05-type.lean

```
import Mathlib
```

`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` (`=`) (first visit)
  - Arithmetic in `CommRing`
- Defining terms and functions
  - Definitional equality vs propositional equality

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.

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.

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`.

3 Equality `Eq` (`=`) (first visit)

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. We postpone the real, full discussion of equality to later chapters.

```
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.
```

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]
```

3.2 Working in `CommRing`

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

3.2.1 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]
```

3.2.2 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
```

3.2.3 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]
```

3.2.4 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
```

3.2.5 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`
```

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]
```

4 Defining terms and functions

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.

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
```

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
```

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
```

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.

4.4.1 `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
```

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
```

4.4.2 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
```

4.4.3 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
```

5 Equality (second visit)

[TODO] (This section is for future chapters.)

```
#print Eq
#print Equivalence
#print Setoid
#print PartialOrder
```