Introduzione alla formalizzazione della matematica in Lean
Lesson 9/10 More advanced structures

Yoh Tanimoto

Corso di dottorato, University of Rome "Tor Vergata"

28 Gennaio 2026

References

- We follow Mathematics in Lean, Chapters 10, 11.
- Make sure to install Lean, Git and to `clone` the repository (follow this link) (or use a desktop app).

Vector spaces

We can define a vector space as

```
variable {K : Type*} [Field K] {V : Type*} [AddCommGroup V]
[Module K V]
```

that is, `K` is a field, `V` is an additive (commutative) group and `V` is a module of `K`. The addition and scalar multiplication are denoted by `+` and `•`. Basic properties are present in `mahtlib`.

```
example (a :  K) (u v :  V) : a • (u + v) = a • u + a • v
    := smul_add a u v
example (a b :  K) (u :  V) : (a + b) • u = a • u + b • u
    := add_smul a b u
example (a b :  K) (u :  V) : a • b • u = b • a • u
    := smul_comm a b u
```

Linear maps

Between two vector spaces (with the same scalar), we have linear maps.

```
variable {W : Type*} [AddCommGroup W] [Module K W]
variable (φ :  V →ₗ[K] W)

example (a :  K) (v :  V) : φ (a • v) = a • φ v :=
    map_smul φ a v
example (v w :  V) : φ (v + w) = φ v + φ w :=
    map_add φ v w
```

`LinearMap` is a `structure` having the following fields link. Using `LinearMap.mk` which takes these variables, we can make a `LinearMap`.

```
toFun :  M → M₂
map_add' (x y :  M) :
  self.toFun (x + y) = self.toFun x + self.toFun y
map_smul' (m :  R) (x :  M) :
  self.toFun (m • x) = m • self.toFun x
```

Linear maps

There are natural operations such as scalar multiplication and addition on linear maps. This means that `AddCommGroup, Module` instances are declared for `LinearMap`.

```
variable (φ ψ :  V →ₗ[K] W)
#check (2 • φ + ψ :  V →ₗ[K] W)
```

When we have two linear maps between vector spaces, we can compoose them. (Note that ∘ gives only an object of type `W → W`, losing the linearity)

```
variable (θ :  W →ₗ[K] V)
#check (φ.comp θ :  W →ₗ[K] W)
#check (φ ∘ₗ θ :  W →ₗ[K] W)
```

Direct sums of vector spaces

When we have two vector spaces `V W`, the direct sum is realized on the type `V × W` (that is the type of ordered pairs) with the natural linear structure. In `mathlib`, `V × W` carries `AddCommGroup, Module` instances. More natural operations are defined as follows.

```
variable {W : Type*} [AddCommGroup W] [Module K W]
variable {U : Type*} [AddCommGroup U] [Module K U]
variable {T : Type*} [AddCommGroup T] [Module K T]
-- First projection map
example :  V × W →ₗ[K] V := LinearMap.fst K V W
-- Second projection map
example :  V × W →ₗ[K] W := LinearMap.snd K V W
-- First inclusion map
example :  V →ₗ[K] V × W := LinearMap.inl K V W
-- Second inclusion map
example :  W →ₗ[K] V × W := LinearMap.inr K V W
```

Indexed direct sums, tensor products

For an indexed family of vector spaces, we can define their direct sum, with the `DirectSum` namespace.

```
variable {ι K : Type*} (V : ι → Type*) [Field K] [∀ i,
AddCommGroup (V i)] [∀ i, Module K (V i)]
open scoped DirectSum
#check (⨁ i, V i)
```

Similarly, for a pair of vector spaces, their tensor product is defined in the namespace `TensorProduct`

```
variable {K V W: Type*} [Ring K] [AddCommGroup V]
[AddCommGroup W] [Module K V] [Module K W]
open scoped TensorProduct
#check V ⊗ W
```

Subspaces, bases

For a vector space `V` on `K`, `Submodule K V` is type of vector subspaces of `V`.

- `Submodule.span K s` is the subspace spanned by `s : Set V`.
- The lattice structure of subspaces is given using the notations $\top, \bot, \sqcap, \sqcup$.
- If `E : Submodule K V`, the quotient space `V ⧸ E` is given instances of `Module K (V ⧸ E)`, `AddCommGroup V ⧸ E`, making it a vector space.
- `Basis ι K V` the type of `K`-bases of `V`. If `B : Basis ι K V`, `i : ι`, then `B i` is a basis element, while `B.repr : V ≃ₗ[K] ι →₀ K` is the coordinate representation of elements of `V` (`ι →₀ K` is the type of functions with finite support).

Topology

A topology can be defined using open sets.

```
class TopologicalSpace (X : Type u) where
  IsOpen :  Set X → Prop
  isOpen_univ :  IsOpen univ
  isOpen_inter :
    ∀ s t, IsOpen s → IsOpen t → IsOpen (s ∩ t)
  isOpen_sUnion :
    ∀ s, (∀ t ∈ s, IsOpen t) → IsOpen (⋃₀ s)
```

- `IsOpen` says that a set is an open set.
- `Set.univ` is open.
- The intersection of two open sets is an open set.
- The union of a family of open sets is an open set (`→ IsOpen ∅`).

This is a `class` depending on X. Whenever there is a declaration `[TopologicalSpace X]`, we can talk about open sets.

Topology

We can define continuity of `f`.

```
variable {X Y : Type} [TopologicalSpace X]
[TopologicalSpace Y]

structure Continuous (f :  X → Y) : Prop where
  isOpen_preimage :  ∀ s, IsOpen s → IsOpen (f ⁻¹' s)
```

We can talk about convergence of sequences or nets, but it is more convenient to use filters...

Filter

Convergence in a topological space can be defined using (neighbourhood) filters. Filters are in fact more general than neighbourhood filters, one can talk about various limits (not necessarily defined using neighbourhood system) in the same way (such as $\lim_{n\to\infty}, \lim_{x\to 0^+}, \lim_{x\to x_0} f(x) = \infty \ldots$)

```
structure Filter (α :  Type*) where
  sets :  Set (Set α)
  univ_sets :  Set.univ ∈ sets
  sets_of_superset {x y} :  x ∈ sets → x ⊂ y → y ∈ sets
  inter_sets {x y} :  x ∈ sets → y ∈ sets → x ∩ y ∈ sets
```

- The set of sets that belong to the filter.
- The set `Set.univ` belongs to any filter.
- If a set belongs to a filter, then its superset belongs to the filter.
- If two sets belong to a filter, then their intersection belongs to the filter.

Filter

Typical examples of filters are the following.
The principal filter (the filter of all sets including a given `s : Set α`):

```
def principal {α : Type*} (s : Set α) : Filter α where
  sets := { t | s ⊂ t }
  univ_sets := sorry
  sets_of_superset := sorry
  inter_sets := sorry
```

The neighbourhood filter of ∞:

```
def atTop' : Filter ℕ :=
  { sets := { s | ∃ a, ∀ b, a ≤ b → b ∈ s }
    univ_sets := sorry
    sets_of_superset := sorry
    inter_sets := sorry }
```

Convergence Tendsto

The convergence of a function f can be stated as follows.

```
def Tendsto' {X Y : Type*} (f :  X → Y) (F : Filter X) (G :
Filter Y) :=
    ∀ V ∈ G, f ⁻¹' V ∈ F
```

For example, consider convergence of sequences in $\mathbb{R}$.

```
variable (a :  ℕ → ℝ)
#check Filter.Tendso a atTop (Filter.nhds 0)
```

This means that for any open s : Set $\mathbb{R}$ containing 0, there is a t : Set $\mathbb{R}$ of the form t = { m : N ≤ m} for some N : $\mathbb{N}$ and a '' t ⊂ s.

That is, for a given $\epsilon > 0$, thee is N such that if N ≤ m, then a m < ϵ.

Metric space

A special case of `TopologicalSpace` is given by a `MetricSpace`, which is defined as a pair of the meric `dist` satisfying various properties.
In this case, there is an instance of `TopologicalSpace X` given by the metric.

```
variable {X : Type} [MetricSpace X]
#synth TopologicalSpace X
#synth T2Space X
```

There is also the definition of `T2Space X` (Hausdorff space) and it follows automatically (there is an instance of `T2Space X` from `MetricSpace X`)

```
variable {X : Type} [MetricSpace X] (f :  X → ℝ)
#check UniformContinuous f
```

`MetricSpace X` gives also uniform structure.

Normed vector space

We can talk about normed vector spaces. We do not need to separately declare `AddCommGroup X, Module ℝ X`, etc. (and we should not, as they would define independent structures).
Here `NormedAddCommGroup X` defines a norm, while `NormedSpace ℝ X` means that `X` is a module and the operations are continuous.
A Banach space can be declared as follows. The *Banach open mapping theorem* is available in `mathlib`.

```
variable {X Y : Type}
 [NormedAddCommGroup X] [NormedSpace ℝ X] [CompleteSpace X]
 [NormedAddCommGroup Y] [NormedSpace ℝ Y] [CompleteSpace Y]

example (f :  X →L[ℝ] Y) (h :  Function.Surjective f) :
    IsOpenMap f := ContinuousLinearMap.isOpenMap f h
```

Here, `(f :  X →L[ℝ] Y)` means that `f` is a $\mathbb{R}$-linear continuous map.

Concrete vector spaces

The ℓ^p spaces are defined as Banach spaces as follows, where `ENNReal` is the extended real $[0, \infty]$.

```
import Mathlib
variable (p :  ENNReal) [hp :  Fact (1 ≤ p)]
#synth NormedAddCommGroup (lp (fun (_ :  ℕ) => ℂ) p)
#synth NormedSpace ℂ (lp (fun (_ :  ℕ) => ℂ) p)
#synth CompleteSpace (lp (fun (_ :  ℕ) => ℂ) p)
```

For `p = 2`, it is also a Hilbert space.

```
import Mathlib
#synth InnerProductSpace ℂ (lp (fun (_ :  ℕ) => ℂ) 2)
open scoped InnerProductSpace
variable (x y :  (lp (fun (_ :  ℕ) => ℂ) 2))
#check ⟪x, y⟫_ℂ
```

Practice

- Let us do some exercises in `Mathematics in Lean`.