Ax-Grothendieck and Lean

Joseph Hua

February 9, 2022

Contents

1 Introduction

2 Model Theory Background

For most definitions and proofs in this section we reference David Marker's book on Model Theory [2]. We introduce the formalisations of the content in `lean` alongside the theory, walking through the basics of definitions made in the `flypitch` project [1]. My work is based on a slightly updated (3.33.0) version of the flypitch project, combined with some of the model theory material put in `mathlib` (which was edited for compatibility).

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* of *arity* n for the language. 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* of *arity* n for the language. 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.*
- *The language of simple graphs can be used to define the theory of simple graphs*

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 multi-variable 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.

2.2 Terms and formulas

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 "any degree (up to) two polynomial over our ring 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 (classical first order) *$\mathcal{L}$-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. For example $x = 0$ is not an assersion about rings, but $\forall x : A, x = 0$ is.

```
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 this set up 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 later collect all of these into one set and call it the theory of rings.

2.3 Interpretation of symbols

In the above we set up a symbol treatment of logic. In this subsection we try to make these symbols into tangible mathematical objects.

We intend to apply the statement "any degree two polynomial over our ring 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 – 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)
> ```
>
> The `flypitch` library uses `dvector A n` for n-ary tuples of terms in `A`.
>
> Note that rather comically `Structure` is itself a mathematical structure. This is sensible, since `Structure` is meant to generalize the algebraic (and relational) definitions of mathematical structures such as groups and rings.
>
> Also note that for constant symbols the interpretation has domain empty tuples, i.e. only the term `dvector.nil` as its domain. Hence it is a constant map - a term of the interpreted carrier type.

The structures in a language will become the models of theories. For example $\mathbb{Z}$ is a structure in the language of rings, a model of the theory of rings but not a model of the theory of fields. In the language of binary relations, $\mathbb{N}$ with the usual ordering $\leq$ is a structure that models of the theory of partial orders (with the order relation) but not the theory of equivalence relations (with $\leq$).

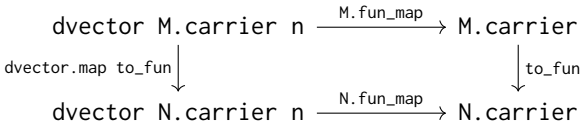
Before continuing on formalizing "any degree two polynomial over our ring has a root", we stop to make the remark that the collection of all structures in a language forms a category. To this end we define morphisms of structures.

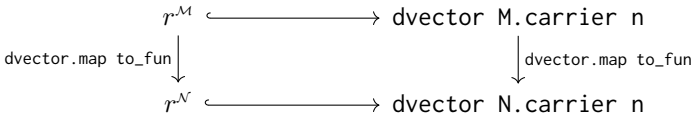
Definition – $\mathcal{L}$-morphism, $\mathcal{L}$-embedding

The collection of all $\mathcal{L}$-structures forms a category with objects as $\mathcal{L}$-structures and morphisms as $\mathcal{L}$-morphisms.

```
protected structure hom :=
(to_fun : M → N)
(map_fun' : ∀{n} (f : L.functions n) x, to_fun (M.fun_map f x)
  = N.fun_map f (dvector.map to_fun x) . obviously)
(map_rel' : ∀{n} (r : L.relations n) x, M.rel_map r x
  → N.rel_map r (dvector.map to_fun x) . obviously)
```

The induced map between the n-ary tuples is called `dvector.map`. The above says a morphism is a mathematical structure consisting of three pieces of data. The first says that we have a functions between the carrier types, the second gives a sensible commutative diagram for functions, and the last gives a sensible commutative diagram for relations[†].





The notion of morphisms here will be the same as that of morphisms in the algebraic setting. For example in the language of rings, preserving interpretation of function symbols says the zero is sent to the zero, one is sent to one, subtraction, multiplication and addition is preserved. In languages that have relation symbols, such as that of simple graphs, preserving relations says that if the relation holds for terms in the domain, then the relation holds for their images.

[†]The way to view relations on a structure categorically is to view it as a subobject of the carrier type.

Returning to our objective, we realize that we need to interpret our degree two polynomial (a term) is something in our ring. The term

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

Should be a map from 4-tuples from the ring to a value in the ring, namely, taking (a, b, c, d) to

$$ac^2 + bc + d$$

We thus need to figure out how terms in the language interact with structures in the language.

Definition – Interpretation of terms

Given $\mathcal{L}$-structure $\mathcal{M}$ and a $\mathcal{L}$-term t with up to n-variables. Then we can naturally interpret (a.k.a realize) t in the $\mathcal{L}$-structure $\mathcal{M}$ as a map from the n-tuples of $\mathcal{M}$ to $\mathcal{M}$ that commutes with the interpretation of function symbols.

```
@[simp] def realize_bounded_term {M : Structure L} {n} (v : dvector M n) :
  ∀{l} (t : bounded_preterm L n l) (xs : dvector M l), M.carrier
| _ (x_ k)         xs := v.nth k.1 k.2
```

```
| _ (bd_func f)    xs := M.fun_map f xs
| _ (bd_app t₁ t₂) xs := realize_bounded_term t₁ (realize_bounded_term t₂ ([])::xs)
```

This is defined by induction on (pre)terms. When the preterm t is a variable x_k, we interpret t as a map that picks out the k-th part of the n-tuple `xs`. This is like projecting to the n-th axis if the structure looks like an affine line. When the term is a function symbol, then we automatically get a map from the definition of structures. In the last case we are applying a preterm t_1 to a term t_2, and by induction we already have interpretation of these two preterms in our structure, so we compose these in the obvious way.

We can finally completely formalize "any (at most) degree two polynomial has a root".

Definition – Interpretation of formulas

Given $\mathcal{L}$-structure $\mathcal{M}$ and a $\mathcal{L}$-formula f with up to n-variables. Then we can interpret (a.k.a realize or satisfy) f in the $\mathcal{L}$-structure $\mathcal{M}$ as a proposition about n terms from the carrier type.

```
@[simp] def realize_bounded_formula {M : Structure L} :
  ∀{n l} (v : dvector M n) (f : bounded_preformula L n l) (xs : dvector M l), Prop
| _ _ v bd_falsum       xs := false
| _ _ v (t₁ ≃ t₂)       xs := realize_bounded_term v t₁ xs = realize_bounded_term v t₂ xs
| _ _ v (bd_rel R)      xs := M.rel_map R xs
| _ _ v (bd_apprel f t) xs := realize_bounded_formula v f (realize_bounded_term v t ([])::xs)
| _ _ v (f₁ ⟹ f₂)       xs := realize_bounded_formula v f₁ xs → realize_bounded_formula v
    f₂ xs
| _ _ v (∀' f)          xs := ∀(x : M), realize_bounded_formula (x::v) f xs
```

This is defined by induction on (pre)formulas.

- $\bot$ is interpreted as the type theoretic proposition `false`.
- $t = s$ is interpreted as type theoretic equality of the interpreted terms.
- Interpretation of relation symbols is part of the data of an $\mathcal{L}$-structure (`rel_map`).
- If f is a preformula with n free variables that is missing $l + 1$ inputs and t is a term with n variables then f applied to t can be interpreted using the interpretation of f and applied to the interpretation of t, both of which are given by induction.
- An implication can be interpreted as a type theoretic implication using the inductively given interpretations on each formula.
- $\forall x_0, f$ can be interpreted as the type theoretic proposition "for each x in the carrier set P", where P is the inductively given interpretation.

We write $\mathcal{M} \models f(a)$ to mean "the realization of f holds in $\mathcal{M}$ for the terms a". We are particularly interested in the case when the formula is a sentence, which we denote as $\mathcal{M} \models f$ (since we need no terms).

```
@[reducible] def realize_sentence (M : Structure L) (f : sentence L) : Prop :=
realize_bounded_formula ([] : dvector M 0) f ([])
```

Now we are able to express "this structure in the language of rings has roots of all degree two polynomials", using interpretation of sentences.

2.4 Theories

Definition – Theory

Given a language $\mathcal{L}$, a set of sentences in the language is a theory in that language.

```
def Theory := set (sentence L)
```

Definition – Models

Given an $\mathcal{L}$-structure $\mathcal{M}$ and $\mathcal{L}$-theory T, we write $\mathcal{M} \models T$ and say $\mathcal{M}$ *is a model of* T when for all sentences $f \in T$ we have $\mathcal{M} \models f$.

```
def all_realize_sentence (M : Structure L) (T : Theory L) := ∀ f, f ∈ T → M ⊨ f
```

A model of the theory of rings should be exactly the data of a ring. Before converting between algebraic objects and their model theoretic counterparts, so we first write down the theories of rings, fields, and algebraically closed fields.

Definition – The theories of rings, fields and algebraically closed fields

The theory of rings is just the set of the sentences describing a ring

```
def ring_theory : Theory ring_signature :=
{add_assoc, add_id, add_inv, add_comm, mul_assoc, mul_id, mul_comm, add_mul}
```

To make the theory of fields we can add two sentences saying that the ring is non-trivial and has multiplicative inverses:

```
def mul_inv : sentence ring_signature :=
∀' (x_ 1 ≃ 0) ⊔ (∃' x_ 1 * x_ 0 ≃ 1)

def non_triv : sentence ring_signature := ∼ (0 ≃ 1)

def field_theory : Theory ring_signature := ring_theory ∪ {mul_inv , non_triv}
```

To make the theory of algebraically closed fields we need to express "every non-constant polynomial has a root". We replace this with the equivalent statement "every monic polynomial has a root". We do this by first making "generic polynomials" in the form of $a_{n+1}x^n + \cdots + a_2x + a_1$, then adding x^{n+1} to it, making it a "generic monic polynomial". The (polynomial) variable x will be represented by the variable `x_ 0`, and the coefficient a_k for each $0 < k$ will be represented by the variable `x_ k`.

We define generic polynomials of degree (at most) n as bounded ring signature terms in $n + 2$ variables by induction on n: when the degree is 0, we just take the constant polynomial x_1 and supply a proof that $1 < 0 + 2$ (we omit these below using underscores). When the degree is $n + 1$, we can take the previous generic polynomial, lift it up from a term in $n + 2$ variables to $n + 3$ variables (this is `lift_succ`), then add $x_{n+2}x_0^{n+1}$ at the front.

```
def gen_poly : Π (n : ℕ), bounded_ring_term (n + 2)
| 0       := x_ ⟨ 1 , _ ⟩
| (n + 1) := (x_ ⟨ n + 2 , _ ⟩) * (npow_rec (n + 1) (x_ ⟨ 0 , _ ⟩))
  + bounded_preterm.lift_succ (gen_poly n)
```

Since the type of terms in the language of rings has notions of addition and multiplication (using the function symbols), we automatically have a way of taking (natural number) powers. This is `npow_rec`.

We proceed to making generic monic polynomials by adding x_0^{n+2} at the front of the generic polynomial.

```
def gen_monic_poly (n : ℕ) : bounded_term ring_signature (n + 2) :=
npow_rec (n + 1) (x_ 0) + gen_poly n

/-- ∀ a₁ ⋯ ∀ aₙ, ∃ x₀, (aₙ x₀^(n-1) + ⋯ + a₂ x₀+ a₁ = 0) -/
def all_gen_monic_poly_has_root (n : ℕ) : sentence ring_signature :=
fol.bd_alls (n + 1) (∃' gen_monic_poly n ≃ 0)
```

We can then easily state "all generic monic polynomials have a root". The order of the variables is important here: the $\exists$ removes the first variable x_0 in the $n+2$ variable formula `gen_monic_poly`$n \simeq 0$, and moves the index of all the variables down by 1, making the remaining expression $\exists$`gen_monic_poly`$n \simeq 0$ a formula in $n+1$ variables. The function `fol.bd_alls n` then adds $n+1$ many "foralls" in front, leaving us a formula with no free variables, i.e. sentence.

```
/-- The theory of algebraically closed fields -/
def ACF : Theory ring_signature := field_theory ∪ (set.range all_gen_monic_poly_has_root)
```

Since `all_gen_monic_poly_has_root` is a function from the naturals, we can take its set theoretic image (called `set.range`), i.e. a sentence for each degree n saying "any monic polynomial of degree n has a root".

Proposition – Algebraic objects $\leftrightarrow$ models

The following are true

- A type A is a ring (according to lean) if and only if A is a structure in the language of rings that models the theory of rings.
- A type A is a field (according to lean) if and only if it is a model of the theory of fields.
- A type A is an algebraically closed field (of characteristic p) if and only if it is a model of $\mathrm{ACF}_{(p)}$.

For the purposes of design in lean it is more sensible to split each "if and only if" into seperate constructions, for converting the algebraic objects into their model theoretic counterparts and vice versa. Although these are very obvious facts on paper, converting between them takes a bit of work in `lean`, especially for the last, where some ground work needs to be done for interpreting `gen_monic_poly`.

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] Flypitch project. https://github.com/flypitch/flypitch.

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