Ax-Grothendieck and Lean

Joseph Hua

December 20, 2021

Contents

1 Introduction

2 Model Theory Background

For most definitions and proofs in this section we reference David Marker's book on Model Theory [1]. We introduce the formalisations of the content in `lean` alongside the theory.

2.1 Languages

> **Definition – Language**
> A language (also known as a *signature*) $\mathcal{L} = (\text{functions}, \text{relations})$ consists of
>
> - A sort symbol A, which we will have in the background for intuition.
> - For each natural number n we have $\text{functions}\, n$ - the set of *function symbols* for the language of *arity* n. For some $f \in \text{functions}\, n$ we might write $f : A^n \to A$ to denote f with its arity.
> - For each natural number n we have $\text{relations}\, n$ - the set of *relation symbols* for the language of *arity* n. For some $r \in \text{relations}\, n$ we might write $r \hookrightarrow A^n$ to denote r with its arity.
>
> The `flypitch` project implements the above definition as
>
> ```
> structure Language : Type (u+1) :=
>   (functions : ℕ → Type u)
> ```

```
(relations : ℕ → Type u)
```

This says that `Language` is a mathematical structure (like a group structure, or ring structure) that consists of two pieces of data, a map called `functions` and another called `relations`. Both take a natural number and spit out a *type* (which in `lean` might as well mean *set*) that consists respectively of all the function symbols and relation symbols of arity n.

In more detail: in type theory when we write `a : A` we mean `a` is something of *type* `A`. We can draw an analogy with the set theoretic notion $a \in A$, but types in `lean` have slightly different personalities, which we will gradually introduce. Hence in the above definitions `functions n` and `relations n` are things of type `Type u`. `Type u` is a collection of all types at level `u`, so things of type `Type u` are types. "Types are type `Type u`."

For convenience we single out 0-ary (arity 0) functions and call them *constant* symbols, usually denoting them by $c : A$. We think of these as 'elements' of the sort A and write $c : A$. This is defined in `lean` by

```
def constants (L : Language) : Type u := functions 0
```

This says that `constants` takes in a language L and returns a type. Following the `:=` we have the definition of `constants L`, which is the type `functions 0`.

EXAMPLE. *The language of rings will be used to define the theory of rings, the theory of integral domains, the theory of fields, and so on. In the appendix we give examples:*

- *The language with just a single binary relation can be used to define the theory of partial orders with the interpretation of the relation as $<$, to define the theory of equivalence relations with the interpretation of the relation as $\sim$, and to define the theory* ZFC *with the relation interpreted as $\in$.*
- *The language of categories can be used to define the theory of categories.*

We will only be concerned with the language of rings and will focus our examples around this.

Definition – Language of rings

Let the following be the language of rings:

- The function symbols are the constant symbols $0, 1 : A$, the symbols for addition and multiplication $+, \times : A^2 \to A$ and taking for inverse $- : A \to A$.
- There are no relation symbols.

We can break this definition up into steps in `lean`. We first collect the constant, unary and binary symbols:

```
/-- The constant symbols in RingLanguage -/
inductive ring_consts : Type u
| zero : ring_consts
| one : ring_consts

/-- The unary function symbols in RingLanguage-/
inductive ring_unaries : Type u
| neg : ring_unaries

/-- The binary function symbols in RingLanguage-/
inductive ring_binaries : Type u
| add : ring_binaries
| mul : ring_binaries
```

These are *inductively defined types* - types that are 'freely' generated by their constructors, listed below after each bar '|'. In these above cases they are particularly simple - the only constructors are terms in the type. In the appendix we give more examples of inductive types

- The natural numbers are defined as inductive types
- Lists are defined as inductive types
- The integers can be defined as inductive types

We now collect all the above into a single definition `ring funcs` that takes each natural `n` to the type of `n`-ary function symbols in the language of rings.

```
/-- All function symbols in RingLanguage-/
def ring_funcs : ℕ → Type u
| 0 := ring_consts
| 1 := ring_unaries
| 2 := ring_binaries
| (n + 3) := pempty
```

The type `pempty` is the empty type and is meant to have no terms in it, since we wish to have no function symbols beyond arity 2. Finally we make the language of rings

```
/-- The language of rings -/
def ring_language : Language :=
(Language.mk) (ring_funcs) (λ n, pempty)
```

We use languages to express logical assertions about our structures, such as "any degree two polynomial over my ring has a root". In order to do so we must introduce terms (polynomials in our case), formulas (the assertion itself), structures and models (the ring), and the relation between structures and formulas (that the ring satisfies this assertion).

We want to express "all the combinations of symbols we can make in a language". We can think of multivariable polynomials over the integers as such: the only things we can write down using symbols $0, 1, -, +, *$ and variables are elements of $\mathbb{Z}[x_k]_{k\in\mathbb{N}}$. We formalize this as terms.

Definition – Terms

Let $\mathcal{L} = (\text{functions}, \text{relations})$ be a language. To make a *preterm* in $\mathcal{L}$ with up to n variables we can do one of three things:

| For each natural number $k < n$ we create a symbol x_k, which we call a *variable* in A. Any x_k is a preterm (that is missing nothing).

| If $f : A^l \to A$ is a function symbol then f is a preterm that is missing l inputs.

$$f(?, \cdots, ?)$$

| If t is a preterm that is missing $l + 1$ inputs and s is a preterm that is missing no inputs then we can *apply* t to s, obtaining a preterm that is missing l inputs.

$$t(s, ?, \cdots, ?)$$

We only really want *terms* with up to n variables, which are defined as preterms that are missing nothing.

```
inductive bounded_preterm (n : ℕ) : ℕ → Type u
| x_ : ∀ (k : fin n), bounded_preterm 0
| bd_func : ∀ {l : ℕ} (f : L.functions l), bounded_preterm l
```

```
  | bd_app : ∀ {l : ℕ} (t : bounded_preterm (l + 1)) (s : bounded_preterm 0),
  bounded_preterm l

def bounded_term (n : ℕ) := bounded_preterm L n 0
```

To explain notation

- The second constructor says "for all natural numbers l and function symbols f, `bd_func f` is something in `bounded_preterm l`". This makes sense since `bounded_preterm l` is a type by the first line of code.
- The curley brackets just say "you can leave out this input and `lean` will know what it is".

To give an example of this in action we can write $x_1 * 0$. We first write the individual parts, which are `x_ 1`, `bd_func mul` and `bd_func zero`. Then we apply them to each other

```
bd_app (bd_app (mul) (x_ 1)) zero
```

Naturally, we will introduce nice notation in `lean` to replace all of this.

Remark. There are many terminology clashes between model theory and type theory, since they are closely related. The word "term" in type theory refers to anything on the left of a `:` sign, or anything in a type. Terms in inductively defined types are (as mentioned before) freely generated symbols using the contructors. Analogously terms in a language are freely generated symbols using the symbols from the language.

One can imagine writing down any degree two polynomial over the integers as a term in the language of rings. In fact, we could even make degree two polynomials over any ring (if we had one):

$$x_0 x_3^2 + x_1 x_3 + x_2$$

Here our variable is x_3, and we imaging that the other variables represent elements of our ring.

To express "for *any* polynomial over our ring, it has a root", we might write

$$\forall x_2 x_1 x_0 : A, \exists x_3 : A, x_0 x_3^2 + x_1 x_3 + x_2 = 0$$

Formulas allow us to do this.

Definition – Formulas

Let $\mathcal{L}$ be a language. A *preformula* in $\mathcal{L}$ with (up to) n *free* variables can be built in the following ways:

- $\bot$ is an atomic preformula with n free variables (and missing nothing).
- Given terms t, s with n variables, $t = s$ is a formula with n free variables (missing nothing).
- Any relation symbol $r \hookrightarrow A^l$ is a preformula with n free variables and missing l inputs.

$$r(?, \cdots, ?)$$

- If ϕ is a preformula with n free variables that is missing $l+1$ inputs and t is a term with n variables then we can *apply* ϕ to t, obtaining a preformula that is missing l inputs.

$$\phi(t, ?, \cdots, ?)$$

- If ϕ and ψ are preformulas with n free variables and *nothing missing* then so is $\phi \Rightarrow \psi$.
- If ϕ is a preformula with $n+1$ free variables and *nothing missing* then $\forall x_0, \phi$ is a preformula with n free variables and nothing missing.

We take formulas to be preformulas with nothing missing. Note that we take the de Brujn index convension here. If ϕ were the formula $x_0+x_1 = x_2$ then $\forall\phi$ would be the formula $\forall x_0 : A, x_0+x_1 = x_2$, which is really $\forall x : A, x + x_0 = x_1$, so that all the remaining free variables are shifted down.

We write this in `lean`, and also define sentences as preformulas with 0 variables and nothing missing. Sentences are what we usually come up with when we make assertions.

```
inductive bounded_preformula : ℕ → ℕ → Type u
| bd_falsum {n : ℕ} : bounded_preformula n 0
| bd_equal {n : ℕ} (t₁ t₂ : bounded_term L n) : bounded_preformula n 0
| bd_rel {n l : ℕ} (R : L.relations l) : bounded_preformula n l
| bd_apprel {n l : ℕ} (f : bounded_preformula n (l + 1)) (t : bounded_term L n) :
bounded_preformula n l
| bd_imp {n : ℕ} (f₁ f₂ : bounded_preformula n 0) : bounded_preformula n 0
| bd_all {n : ℕ} (f : bounded_preformula (n+1) 0) : bounded_preformula n 0

def bounded_formula (n : ℕ) := bounded_preformula L n 0
def sentence := bounded_preformula L 0 0
```

Since we are working with classical logic we make everything else we need by use of the excluded middle:

```
/-- ⊥ is for bd_falsum, ≃ for bd_equal, ⟹ for bd_imp, and ∀' for bd_all -/
/-- we will write ∼ for bd_not, ⊓ for bd_and, and infixr ⊔ for bd_or -/
def bd_not {n} (f : bounded_formula L n) : bounded_formula L n := f ⟹ ⊥
def bd_and {n} (f₁ f₂ : bounded_formula L n) : bounded_formula L n := ∼ ( f₁ ⟹ ∼
f₂ )
def bd_or {n} (f₁ f₂ : bounded_formula L n) : bounded_formula L n := ∼ f₁ ⟹ f₂
def bd_biimp {n} (f₁ f₂ : bounded_formula L n) : bounded_formula L n := ( f₁ ⟹ f₂ )
⊓ ( f₂ ⟹ f₁ )
def bd_ex {n} (f : bounded_formula L (n+1)) : bounded_formula L n := ∼ ( ∀' ∼ f ))
```

With just this much set up (and giving nice symbols to all of the above) we can already write down the sentences that describe rings.

```
/-- Assosiativity of addition -/
def add_assoc : sentence ring_signature :=
  ∀' ∀' ∀' ( (x_ 0 + x_ 1) + x_ 2 ≃ x_ 0 + (x_ 1 + x_ 2) )

/-- Identity for addition -/
def add_id : sentence ring_signature := ∀' ( x_ 0 + 0 ≃ x_ 0 )

/-- Inverse for addition -/
def add_inv : sentence ring_signature := ∀' ( - x_ 0 + x_ 0 ≃ 0 )

/-- Commutativity of addition-/
def add_comm : sentence ring_signature := ∀' ∀' ( x_ 0 + x_ 1 ≃ x_ 1 + x_ 0 )

/-- Associativity of multiplication -/
def mul_assoc : sentence ring_signature :=
∀' ∀' ∀' ( (x_ 0 * x_ 1) * x_ 2 ≃ x_ 0 * (x_ 1 * x_ 2) )

/-- Identity of multiplication -/
def mul_id : sentence ring_signature :=  ∀' ( x_ 0 * 1 ≃ x_ 0 )

/-- Commutativity of multiplication -/
def mul_comm : sentence ring_signature := ∀' ∀' ( x_ 0 * x_ 1 ≃ x_ 1 * x_ 0   )
```

```
/-- Distributibity -/
def add_mul : sentence ring_signature :=
  ∀' ∀' ∀' ( (x_ 0 + x_ 1) * x_ 2 ≃ x_ 0 * x_ 2 + x_ 1 * x_ 2 )
```

We can collect all of these into one and call it the *theory of rings*.

> **Definition – Theory**
> Given a language $\mathcal{L}$, a set of sentences in the language is a theory in that language.

> **Definition – The theories of rings and fields**
>
> ```
> def ring_theory : Theory ring_signature :=
> {add_assoc, add_id, add_inv, add_comm, mul_assoc, mul_id, mul_comm, add_mul}
> ```

We intend to apply the statement "for *any* polynomial over our ring, it has a root" to a real, usable, tangible ring. We would like the sort symbol A to be interpreted as the underlying type (set) for the ring and the function symbols to actually become maps from the ring to itself.

> **Definition – Interpretation and structures**
> Given a language $\mathcal{L}$, a *$\mathcal{L}$-structure* `M` interpreting $\mathcal{L}$ consists of the following
>
> - An underlying type `carrier`.
> - Each function symbol $f : A^n \to A$ is interpreted as a function that takes an n-ary tuple in `carrier` to something in `carrier`.
> - Each relation symbol $r \hookrightarrow A^n$ is interpreted as a proposition about n-ary tuples in `carrier`, which can also be viewed as the subset of the set of n-ary tuples satisfying that proposition.
>
> ```
> structure Structure :=
> (carrier : Type u)
> (fun_map : ∀{n}, L.functions n → dvector carrier n → carrier)
> (rel_map : ∀{n}, L.relations n → dvector carrier n → Prop)
> ```

Naturally any ring is a ring structure and any

3 Model Theory of Algebraically Closed Fields

4 The Lefschetz Principle

5 Reducing to Locally Finite Fields

6 Proving of the Locally Finite Case

References

[1] D. Marker. *Model Theory - an Introduction*. Springer.