, but it focuses on proving propositions about quotient types. As `Prop` is proof-irrelevant, a different proof for different representatives does not matter. Hence, the equivalence relation does not need to be verified here. ([IGNORE] and a motive depending on the input is allowed here)

In effect, `Quotient.ind` states that we may assume any quotient element is constructed from a representative in `α` when proving propositions about quotient types. Or, the canonical map `α → Quotient s` is surjective when used in `Prop`.

```
#check Quotient.ind
```

As an example, we prove that `(-z) * (-z) = z * z` for any integer `z`.

```
example : (z : Z) → (-z) * (-z) = z * z := by
  apply Quotient.ind
  intro ⟨z1, z2⟩
  apply Quotient.sound
  simp; ring
```

In tactic mode, `induction' ... using Quotient.ind with ...` is often used.

You should convince yourself that this in effect act like an `rcases ... with ...` decomposition, and `using Quotient.ind` emphasizes that we are using the elimination rule for quotients. The full use of `induction'` can be touched only when inductive types are fully covered.

```
#help tactic induction'
example (z : Z) : (-z) * (-z) = z * z := by
  induction' z using Quotient.ind with z
  rcases z with ⟨z1, z2⟩
  apply Quotient.sound
  simp; ring
```

There is also `Quotient.ind₂` for proving propositions about two quotient elements. Many other variations for `Quotient.ind` and `Quotient.lift` exist as well.

```
#check Quotient.ind₂
#check Quotient.ind'
#check Quotient.liftOn
#check Quotient.liftOn'
#check Quotient.inductionOn
#check Quotient.inductionOn'

end
```

10.2 Quotient groups

We are now ready to define quotient groups.

```
section
variable (G : Type*) [Group G] (H : Subgroup G)
```

10.2.1 Left coset relation

The notation `G / H` is, mathematically, the left cosets of `H` in `G`. In Lean, it is defined as the quotient type of `G` by the left coset equivalence relation induced by `H`.

```
#check QuotientGroup.leftRel H
```

The definition of `QuotientGroup.leftRel H` is hidden deep inside Mathlib for general usability. Practically, we only need to know its meaning (up to logical equivalence). [IGNORE] See the small section below for tracing down its definition.

```
example (a b : G) : (QuotientGroup.leftRel H) a b ↔ a⁻¹ * b ∈ H :=
  QuotientGroup.leftRel_apply
```

This in turn allows us to define the left coset type `G / H` via quotient types.

```
#synth HasQuotient G (Subgroup G)
#check G / H
```

tracing down the definition of coset relations

[IGNORE]

```
example (a b : G) : QuotientGroup.leftRel H a b = (∃ (h : H.op), b * (h : Gᵐᵒᵖ).unop = a) := calc
  QuotientGroup.leftRel H a b
    = (QuotientGroup.leftRel H).r a b := rfl
  _ = (MulAction.orbitRel H.op G).r a b := rfl
  _ = (a ∈ MulAction.orbit H.op b) := rfl
  _ = (∃ (h : H.op), h • b = a) := rfl
  _ = (∃ (h : H.op), b * (h : Gᵐᵒᵖ).unop = a) := rfl

example (a b : G) : QuotientGroup.rightRel H a b = (∃ (h : H), h * b = a) := calc
  QuotientGroup.rightRel H a b
    = (QuotientGroup.rightRel H).r a b := rfl
  _ = (MulAction.orbitRel H G).r a b := rfl
  _ = (a ∈ MulAction.orbit H b) := rfl
  _ = (∃ (h : H), h • b = a) := rfl
  _ = (∃ (h : H), h * b = a) := rfl
```

10.2.2 The group structure

With the normality of `H`, the group structure on `G / H` is induced from that of `G`.

```
variable [H.Normal]
```

We illustrate how to define the group structure on `G / H` manually here, from `Semigroup` to `Monoid` to `Group`.

Note that it is just an illustration; in practice, to construct a group structure, you may wish to use the `Group.ofLeftAxioms` constructor from Mathlib instead.

```
#synth Semigroup (G / H)
example : (QuotientGroup.Quotient.group H).toSemigroup = (show Semigroup (G / H) from {
  mul := by
    apply Quotient.lift₂ (fun (a b : G) ↦ ⟦a * b⟧)
    intro a1 a2 b1 b2 h1 h2
    apply Quotient.sound
    change QuotientGroup.leftRel H a1 b1 at h1
    change QuotientGroup.leftRel H a2 b2 at h2
    change QuotientGroup.leftRel H (a1 * a2) (b1 * b2)
    rw [QuotientGroup.leftRel_apply] at h1 h2 ⊢

    have hn : H.Normal := inferInstance; rcases hn with ⟨hn⟩
    specialize hn (a1⁻¹ * b1) h1 a2⁻¹
    simp only [inv_inv] at hn

    haveI : (a1 * a2)⁻¹ * (b1 * b2) =
        a2⁻¹ * (a1⁻¹ * b1) * a2 * (a2⁻¹ * b2) := by group
    rw [this]

    apply mul_mem hn h2

  mul_assoc := by
    intro a b c
    induction' a using Quotient.ind with a
    induction' b using Quotient.ind with b
    induction' c using Quotient.ind with c
    apply Quotient.sound
    rw [mul_assoc]
    apply refl
}) := by rfl

#synth Monoid (G / H)
example : (QuotientGroup.Quotient.group H).toMonoid = (show Monoid (G / H) from {
  one := ⟦1⟧
  one_mul := by
    intro a
    induction' a using Quotient.ind with a
    apply Quotient.sound; dsimp
    rw [one_mul]
    apply refl
  mul_one := by
    intro a
    induction' a using Quotient.ind with a
    apply Quotient.sound; dsimp
    rw [mul_one]
    apply refl
}) := by ext; rfl

#synth Group (G / H)
```

```
example : QuotientGroup.Quotient.group H = ( show Group (G / H) from {
  inv := by
    apply Quotient.lift (fun a ↦ ⟦a⁻¹⟧)
    intro a1 a2 h
    apply Quotient.sound
    change QuotientGroup.leftRel H a1 a2 at h
    change QuotientGroup.leftRel H a1⁻¹ a2⁻¹
    rw [QuotientGroup.leftRel_apply] at h ⊢
    simp only [inv_inv]

    replace h := inv_mem h
    simp at h ⊢

    have hn : H.Normal := inferInstance; rcases hn with ⟨hn⟩
    specialize hn (a2⁻¹ * a1) h a2
    simp at hn; exact hn

  div a b := a * b⁻¹

  inv_mul_cancel := by
    intro a
    induction' a using Quotient.ind with a
    apply Quotient.sound; simp
}) := by ext; rfl
```

the canonical group epimorphism from `G` to `G / H`

```
#check QuotientGroup.mk'

end
```

10.2.3 Lifting a group homomorphism

We now need to upgrade `Quotient.lift` to respect the group structure, so that we can define morphisms between quotient groups via universal properties.

```
section

variable {G M : Type*} [Group G] [Group M] (N : Subgroup G) [N.Normal]
```

Now we upgrade `Quotient.lift` to respect the group structure.

```
variable (φ : G →* M)

example (HN : N ≤ φ.ker) : (G / N) →* M where
  toFun := by
```

```
    apply Quotient.lift φ
    intro a b h
    change QuotientGroup.leftRel N a b at h
    rw [QuotientGroup.leftRel_apply] at h
    haveI := HN h; simp at this
    exact eq_of_inv_mul_eq_one this
  map_mul' := by
    intro a b
    induction' a using Quotient.ind with a
    induction' b using Quotient.ind with b
    repeat rw [Quotient.lift_mk]
    apply map_mul
  map_one' := by
    conv => lhs; rhs; change ⟦1⟧
    rw [Quotient.lift_mk]
    apply map_one

#check QuotientGroup.lift
```

10.2.4 The first isomorphism theorem

At last, we come to our grand finale.

Now we can lift the group homomorphisms `φ : G →* M` to `G / ker φ →* M`.

```
example : G / φ.ker →* M := QuotientGroup.lift φ.ker φ (by simp)
```

Recall the range restriction of `φ`.

```
example : G →* φ.range := MonoidHom.rangeRestrict φ
```

Recall the kernel of the range restriction

```
example : φ.rangeRestrict.ker = φ.ker := MonoidHom.ker_rangeRestrict φ
```

Combining above gives our desired homomorphism.

```
example : QuotientGroup.rangeKerLift φ = (show G / φ.ker →* φ.range by
  let rangeRestricted := MonoidHom.rangeRestrict φ
  apply QuotientGroup.lift φ.ker rangeRestricted
  rw [MonoidHom.ker_rangeRestrict]
) := by rfl
```

It remains to show that `QuotientGroup.rangeKerLift φ` is an isomorphism. Let's attack this by showing it's both injective and surjective.

```
#check MonoidHom.ker_eq_bot_iff -- recall the kernel criterion for injectivity
example : Function.Injective (QuotientGroup.rangeKerLift φ) := by
  rw [← MonoidHom.ker_eq_bot_iff, Subgroup.eq_bot_iff_forall]
  intro gq hgq

  induction' gq using Quotient.ind with g
  unfold QuotientGroup.rangeKerLift MonoidHom.rangeRestrict at hgq
  simp only [MonoidHom.mem_ker, QuotientGroup.lift_mk, MonoidHom.codRestrict_apply] at hgq
  apply_fun Subtype.val at hgq; dsimp at hgq

  apply Quotient.sound
  change QuotientGroup.leftRel _ _ _
  rw [QuotientGroup.leftRel_apply]
  simp only [mul_one, inv_mem_iff, MonoidHom.mem_ker]

  exact hgq
#check QuotientGroup.rangeKerLift_injective -- corresponding Mathlib lemma

example : Function.Surjective (QuotientGroup.rangeKerLift φ) := by
  unfold QuotientGroup.rangeKerLift MonoidHom.rangeRestrict
  rintro ⟨m, g, rfl⟩
  use ⟦g⟧; simp

#check QuotientGroup.rangeKerLift_surjective -- corresponding Mathlib lemma
```

Recall that we've shown before in the exercises that an `Equiv` can be reached from a bijective function. Nothing special here for `MulEquiv`.

```
#check Equiv.ofBijective
#check MulEquiv.ofBijective
```

The First Isomorphism Theorem for groups.

```
example : QuotientGroup.quotientKerEquivRange φ = (show G / φ.ker ≃* φ.range from
  MulEquiv.ofBijective
    (QuotientGroup.rangeKerLift φ)
    ⟨QuotientGroup.rangeKerLift_injective φ, QuotientGroup.rangeKerLift_surjective φ⟩
) := by rfl

end
```

Appendix A

形式化数学与 Lean 4 定理证明入门

Introduction to Formal Mathematics with Lean 4

A.1 背景简介

形式化数学是将数学定义、定理和证明转化为计算机可验证的精确形式的过程。可以认为，数学形式化 = 编写程序，而正确的证明 = 代码通过编译。严谨性是数学研究的基石，形式化数学通过严格的逻辑框架和程序语言确保数学结论的正确性和可复用性，在现代数学体系日渐庞大复杂的背景下具有重要意义。

Lean 4 是一个专为形式化数学设计的编程语言和证明助手，支持数学家将数学定理转化为计算机可验证的代码。它具备高效的编译器和灵活的类型系统，适合构建复杂的数学证明；其数学库 Mathlib 正在飞速扩展，目前已覆盖本科数学大部分内容。Lean 4 已成为目前数学形式化工作的主流选择。

AI4Math 是将人工智能技术应用于数学研究的跨学科领域。其目标是通过机器学习、大语言模型等方法，简化形式化过程中的繁重代码编写工作，辅助数学家进行数学形式化、定理证明乃至提出猜想。AI4Math 或将在将来大幅提高数学形式化和数学研究的效率。

A.2 课程信息

本兴趣课程旨在向数学或计算机相关专业学生普及形式化数学的基本概念和方法，掌握 Lean 4 定理证明的基本技能，并了解 AI4Math 的最新进展，为后续更深入的探索做好引导。课程设计为 2025 秋季学期范洋宇老师《抽象代数》的配套课余活动，欢迎感兴趣的同学参与。

本课程原则上不要求任何编程或数学背景，但建议具备至少一侧的常识。我们建议有意动手实操的同学携带电脑，并提前配置好网络、Git、VSCode 等相关环境。

时间：1–12 周周三 18:30–20:05
地点：文萃楼 F502
主讲人：钟星宇

北京理工大学 2022 级强基数学专业本科生
Mathlib 4 Contributor
ICPC Regional Silver Medalist
2025 BICMR–RUC 代数与形式化暑期学校学员
BICMR–Ubiquant AI4Math 数据标注团队实习经验

A.3 课程大纲

Main topics:

- Introduction to Formal Mathematics with Lean 4
- Logic and Proofs
- The Type Universe and Equality
- Inequalities
- Mathematical Analysis: Taking Limits on the Real Numbers
- Abstract Algebra: Groups and Homomorphisms
- Substructures and Subgroups
- Quotient Types and Quotient Groups

If time permits:

- Inductive Types and Induction Methods
- Classes and Instances
- Coercions

A.4 参考材料

- Introductory:
  - CAV2024
  - Terence Tao at IMO 2024: AI and Mathematics
  - Lean 的前世今生
  - Natural Number Game
  - Computational Trilogy - nLab
- Bibles
  - Mathematics in Lean 4
  - Theorem Proving in Lean 4
  - Lean Language Manual
  - Type Theory - nLab
  - Other bibles
- Courses
  - Kevin Buzzard's 2024 course on formalising mathematics in the Lean theorem prover

A.5 链接

- Course materials:
  - course repository
  - online documentation
- Online compiler:
  - Lean 4 Web
- Community
  - Lean Zulip
- Miscellaneous
  - Lean 4 tactics cheatsheet