Introduzione alla formalizzazione della matematica in Lean
Lesson 5/10 Sets and functions

Yoh Tanimoto

Corso di dottorato, University of Rome "Tor Vergata"

08 Gennaio 2026

References

- We follow Mathematics in Lean, Chapter 4.
- Make sure to install Lean, Git and to `clone` the repository (follow this link) (or use a desktop app).

Operations about sets

To prove an inclusion of sets, we can start with `intro`.

```
import Mathlib
variable (X : Type*) (s :  Set X)
example :  s ⊂ s := by
  intro x hx
  exact hx
```

- This works because internally `s ⊂ t` is defined as `∀ x, x ∈ s → x ∈ t`.
- It is also useful to remember that
  - `x ∈ s ∩ t` is defined to be `x ∈ s ∧ x ∈ t` and we can use `.left`, `.right` or the tactic `constructor`
  - `x ∈ s ∪ t` is defined to be `x ∈ s ∨ x ∈ t` and we can use the tactic `cases` or the tactic `left, right`

Operations about sets

To prove an equality of sets, we can start with `ext`.

```
import Mathlib
variable (X : Type*) (s t :  Set X)
example (hst :  s ⊂ t) (hts :  t ⊂ s) :  s = t := by
  ext x
  constructor
  · intro h
    exact hst h
  intro h
    exact hts h
```

- This works because internally `s = t` is defined as an equality of functions, where `s :  X → Prop`.
- `P ↔ Q` can be proved using `constructor`.
- If `h :  P ↔ Q`, then `h.mp :  P → Q` and `h.mpr :  Q → P` (`mp` stands for modus ponens).

Operations about sets

The set difference `x ∈ s \ t` is defined as `x ∈ s ∧ x ∉ t`, and `x ∉ t` means `¬ x ∈ t`.

```
import Mathlib
variable (X : Type*) (s t :  Set X)
example :  (s \ t) ∪ t ⊂ s ∪ t := by
  intro x hx
  rcases hx with h | h
  · left
    exact h.1
  · right
    exact h
```

link

- It is useful to remember that `¬ p` is defined to be `p → False`.

Sets by specification

For `X : Type` and a predicate `p : X → Prop`, we can define the set of elements satisfying `p` by `{ x : X | p x }`.

```
import Mathlib
def evens : Set ℕ :=
  { n | Even n }
```

link

- Here, `Even : ℕ → Prop` is defined by `∃ (r : ℕ), a = r + r`, link
- (actually it is defined not only for $\mathbb{N}$, but any type with addition $+$. Note `[Add α]`)
- The whole set is called `Set.univ X`, defined as `{ x : X | True}`
- The empty set $\emptyset$ is defined as `{ x : X | False}`

Indexed sets

In Lean, an indexed sets in `X` with the index type `I` can be realized as a function `X : I → Set X`.
Indexed union and indexed intersection are denoted by `⋃ i, X i` and `⋂ i, X i`, respectively. In `mathlib`, they are called `iUnion, iInter`.

```
import Mathlib
variable {X I : Type*} (A B : I → Set X) (s :  Set X)
example :  (s ∩ ⋃ i, A i) = ⋃ i, A i ∩ s := by
  ext x
  simp only [Set.mem_inter_iff, Set.mem_iUnion]
  constructor
  · rintro ⟨xs, ⟨i, xAi⟩⟩
    exact ⟨i, xAi, xs⟩
  rintro ⟨i, xAi, xs⟩
  exact ⟨xs, ⟨i, xAi⟩⟩
```

link

Indexed sets

If we want to take a bounded union/intersection (over a subset `s`), one can write `⋃ i ∈ s, X i` and `⋂ i ∈ s, X i`, respectively. In `mathlib`, they are called $\texttt{iUnion}_2$, $\texttt{iInter}_2$.

```
import Mathlib
def primes : Set ℕ :=
  { x | Nat.Prime x }
example : (⋃ p ∈ primes, { x | p ^ 2 | x }) = { x | ∃ p ∈
primes, p ^ 2 | x } := by
  ext
  rw [Set.mem_iUnion₂]
  simp
```

link

- Here `simp` applies lemmas (theorems) marked with `@[simp]`.
- There many such `simp` lemmas. It is useful just to try `simp` in a proof.

Sets of sets

If `s : Set (Set X)` is a collection of sets in `X`, one can write the union and intersection of sets in `s` as $\bigcup_0$ `s` and $\bigcap_0$ `s`, respectively. In `mathlib`, they are called `sUnion, sInter`.

```
import Mathlib
variable {X : Type*} (s : Set (Set X))
example : ⋃₀ s = ⋃ t ∈ s, t := by
  ext x
  rw [Set.mem_iUnion₂]
  simp
```

link

Functions

If `X Y : Type`, any `f : X → Y` is a function from `X` to `Y`.

Differently from the usual mathematics, it is more common to consider `f` on the whole type `X` rather than considering the domain of `f`.
Here the division of rational numbers, `a / b`, is **defined to be** `0` if `b = 0`.

Instead, **properties of the division** are often stated with the assumption that `b ≠ 0`. This is because in Lean one can define functions or operations between types, and it is very inconvenient to introduce $\mathbb{Q} \setminus \{0\}$.

(For example, `x = y` follows from `x / a = y / a` only under `a ≠ 0`. `div_left_inj'`)

Axioms

Although Lean and `mathlib` are written in dependent type theory and not set theory, there are some additional axioms that are almost always used.

- `propext : {a b : Prop} : (a ↔ b) → a = b`.
  (equivalent propositions are the same. This introduces the equality `=` in `Prop`)
- `Classical.choice {X : Sort X} : Nonempty X → X`.
  (the axiom of choice. link)

```
variable (I : Type) (X : I → Type)
  (h : ∀ (i : I), Nonempty (X i))
def f := fun (i : I) => Classical.choice (h i)
```

- `Quot.sound {X : Sort u} {r : X → X → Prop} {a b : X} : r a b → mk r a = mk r b`. (Quotient. If `r` is an equivalence relation, then the relation `r a b` means that `a/r = b/r`)

Image and preimage of a function

If `X Y : Type`, `f : X → Y` and `s : Set X`, `t : Set Y`, we can consider

- The image of `s` under `f`: `f '' s` or `image f s`. It is defined as $\{$`y | ∃ x, x ∈ s ∧ f x = y`$\}$.
- The preimage of `t` under `f`: `f ⁻¹' t` or `preimage f t`. It is defined as $\{$`x | f x ∈ t`$\}$

Injectivity and surjectivity

If `X Y : Type`, `f : X → Y` mathlib has the standard predicates:

- `Function.Surjective f`: `∀ (b : Y), ∃ (a : X), f a = b`
- `Function.Injective f` : `∀ {a b : X}, f a = f b → a = b`

Here, the curly brackets means that, when we use `Function.Injective f`, we do not have to write `a b` explicitly, but Lean tries to find the write terms.

```
open Function
example (h : Injective f) : f⁻¹' (f '' s) ⊂ s := by
  intro x hx
  simp at hx
  obtain ⟨y, hy⟩ := hx
  rw [← h hy.2]
  exact hy.1
```

link

Injectivity and surjectivity

If `X Y : Type`, `f : X → Y` and `s : Set X`, `t : Set Y`, mathlib has the standard predicates:

- `Set.InjOn f s`: `f` is injective on `s`.
- `Set.SurjOn f s t` : `f` is injective on `s` to `t`.

Moreover,

- `range f`: `{x | ∃ y, f y = x}`.

```
open Function
example : InjOn (fun x => x ^ 2) { x : ℝ | x ≥ 0 } := by
  intro x hx y hy h
  simp at h; simp at hx; simp at hy
  rw [← abs_eq_self.mpr hx, ← abs_eq_self.mpr hy]
  exact (sq_eq_sq_iff_abs_eq_abs x y).mp h
```

link

Inverse function

Let `X Y : Type`, `f : X → Y`.
Assume that `Y` is nonempty. We can define the left invese function, even if `f` is nether injective nor surjective.

```
variable {X Y : Type*} [Inhabited X]
#check (default : X)
variable (P : X → Prop) (h : ∃ x, P x)
#check Classical.choose h
example : P (Classical.choose h) :=
  Classical.choose_spec h
```

link

- `[Inhabitated X]` means that there is a specific element `default : X`.
- `Classical.choose` gives an object of the given type, and `Classical.choose_spec` gives the property.

Inverse function

Let `X Y : Type`, `f : X → Y`. Assume that `Y` is nonempty. We can define the left invese function, even if `f` is nether injective nor surjective. (In `mathlib` this is `invFun`)

```
noncomputable section
open Classical

def inverse (f : X → Y) : Y → X := fun y : Y =>
  if h : ∃ x, f x = y then Classical.choose h else default

theorem inverse_spec {f : X → Y} (y : Y) (h : ∃ x, f x
= y) : f (inverse f y) = y := by
  rw [inverse, dif_pos h]
  exact Classical.choose_spec h
```

link

- Here we define `inverse f y` as some `x` (using the axiom of choice) if `x` exist such that `f x = y`, otherwise it is `default`.

Practice

- Let us do some exercises in Section 4.