Introduzione alla formalizzazione della matematica in Lean
Lesson 8/10 More algebraic structures

Yoh Tanimoto

Corso di dottorato, University of Rome "Tor Vergata"

21 Gennaio 2026

References

- We follow Mathematics in Lean, Chapters 8, 9.
- Make sure to install Lean, Git and to `clone` the repository (follow this link) (or use a desktop app).

Morphisms

We have seen how to define monoids, groups and rings. How can we define morphisms between them?
A (monoid, group, ring) homomorphism is a map which preserves the respective algebraic structures.

```
@[ext]
class MonoidHom₁ (M N : Type) [Monoid M] [Monoid N] where
  toFun :  M → N
  map_one :  toFun 1 = 1
  map_mul :  ∀ g g', toFun (g * g') = toFun g * toFun g'
```

`toFun` is the map `M → N`, and the properties `map_one, map_mul` implement the unit-preserving, and product-preserving properties.
`@[ext]` attribute automatically generates theorem that two such monoid homs `f1, f2` are equal if `∀ g, f1 g = f2 g`.

class and instance

In order to state general structures and examples of them,

- `class` is use to declare a general structure (including data and properties). We can prove theorems about that structure.
- `instance` is used to declare a concrete object or even a family of objects that has the structure given by a `class`. Any theorem proved for that class applies to any particular instance.
- `#synth` command finds an instance declaration.
- In a variable declaration such as `[Group G]`, it is actually an implicit variable, say `[gs : Group G]` where `gs` has the structure type `Group G`. In some cases, it is useful to give a name.
- In an instance declaration such as `instance : AddGroup ℤ :=...` actually the name is given automatically (can be found by `#synth`).

Coercion to function

If `f : MonoidHom₁`, then `f.toFun : M → N`.
To avoid writing every time `toFun`, we can define a *coercion*, a natural identification between `f` and `f.toFun`.

```
instance [Monoid G] [Monoid H] :
    CoeFun (MonoidHom₁ G H) (fun _ => G → H) where
  coe := MonoidHom₁.toFun

example [Monoid G] [Monoid H] (f : MonoidHom₁ G H) :
    f 1 = 1 := f.map_one
```

Ring homomorphisms

Similarly we can define ring homomorphisms.

```
@[ext]
structure AddMonoidHom₁ (G H : Type) [AddMonoid G]
[AddMonoid H] where
  toFun :  G → H
  map_zero :  toFun 0 = 0
  map_add :  ∀ g g', toFun (g + g') = toFun g + toFun g'

instance [AddMonoid G] [AddMonoid H] :
    CoeFun (AddMonoidHom₁ G H) (fun _ => G → H) where
  coe := AddMonoidHom₁.toFun

@[ext]
structure RingHom₁ (R S : Type) [Ring R] [Ring S]
    extends MonoidHom₁ R S, AddMonoidHom₁ R S
```

Ring homomorphisms as monoid homomorphisms

Any ring homomorphism is a monoid homomorphism, so we want a natural identification, with all the theorems on monoid homomorphisms valid also for ring homomorphisms.

```
class MonoidHomClass₂ (F : Type) (M N : outParam Type)
[Monoid M] [Monoid N] where
  toFun :  F → M → N
  map_one :  ∀ f :  F, toFun f 1 = 1
  map_mul :  ∀ f g g', toFun f (g * g') = toFun f g * toFun
f g'

instance [Monoid M] [Monoid N] [MonoidHomClass₂ F M N] :
    CoeFun F (fun _ => M → N) where
  coe := MonoidHomClass₂.toFun
```

`outParam` means that Lean should search for `M N` after other parameters.

Ring homomorphisms as monoid homomorphisms

```
instance (M N : Type) [Monoid M] [Monoid N] :
    MonoidHomClass₂ (MonoidHom₁ M N) M N where
  toFun := MonoidHom₁.toFun
  map_one := fun f => f.map_one
  map_mul := fun f => f.map_mul

instance (R S : Type) [Ring R] [Ring S] :
    MonoidHomClass₂ (RingHom₁ R S) R S where
  toFun := fun f => f.toMonoidHom₁.toFun
  map_one := fun f => f.toMonoidHom₁.map_one
  map_mul := fun f => f.toMonoidHom₁.map_mul
```

This applies either `RingHom₁ R S` or `MonoidHom₁` as `F` in `MonoidHomClass₂`.

Sub-objects

We can consider submonoids of a given monoid.

```
@[ext]
structure Submonoid₁ (M : Type) [Monoid M] where
  carrier :  Set M
  mul_mem a b :
    a ∈ carrier → b ∈ carrier → a * b ∈ carrier
  one_mem :  1 ∈ carrier
```

`carrier` is the carrier of a submonoid, `mul_mem` says that the product of two elements of a submonoid belongs to the submonoid, `one_mem` says thatt the unit element belongs to the submonoid.

Sub-objects

To treat submonoid as subsets (including the notations such as $\in$), we can declare a `SetLike` instance.

```
/-- Submonoids in `M` can be seen as sets in `M`.  -/
instance [Monoid M] : SetLike (Submonoid₁ M) M where
  coe := Submonoid₁.carrier
  coe_injective' _ _ := Submonoid₁.ext
```

Sub-objects

Any `N : Submonoid₁ M` becomes automatically a type.
If `x : N`, then `x.val : M` is the corresponding object and `x.property` is the proof of `x.val ∈ N`.

```
example [Monoid M] (N : Submonoid₁ M) (x : N) : (x : M) ∈ N := x.property
```

There is also a coercion `N → M`.

Sub-objects

Any `N : Submonoid₁ M` can actually be equipped with the natural monoid structure.

```
instance SubMonoid₁Monoid [Monoid M] (N : Submonoid₁ M) :
Monoid N where
  mul := fun x y => ⟨x*y, N.mul_mem x.property y.property⟩
  mul_assoc := by
    intro x y z
    ext
    exact (mul_assoc (x :  M) y z)
  one := ⟨1, N.one_mem⟩
  one_mul := fun x => SetCoe.ext (one_mul (x :  M))
  mul_one := fun x => SetCoe.ext (mul_one (x :  M))
```

Sub-objects

The intersection of any S_1 S_2 : Submonoid$_1$ M is a submonoid.

```
instance [Monoid M] : Min (Submonoid₁ M) where
  min := fun S₁ S₂ =>
    { carrier := S₁.carrier ∩ S₂.carrier
      one_mem := ⟨S₁.one_mem, S₂.one_mem⟩
      mul_mem := fun ⟨hx, hx'⟩ ⟨hy, hy'⟩ =>
        ⟨S₁.mul_mem hx hy, S₂.mul_mem hx' hy'⟩}
```

This allows us to write $S_1 \cap S_2$.

Concrete groups

For any pair `X Y : Type`, `mathlib` has `Equiv X Y` of bijective maps `X → Y` and `Equiv.Perm X = Equiv X X`.

`Equiv.Perm X` is given a group structure link

If `X` is equipped with more structures, one can define the group preserving that structure.

- For `G` a group, the automorphism group link
- For `X` a vector space (see the next lecture), the general linear group link
- For `X` a normed vector space, the continuous linear group link

Equivalence relations

Let `[Group G] [H : Subgroup G]`. For a pair `a b : G`, we can consider the equivalence relation `r a b ↔ b⁻¹ * a ∈ H`.
We can identify equivalent elements, giving a `Setoid G`:

```
def leftRel' {G : Type} [Group G] (H : Subgroup G) :
    Setoid G where
  r a b := b⁻¹ * a ∈ H
  iseqv := ⟨by intro a; simp,
    by intro a b h; rw [← Subgroup.inv_mem_iff]; simp
    exact h,
    by
    intro a b c h1 h2
    let h := Subgroup.mul_mem _ h2 h1
    rw [mul_assoc, ← mul_assoc b] at h
    simp at h; exact h⟩
```

Quotient group

When we have a `S : Setoid G`, we can consider the quotient type, given by `Quotient (Subgroup G)`. By declaring `instance : HasQuotient G (Subgroup G) := ⟨fun s => Quotient (leftRel' s)⟩`, we can use the notation `G / H` (note that / is actually an unicode character, typed \quot)

In `mathlib` there is a `class` predicate `Subgroup.Normal H`, stating that `H` is a normal subgroup (of `G`). Using the equivalence relation, one can define the group structure on `G / H`, which is declared as an instance:

```
instance QuotientGroup.Quotient.group {G : Type u_1} [Group G] (N : Subgroup G) [nN : N.Normal] : Group (G / N)
```

Note that `G / N` is **not** defined as the type of coset. Rather, it is the type to which there is the quotient map `Quotient.mk` such that `Quotient.mk a = Quotient.mk b` if and only if `r a b`.

Practice

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