Introduzione alla formalizzazione della matematica in Lean
Lesson 6/10 Structures

Yoh Tanimoto

Corso di dottorato, University of Rome “Tor Vergata”

14 Gennaio 2026

References

- We follow Mathematics in Lean, Chapters 7.
- Make sure to install Lean, Git and to `clone` the repository (follow this link) (or use a desktop app).

structure

We can collect data and create new types.

```
structure Point where
  x : ℝ
  y : ℝ
  z : ℝ
```

`P : Point` has components `P.x, P.y, P.z : ℝ`. To make an object,

```
def myPoint1 : Point where
  x := 2
  y := -1
  z := 4
def myPoint2 : Point :=
  ⟨2, -1, 4⟩
def myPoint3 :=
  Point.mk 2 (-1) 4
```

structure

By marking `@[ext]`, Lean automatically generates a lemma that two objects of type `Point` are the same when their components are the same, so we can use the tactic `ext`.

```
@[ext]
structure Point where
  x : ℝ
  y : ℝ
  z : ℝ

example (a b : Point) (hx : a.x = b.x) (hy : a.y = b.y)
(hz : a.z = b.z) : a = b := by
  ext
  repeat' assumption
```

Operations on `structure`

We can define operations on `Point` as a function taking two objects of `Point` and one output of `Point`.

```
namespace Point
def add (a b :  Point) :  Point :=
  ⟨a.x + b.x, a.y + b.y, a.z + b.z⟩

def add' (a b :  Point) :  Point where
  x := a.x + b.x
  y := a.y + b.y
  z := a.z + b.z
```

- By writing `namespace Point`, any definition of, say, `XXX` below this line has actually the name `Point.XXX`.
- Inside the namespace, it can be referred to as `XXX`.
- When there are `a b :  Point` (with the same name as the namespace), we can also write `a.add b` instead of `Point.add a b`.

namespace and structure

If we want to avoid the effect of namespace, we can add `protected`. In this way, the new theorem must be referred to as `Point.add_comm`. On the other hand, there is a generic theorem `add_comm` here.

```
namespace Point

protected theorem add_comm (a b :  Point) :
    add a b = add b a := by
  rw [add, add]
  ext <;> dsimp
  repeat' apply add_comm

example (a b :  Point) :  add a b = add b a := by
  simp [add, add_comm]
```

link

- Here in the last `simp`, the generic `add_comm` is used.

Some automatization

Because Lean can unfold definitions and simplify projections internally, sometimes the equations we want hold definitionally.

```
theorem add_x (a b :  Point) :  (a.add b).x = a.x + b.x :=
  rfl
```

- When you want to prove something, you can try
  - `rfl` (unfold some definitions)
  - `exact?` (find an exact proof)
  - `simp` (apply some lemmas marked with `@[simp]`)
  - `grind` (combine current hypotheses)
  - `apply?` (match the goal with the conclusion of lemmas)
- Tactic reference

structure bundling properties

We can add `Prop` in a structure.

```
@[ext]
structure StandardTwoSimplex where
  x : ℝ
  y : ℝ
  z : ℝ
  x_nonneg : 0 ≤ x
  y_nonneg : 0 ≤ y
  z_nonneg : 0 ≤ z
  sum_eq : x + y + z = 1
```

- To show that two objects are the same, we only have to show the equality of data components, not `Prop`.

structure bundling properties

To construct an object in a structured type with properties, we need to prove them (add their proofs).

```
def swapXy (a :  StandardTwoSimplex) :  StandardTwoSimplex
    where
  x := a.y
  y := a.x
  z := a.z
  x_nonneg := a.y_nonneg
  y_nonneg := a.x_nonneg
  z_nonneg := a.z_nonneg
  sum_eq := by rw [add_comm a.y a.x, a.sum_eq]
```

structure with parameters

We can define the n-dimensional standard simplexes at once.

```
open BigOperators

structure StandardSimplex (n :  ℕ) where
  V : Fin n → ℝ
  NonNeg :  ∀ i :  Fin n, 0 ≤ V i
  sum_eq_one :  (∑ i, V i) = 1
```

- BigOperators allows us to use the notation $\sum$ over finite type.
- Fin n is the type consisting of $\{0, 1, \cdots, n-1\}$.
- As V : Fin n → ℝ, Lean recognizes automatically that, whenever we write V i, i has type Fin n.

structure with parameters

```
namespace StandardSimplex
def midpoint (n : ℕ) (a b : StandardSimplex n) :
    StandardSimplex n where
  V i := (a.V i + b.V i) / 2
  NonNeg := by
    intro i
    apply div_nonneg
    · linarith [a.NonNeg i, b.NonNeg i]
    norm_num
  sum_eq_one := by
    simp [div_eq_mul_inv, ← Finset.sum_mul,
      Finset.sum_add_distrib, a.sum_eq_one, b.sum_eq_one]
    field_simp
```

`norm_num` does numerical operations, `field_simp` removes denominators.

structure with Prop

We can also bundle properties of an object, by taking an object as a parameter.
`IsLinear f` has `Prop`.

```
structure IsLinear (f : ℝ → ℝ) where
  is_additive : ∀ x y, f (x + y) = f x + f y
  preserves_mul : ∀ x c, f (c * x) = c * f x

section
variable (f : ℝ → ℝ) (linf : IsLinear f)
#check linf.is_additive
#check linf.preserves_mul

end
```

Subtypes

For a given type, one can consider a type of object satisfying certain properties, using a notation similar to that of the set theory.

```
def PReal := { y :  ℝ // 0 < y }

variable (x :  PReal)

#check x.val
#check x.property
#check x.1
#check x.2
end
```

“Algebraic” structures

In Lean, we can define the following mathematical structures and we do have them in `mathlib`.

- partially ordered set
- group
- commutative group
- lattice (partially ordered sets)
- ring
- ordered ring
- metric space
- topological space

Each of them come with an underlying type with operations or relations. There is a hierarchy of structures: group $>$ semigroup $>$ monoid $>$ magma

The group structure

A given type `X`, we can define a group structure on it.

```
structure Group₁ (X : Type*) where
  mul : X → X → X
  one : X
  inv : X → X
  mul_assoc : ∀ x y z : X,
    mul (mul x y) z = mul x (mul y z)
  mul_one : ∀ x : X, mul x one = x
  one_mul : ∀ x : X, mul one x = x
  inv_mul_cancel : ∀ x : X, mul (inv x) x = one
```

The group of permutations

Given types `X Y`, we can consider all bijections between them. It is defined as `Equiv X Y` in `mathlib`, with the notation `X ≃ Y`.

```
variable (X Y Z : Type*)
variable (f : X ≃ Y) (g : Y ≃ Z)
#check Equiv X Y
#check (f.toFun : X → Y)
#check (f.invFun : Y → X)
#check (f.right_inv : ∀ x : Y, f (f.invFun x) = x)
#check (f.left_inv : ∀ x : X, f.invFun (f x) = x)
#check (Equiv.refl X : X ≃ X)
#check (f.symm : Y ≃ X)
#check (f.trans g : X ≃ Z)
```

The group of permutations

We can give `Equiv X X` a $\texttt{Group}_1$ structure.

```
def permGroup {X : Type*} :  Group₁ (Equiv X X)
    where
  mul f g := Equiv.trans g f
  one := Equiv.refl X
  inv := Equiv.symm
  mul_assoc f g h := (Equiv.trans_assoc _ _ _).symm
  one_mul := Equiv.trans_refl
  mul_one := Equiv.refl_trans
  inv_mul_cancel := Equiv.self_trans_symm
```

`one` is the unit element, `inv` takes an element to its inverse,...

The group of permutations

Actually, `mathlib` has the definition `Group` and a `Group` structure on `Equiv X X`. Moreover, we can use the usual group notations!

```
variable X : Type* (f g :  Equiv X X) (n :  ℕ)

#synth Group (Equiv X X)
#check f * g

example :  f * g * g⁻¹ = f := by rw [mul_assoc,
  mul_inv_cancel, mul_one]

example :  f * g * g⁻¹ = f :=
  mul_inv_cancel_right f g
```

We will see how this is possible, using logic, implicit arguments and *type class inference*. This will also explain the square bracket variable, `[Group (Equiv X X)]`

The group class

Instead of `structure`, let us use the keyword `class`.

```
class Group₂ (X : Type*) where
  mul : X → X → X
  one : X
  inv : X → X
  mul_assoc : ∀ x y z : X,
    mul (mul x y) z = mul x (mul y z)
  mul_one : ∀ x : X, mul x one = x
  one_mul : ∀ x : X, mul one x = x
  inv_mul_cancel : ∀ x : X, mul (inv x) x = one
```

An instance of a class

Instead of `def`, let us use the keyword `instance` (no need to give a name).

```
instance {X : Type*} :  Group₂ (Equiv X X)
    where
  mul f g := Equiv.trans g f
  one := Equiv.refl X
  inv := Equiv.symm
  mul_assoc f g h := (Equiv.trans_assoc _ _ _).symm
  one_mul := Equiv.trans_refl
  mul_one := Equiv.refl_trans
  inv_mul_cancel := Equiv.self_trans_symm
```

An use of instances

After these preparations, the following code works.

```
def mySquare {Z : Type*} [Group₂ Z] (x :  Z) :=
  Group₂.mul x x

variable {Y : Type*} (f g :  Equiv Y Y)

example :  Group₂.mul f g = g.trans f := rfl
example :  mySquare f = f.trans f := rfl
```

- For any type `Z`, assuming that it has a `Group₂` structure by writing `[Group₂ Z]`, the notation `Group₂.mul x x` makes sense.
- (it is also possible to give a name to an instance, say writing `[g2Z : Group₂ Z]`)
- As we wrote an `instance` of `Group₂ (Equiv X X)` for any type X, `Equiv Y Y` is given automatically a `Group₂` structure.

Notations

`mathlib` has generic classes for notations, such as

- `Add` for $+$
- `Mul` for $*$
- `Inv` for $^{-1}$

By registering instances for `Group₂`, we can use these notations.

```
instance {Z : Type*} [Group₂ Z] : Mul Z := ⟨Group₂.mul⟩
instance {Z : Type*} [Group₂ Z] : One Z := ⟨Group₂.one⟩
instance {Z : Type*} [Group₂ Z] : Inv Z := ⟨Group₂.inv⟩

variable {X : Type*} (f g :  Equiv X X)
#check f * 1 * g⁻¹
def foo :  f * 1 * g⁻¹ = g.symm.trans ((Equiv.refl Z).trans
f) := rfl
```

If there are more instances of a class, Lean considers the last one if no priority is given.

Practice

- Let us do some exercises in Section 7.