Introduzione alla formalizzazione della matematica in Lean
Lesson 7/10 Hierarchies

Yoh Tanimoto

Corso di dottorato, University of Rome “Tor Vergata”

15 Gennaio 2026

References

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

Hierarchy of structures

We have seen how to define `structure`. How can we define group?

- Group is a type
  - equipped with the product $\cdot$
  - the product is associative
  - there is the unit element
  - for each element g, there is an inverse

We want to express, for example,

- Any group is a semigroup
- For a given group G, its subgroups form a complete lattice
- For any pair of groups G_1, G_2, $G_1 \times G_2$ has a natural group structure
- $\mathbb{R}$ is an (additive) group, and thus, we want thoerems about groups applied to $\mathbb{R}$

We want to make some of these automatic.

class

Lean has a special command `class`. It works like `structure`, but it allows to declare `instance`s.

```
class One₁ (α :  Type) where
/-- The element one -/
  one :  α
```

This carries only a distinguished element `one`.

```
example (α :  Type) [One₁ α] :  α := One₁.one
```

The square blacket `[One₁ α]` means that α carries *some* structure of `One₁`, that is, a distinguished element `one`.
In practice, there is a separate declaration of that structure somewhere.
link

```
import Mathlib
#check @One.one Nat _
```

Notation

For any type carrying the structure `One₁`, we can use the common notation.

```
@[inherit_doc] notation "𝟙" => One₁.one
example {α : Type} [One₁ α] : α := 𝟙
example {α : Type} [One₁ α] : (𝟙 : α) = 𝟙 := rfl
```

In `mathlib`, $\mathbb{N}, \mathbb{Z}, \mathbb{Q}, \mathbb{R}$ have all `One` structure and thus the notation `1` makes sense.

```
import Mathlib
#check 1
#check (1 : ℤ)
#check Real.pi + 1
```

Binary operation

Let us consider a class of types having a binary operation, with the notation.

```
class Dia₁ (α :  Type) where
  dia :  α → α → α
infixl:70 " ◇ " => Dia₁.dia
```

$\texttt{Dia}_1$ assumes no property about `dia`, so mathematically it is a *magma*.

extends

Let us also consider *semigroup* structure, using the same notation $\diamond$, using the command `extends`.

```
class Semigroup₂ (α :  Type) extends Dia₁ α where
/-- Diamond is associative -/
  dia_assoc :  ∀ a b c :  α, a ◇ b ◇ c = a ◇ (b ◇ c)
```

This means that any type carrying the structure $\texttt{Semigroup}_2$ has a binary operation $\texttt{Dia}_1\texttt{.dia}$ and it is associative.

```
variable (α :  Type) [Semigroup₂ α]
#check @Dia₁.dia α _
```

link

(the underscore _ means that we want to use the most recently declared $\texttt{Semigroup}_2$ structure on α)

extends

Let us also consider a combination of structures, using the same notation $\diamond$, using the command `extends`.

```
class DiaOneClass₁ (α :  Type) extends One₁ α, Dia₁ α where
/-- One is a left neutral element for diamond.  -/
  one_dia :  ∀ a :  α, 𝟙 ◇ a = a
/-- One is a right neutral element for diamond -/
  dia_one :  ∀ a :  α, a ◇ 𝟙 = a
```

This means that α has a (left and right) unit $\mathbb{1}$.

```
class Monoid₁ (α :  Type) extends Semigroup₁ α,
DiaOneClass₁ α
```

In this way, `Monoid₁ α` gives a product $\diamond$ which is associative and a unit $\mathbb{1}$. This is a monoid structure.

instance

At this point, we can declare a $\texttt{Monoid}_1$ structure on a concrete type such as $\mathbb{N}$.

```
instance :  Monoid₁ ℕ where
  dia := Nat.mul
  dia_assoc := Nat.mul_assoc
  one := 1
  one_dia := Nat.one_mul
  dia_one := Nat.mul_one

#check 2 ◇ 2
```

#synth

Using the command `#synth`, we can see whether a `class` structure has been declared for a type.

```
import Mathlib
#synth AddGroup ℝ
#synth AddCommGroup (ℕ → ℝ)
open ZeroAtInfty
#synth CompleteSpace C₀(ℝ, ℝ)
#synth LieAlgebra ℂ (Matrix (Fin 3) (Fin 3) ℂ)
```

The new structure with `extends` becomes an instance of the old structure.

```
class Monoid₁ (α :  Type) extends Semigroup₁ α,
DiaOneClass₁ α
variable (X : Type) [Monoid₁ X]
#synth Semigroup₁ X
```

More on extends

Note that this automatic generation of instances has a side effect that, if we did not declare the hierarchical structures correctly, there would be some unrelated structures.

```
variable (X : Type) [Monoid₁ X] [Semigroup₁ X]

#check (@Monoid₁.toSemigroup₁ X).toDia₁.dia
#check (@Semigroup₁.toDia₁ X).dia

example :  (@Monoid₁.toSemigroup₁ X).toDia₁.dia =
(@Semigroup₁.toDia₁ X).dia := by
  rfl -- fails
```

To avoid this, we should use `extends` where possible and declare only necessary instances.

Defining group

To define groups, we need the inverse.

```
class Inv₁ (α :  Type) where
/-- The inversion function -/
  inv :  α → α

@[inherit_doc]
postfix:max "⁻¹" => Inv₁.inv

class Group₁ (G : Type) extends Monoid₁ G, Inv₁ G where
  inv_dia :  ∀ a :  G, a⁻¹ ◇ a = 𝟙
```

From these properties, we can derive all the properties of abstract groups.

Defining ring

To define rings, we need both addition and multiplication. Let us use the `mathlib` notations for them, `Add` α and `Mul` α, so that `+` and `*` are defined on α

```
class AddSemigroup₃ (α :  Type) extends Add α where
/-- Addition is associative -/
  add_assoc₃ :  ∀ a b c :  α, a + b + c = a + (b + c)

@[to_additive AddSemigroup₃]
class Semigroup₃ (α :  Type) extends Mul α where
/-- Multiplication is associative -/
  mul_assoc₃ :  ∀ a b c :  α, a * b * c = a * (b * c)
```

`@[to_additive]` attribute copies the definition substituting $*$ by $+$, `mul` in theorem names by `add`, etc.

Defining ring

Let us use the `mathlib` notations for $0, 1$, `AddZeroClass` α and `MulOneClass` α, so that `0` and `1` are defined on α

```
class AddMonoid₃ (α :  Type) extends AddSemigroup₃ α,
AddZeroClass α

@[to_additive AddMonoid₃]
class Monoid₃ (α :  Type) extends Semigroup₃ α, MulOneClass
α
```

Now, with `variable [Monoid₃ α] [AddMonoid₃ α]`, α is equipped with both $*, +$ satisfying the monoid properties.

Defining ring

In a ring, the distributional low for $*, +$ should hold.

```
class Ring₃ (R : Type) extends AddGroup₃ R, Monoid₃ R,
MulZeroClass R where
/-- Multiplication is left distributive over addition -/
  left_distrib : ∀ a b c : R, a * (b + c) = a * b + a * c
/-- Multiplication is right distributive over addition -/
  right_distrib : ∀ a b c : R, (a + b) * c = a * c + b *
c
```

From this definition, commutativity of addition *follows*.

Defining module

We can define an action of R on M.

```
class SMul₃ (α : Type) (β : Type) where
  smul : α → β → β

infixr:73 " • " => SMul₃.smul

class Module₁ (R : Type) [Ring₃ R] (M : Type) [AddCommGroup₃
M] extends SMul₃ R M where
  zero_smul : ∀ m : M, (0 : R) • m = 0
  one_smul : ∀ m : M, (1 : R) • m = m
  mul_smul : ∀ (a b : R) (m : M),
    (a * b) • m = a • (b • m)
  add_smul : ∀ (a b : R) (m : M),
    (a + b) • m = a • m + b • m
  smul_add : ∀ (a : R) (m n : M),
    a • (m + n) = a • m + a • n
```

“Diamond”

We can define the action of a ring on itself:

```
instance selfModule (R : Type) [Ring₃ R] : Module₁ R R
where
  smul := fun r s => r*s
  zero_smul := zero_mul
  one_smul := one_mul
  mul_smul := mul_assoc₃
  add_smul := Ring₃.right_distrib
  smul_add := Ring₃.left_distrib
```

“Diamond”

On the other hand, $\mathbb{Z}$ acts on any ring.

```
def nsmul₁ M : Type* [Zero M] [Add M] : ℕ → M → M
  | 0, _ => 0
  | n + 1, a => a + nsmul₁ n a

def zsmul₁ M : Type* [Zero M] [Add M] [Neg M] : ℤ → M → M
  | Int.ofNat n, a => nsmul₁ n a
  | Int.negSucc n, a => -nsmul₁ n.succ a

instance abGrpModule (A : Type) [AddCommGroup₃ A] : Module₁
ℤ A where
  smul := zsmul₁
  zero_smul := sorry
  one_smul := sorry
  mul_smul := sorry
  add_smul := sorry
  smul_add := sorry
```

“Diamond”

When we ask `#synth Module₁ ℤ ℤ`, Lean picks the latter instance.
But in a more complicated situation, Lean may find the instance `selfModule R`, then the action • there is not yet proven to be same as the one in `abGrpModule R`.

This is very inconvenient, so when there are hierarchy of structures, we should make sure that the poorer structure has less data.
In this case in hand, as the action of $\mathbb{N}$ can be defined on any monoid, we should already include `nsmul : ℕ → M → M := nsmul₁` in the definition of `AddMonoid`, and when we make the `AddMonoid` structure on $\mathbb{Z}$, we use the natural identification `Nat.cast : ℕ → ℤ`.

See Section 8.1 of Mathematics in Lean for more details.

Practice

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