Introduzione alla formalizzazione della matematica in Lean
Lesson 4/10 Real numbers

Yoh Tanimoto

Corso di dottorato, University of Rome “Tor Vergata”

18 Dicembre 2025

References

- We follow Mathematics in Lean, Chapters 2 and 3.
- Make sure to install Lean, Git and to `clone` the repository (follow this link) (or use a desktop app).

Using `mathlib`

- We saw
  - how to define `Nat = ℕ` inductively.
  - how to prove `add_comm (m n : ℕ) : m + n = n + m`.
- We can define the product in $\mathbb{N}$ and prove all the properties.
- We can define $\mathbb{Z}, \mathbb{Q}, \mathbb{R}$ and prove all the known properties.
- That has been done and we do not have to repeat it by ourselves. Instead, **we can use the definitions and theorems in** `mathlib`.
- For example, `Nat.add_comm` is proved here.
- (if you don't want trust `mathlib`... you can either check the definitions or the theorems showing that $\mathbb{R}$ is a complete ordered field (and hope that Lean's system is sound))
- Today we see how to use theorems about $\mathbb{R}$ that are already proven and see that we can talk about algebraic and analytic properties.

Using `mathlib`

- The `mathlib` definition of the type of real numbers.
- (`Real` (= $\mathbb{R}$ is defined as the completion of the equivalence classes of Cauchy sequences in $\mathbb{Q}$)
- Actually, the definitions *do not matter*.
- What matters is that, from this definition, most of the properties of $\mathbb{R}$ one nees have been proved, and we can use them.
- For practical purposes it is important to know *how to find* such theorems, and *how to use them*.
- Today we will see how to use given theorems in `mathlib`.
- (When we need some theorem, the textbook tells us which to use)

Some definitions and theorems about $\mathbb{R}$

Let us start with typing the following.
(open folder `mathematics_in_lean`, then open the files in `MIL/C03_Logic/`)

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

Observe that

- $\leq, |\mathtt{x}|$, `x * y` are already defined and you can use them (thanks to `import` lines at the beginning)
- If you write a meaningful sentence, it gives a `Prop` object.
- In principle you can prove them using the definition, but...
- That has been done by other people. We will learn how to use them.

Some definitions and theorems about $\mathbb{R}$

(actually, the first one is simple enough that Lean can find it's proof, while the second one is not)

```
example : ∀ x : ℝ, 0 ≤ x → |x| = x := by
  exact?
example : ∀ x y ϵ : ℝ, 0 < ϵ → ϵ ≤ 1 → |x| < ϵ → |y| <
ϵ → |x * y| < ϵ := by
  exact?
```

We can leave `sorry` and make it a `theorem`. Then we can use it.

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

We can see on which axioms `my_lemma` depends. It includes `sorryAx`, which means that the proof is left as `sorry`.

Variable declarations

We can declare variables as follows. Then they are referred in a theorem, they behave as a bound variable $\forall$ `x`.
It is also possible to declare hypothese as variables. In the following example, `hxpos` is an object (proof) of the type `0 < x`.

```
variable (a : ℝ)
variable (b δ : ℝ)
variable (hδpos : 0 < δ)
```

After declaration, we can use them in lemmas.

```
#check my_lemma a b δ
#check my_lemma a b δ h₀ h₁
variable (x y : ℝ)
#check my_lemma x y δ h₀ h₁
```

Note that the resulting proof depends on the variable used.

Implicit variables

When we write a theorem, we can leave some of the variable *implict*, by using $\{, \}$, if there is an enough context to determine them (that is, if these variables are referred to in the *explicit* hypothesis).

```
theorem my_lemma2 : {∀ x y ϵ : ℝ}, 0 < ϵ → ϵ ≤ 1
    → |x| < ϵ → |y| < ϵ → |x * y| < ϵ := by
  sorry
section
variable (a b δ : ℝ)
variable (h₀ : 0 < δ) (h₁ : δ ≤ 1)
variable (ha : |a| < δ) (hb : |b| < δ)
#check my_lemma2 h₀ h₁ ha hb
#check @my_lemma2
```

Lean can determine `a, b, δ` because they are referred to in `ha, hb`.
If you want to specity the variables explicitly, write `@` at the beginning of the theorem name.

Predicate and hypothese

One can also define a predicate.
A predicate followed by (an) object(s) give `Prop`.
A variable of that type is a proof (hypothesis on these variables).

```
def FnUb (f : ℝ → ℝ) (a : ℝ) : Prop :=
  ∀ x, f x ≤ a
def FnLb (f : ℝ → ℝ) (a : ℝ) : Prop :=
  ∀ x, a ≤ f x
variable (f : ℝ → ℝ) (a : ℝ)
#check FnUb f a
variable (hfUba : FnUb f a)
```

General structures

`mathlib` contains a huge amount of general structures and theorems about them.
When one wants to talk about general group, one can do as follows, which means that `G` is some type with a group structure. (more details in Lecture 7)

```
variable {G : Type*} [Group G]
```

We can talk about a general partially ordered type.

```
variable {α : Type*} [PartialOrder α]
variable (s : Set α) (a b : α)
def SetUb (s : Set α) (a : α) :=
  ∀ x, x ∈ s → x ≤ a
example (h : SetUb s a) (h' : a ≤ b) : SetUb s b :=
  sorry
```

(hint : use `apply le_trans`)

Casting

In Lean, we can write `0, 1, 2` and they are, if not specified, interpreted as `Nat = ℕ`. If we want `0` as a real number, we need a casting.

```
import Mathlib
#check 0
#check 1
#check (0 : ℝ)
#check (0 : ℕ) = (0 : ℝ)
```

link

Look at the output of the last line: In Lean, `(0 : ℕ) = (0 : ℝ)` **does not make sense** as is. In other words, `(0 : ℕ)` is **different** from `(0 : ℝ)`.

Here, the equality makes sense because Lean automatically cast `(0 : ℕ)` to `(0 : ℝ)`, note $\uparrow$ in front of the left-hand side. This is called **coercion**.

In this case, Lean applies the canonical embedding called `Nat.cast`.

Tactic norm_num

When we want to prove something regarding purely numbers (not including variables), we can use the tactic `norm_num`.

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

link

If you would like to understand what is going on behind, you can always add `show_term`, but it will give a unreadable proof term...

Tactics ring, linarith, field_simp

When we want to prove something regarding purely algebraic (including variables) or elementary inequalities, we can use the tactics `ring`, `linarith`, `field_simp`.

```
import Mathlib
example (a b c : ℝ) (ha : a ≠ 0) :
    (b + c / a) ^ 2 = b ^ 2 + 2 * b * c / a + c ^ 2 / a ^ 2
    := by
  field_simp [ha]
  ring
```

link

If you would like to understand what is going on behind, you can always add `show_term`, but it will give a unreadable proof term...

Tactic ext

When we want to prove that two functions are equal, we can use `ext` to introduce an arbitrary variable and move to showing the equality of values.

```
import Mathlib
example :  (fun x :  ℝ ↦ (x + 1) ^ 2) = fun x :  ℝ ↦ x ^
2 + 2 * x + 1
    := by
  ext x
  ring
```

link

If we do not need `x`, we can leave it blank.

Tactic congr

When we want to prove that two values of the same are equal, it is sufficient (of course not necessary) to show that the two variables are equal, using `congr`.

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

link

Practice

- Let us do some exercises in Sections 2 and 3.