Ax-Grothendieck and Lean

Joseph Hua

May 8, 2022

Contents

1 Introduction

It is a basic fact of linear algebra that any linear map between vector spaces of the same finite dimension is injective if and only if it is surjective. Ax-Grothendieck says that this is partly true for polynomial maps.

Here are some examples of polynomial maps

- Surjective but not injective: $f : \mathbb{C} \to \mathbb{C} := x \mapsto x^2$
- Neither surjective nor injective: $f : \mathbb{C}^2 \to \mathbb{C}^2 := (x, y) \mapsto (x, xy)$
- Bijective: $f : \mathbb{C}^3 \to \mathbb{C}^3 := (x, y, z) \mapsto (x, y, z + xy)$

One might try very hard to look for an example of an injective polynomial map that is not surjective. Replacing $\mathbb{C}$ with an arbitrary field, we notice that surjectivity and injectivity coincide on finite fields. Ax-Grothendieck states that over any algebraically closed field, injectivity of a polynomial map implies surjectivity.

2 Model Theory Background

For most definitions and proofs in this section we reference David Marker's book on Model Theory [3]. 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*) `L = (functions, relations)` consists of

- A sort symbol A, which we will have in the background for intuition.
- For each natural number n we have `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 `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* (in set theory a *set*) that consists respectively of all the function symbols and relation symbols of`functions narity` 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. 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 of 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 preparation for expressing algebraic closure). 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 `L = (functions, relations)` be a language. To make a *preterm* in `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 imagine 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 `L` be a language. A (classical first order) `L`-*preformula* in `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_ex {n} (f : bounded_formula L (n+1)) : bounded_formula L n := ∼ (∀' ∼ f))
```

[†]Or rather, we *will* need excluded middle once we start to interpret these sentences.

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 Lean symbols for ring symbols

We define `bounded_ring_term` and `bounded_ring_formula` for convenience.

```
def bounded_ring_formula (n : ℕ) := bounded_formula ring_signature n
def bounded_ring_term (n : ℕ) := bounded_term ring_signature n
```

We supply instances of `has_zero`, `has_one`, `has_neg`, `has_add` and `has_mul` to each type of bounded ring terms `bounded_ring_terms n`. This way we can use the lean symbols when writing terms and formulas.

```
instance bounded_ring_term_has_zero {n} :
  has_zero (bounded_ring_term n) := ⟨ bd_func ring_consts.zero ⟩

instance bounded_ring_term_has_one {n} :
```

```
  has_one (bounded_ring_term n) := ⟨ bd_func ring_consts.one ⟩

instance bounded_ring_term_has_neg {n} : has_neg (bounded_ring_term n) :=
⟨ bd_app (bd_func ring_unaries.neg) ⟩

instance bounded_ring_term_has_add {n} : has_add (bounded_ring_term n) :=
⟨ λ x, bd_app (bd_app (bd_func ring_binaries.add) x) ⟩

instance bounded_ring_term_has_mul {n} : has_mul (bounded_ring_term n) :=
⟨ λ x, bd_app (bd_app (bd_func ring_binaries.mul) x) ⟩
```

Note that in the above a choice as been made for which side addition and multiplication work on (symbolically), but this won't matter once we interpret into commutative rings.

Since we have multiplication, we also can take terms to powers using `npow_rec`.

```
instance bounded_ring_term_has_pow {n} : has_pow (bounded_ring_term n) ℕ :=
⟨ λ t n, npow_rec n t ⟩
```

2.4 Interpretation of symbols

In the above we set up a symbolic 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 `L`, a `L`-*structure* `M` interpreting `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 to formalize "any degree two polynomial over our ring has a root", we stop to notice that structures in a language form suitable objects for a category.

Definition – `L`-morphism, `L`-embedding

The collection of all `L`-structures forms a category with objects as `L`-structures and morphisms as `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[†].

$$\begin{tikzcd} \texttt{dvector M.carrier n} \arrow[r, "\texttt{M.fun\_map}"] \arrow[d, "\texttt{dvector.map to\_fun}"'] & \texttt{M.carrier} \arrow[d, "\texttt{to\_fun}"] \\ \texttt{dvector N.carrier n} \arrow[r, "\texttt{N.fun\_map}"] & \texttt{N.carrier} \end{tikzcd}$$

$$\begin{tikzcd} r^{\mathcal{M}} \arrow[r, hook] \arrow[d, "\texttt{dvector.map to\_fun}"'] & \texttt{dvector M.carrier n} \arrow[d, "\texttt{dvector.map to\_fun}"] \\ r^{\mathcal{N}} \arrow[r, hook] & \texttt{dvector N.carrier n} \end{tikzcd}$$

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 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

$$ac^2 + bc + d$$

Thus terms should be interpreted as maps $A^n \to A$ for an L-structure A.

Definition – Interpretation of terms

Given `L`-structure `M` and a `L`-term t with up to n-variables. Then we can naturally interpret (a.k.a realize) t in the `L`-structure `M` as a map from the n-tuples of `M` to `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 `L`-structure `M` and a `L`-formula f with up to n-variables. Then we can interpret (a.k.a realize or satisfy) f in the `L`-structure `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 `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 `M` $\models f(a)$ to mean "the realization of f holds in `M` for the terms a". We are particularly interested in the case when the formula is a sentence, which we denote as `M` $\vDash f$ (since we need no terms).

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

2.5 Theories

Now we are able to express statements such as "this structure in the language of rings has roots of all degree two polynomials", using interpretation of sentences. A sensible task is to organize algebraic data, such as rings, fields, and algebraically closed fields, in terms of the sentences that axiomatize them. We call these theories.

Note that this is *the whole point of model theory*, we will be able to work with terms, formulas, structures and theories tangibly, as terms in types (or set theoretically as elements of some sets). This allows us to reason about logic itself, and its interaction with the real world.

> **Definition – Theory**
>
> Given a language `L`, a set of sentences in the language is a theory in that language.
>
> ```
> def Theory := set (sentence L)
> ```

> **Definition – Models, consistent**
>
> Given an `L`-structure `M` and `L`-theory `T`, we write `M ⊨ T` and say `M` *is a model of* `T` when for all sentences $f \in$ `T` we have `M ⊨` f.
>
> ```
> def all_realize_sentence (M : Structure L) (T : Theory L) := ∀ f, f ∈ T → M ⊨ f
> ```
>
> We say an `L`-theory is consistent if it has a model.

A model of the theory of rings should be exactly the data of a ring. To show this we must 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

$$\exists \texttt{gen\_monic\_poly n} \simeq 0$$

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".

Lastly, we express the characteristic of fields. Suppose $p : \mathbb{N}$ is a prime. If we view p as a term in the language of rings[†], then we can define the theory of algebraically closed fields of characteristic p as `ACF` with the additional sentence $p = 0$.

```
def ACFₚ {p : ℕ} (h : nat.prime p) : Theory ring_signature :=
set.insert (p ≃ 0) ACF
```

To define the theory of algebraically closed fields of characteristic 0, we add a sentence $p + 1 \neq 0$ for each natural p.

```
def plus_one_ne_zero (p : ℕ) : sentence ring_signature :=
¬ (p + 1 ≃ 0)

def ACF₀ : Theory ring_signature := ACF ∪ (set.range plus_one_ne_zero)
```

[†] lean figures this out automatically using `nat.cast`, which found our instances of has_zero, has_one and has_add.

Whilst completeness and soundness for first order logic is about converting between symbolic and semantic deduction, there is another layer of conversion that is often swept under the rug, between the semantics and native mathematics. Before the project began, the part of this project that Kevin and I were most skeptical about was model theory actually producing results that were usable, in the sense of being compatible with `mathlib`. We can formalize this problem:

- Structures in a language are the same thing as our internal way of describing structures - a ring structure is actually a type with instances $0, 1, -, +, \times$.
- Models of theories in a language are the same things as our internal way of describing algebraic objects - a model of the theory of rings is actually a ring in lean.
- A model theoretic statement of Ax-Grothendieck holds if and only if an algebraic statement of Ax-

Grothendieck holds. (For fixed characteristic.) mathlib

We informally use the term "internal completeness and soundness" for this kind of phenomenon (coined by Kenny Lau).

3 Internal completeness and soundness for ring theories

In this section and the next we focus on proving internal completeness and soundness results:

> **Proposition – Internal completeness and soundness**
> 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 $\mathrm{ACF}_{(p)}$.
> - (Details later) Ax-Grothendieck stated model theoretically corresponds to Ax-Grothendieck stated internally.
>
> 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 rather trivial, converting between these facts takes a bit of work in `lean`, especially for case of ACF_p where some ground work needs to be done for interpreting `gen_monic_poly`.

Before we embark on a proof, we list some general facts and tips about working with models:

- Proofs are easier when working in models, so our proofs tend to first translate everything we can to the ring, then prove the property there, making use of existing lemmas in the library for rings.
- An important instance of the above phenomenon is the lack of algebraic structure for `bounded_ring_terms`. For example, addition for polynomials written as terms is *not commutative* until it is interpreted into a structure satisfying commutativity, even though it is true in a polynomial ring.
- Sometimes there is extra definitional rewriting that needs to happen, and `dsimp` (or something similar) is needed alongside `simp`.

3.1 Ring Structures

We first make the very obvious observation that given the `lean` instances of `[has_zero]` and `[has_one]` in some type `A`, we can make interpretations of the symbols `ring_consts.zero` and `ring_consts.one`. Similarly for the other symbols:

```
def const_map [has_zero A] [has_one A] : ring_consts → dvector A 0 → A
| ring_consts.zero _ := 0
| ring_consts.one  _ := 1

def unaries_map [has_neg A] : ring_unaries → (dvector A 1) → A
| ring_unaries.neg a := - (dvector.last a)

-- Induction on both ring_binaries and dvector
def binaries_map [has_add A] [has_mul A] : ring_binaries → (dvector A 2) → A
| ring_binaries.add (a :: b) := a + dvector.last b
```

```
| ring_binaries.mul (a :: b) := a * dvector.last b

def func_map [has_zero A] [has_one A] [has_neg A] [has_add A] [has_mul A] :
  Π (n : ℕ), (ring_funcs n) → (dvector A n) → A
| 0       := const_map
| 1       := unaries_map
| 2       := binaries_map
| (n + 3) := pempty.elim
```

This allows us to make any type with such instances a ring structure:

```
def Structure : Structure ring_signature :=
Structure.mk A func_map (λ n, pempty.elim)
```

Conversely given any ring structure, we can easily pick out the above instances. For example

```
def add {M : Structure ring_signature} (a b : M.carrier) : M.carrier := @Structure.fun_map _ M 2
    ring_binaries.add ([a , b])

instance : has_add M := ⟨ add ⟩
```

3.2 Rings

If A is a ring, then surely it is a model of the theory of rings. I have supplied `simp` with enough lemmas to reduce the definitions until requiring the corresponding property about rings, and I have chosen the sentences to replicate the format of each property from `mathlib`. For example `add_comm` below is the internal property for the type `A` (it is not visible to `simp`), and it looks exactly like the statement `M ⊨ add_comm`.

```
variables (A : Type*) [comm_ring A]

lemma realize_ring_theory :
  (struc_to_ring_struc.Structure A) ⊨ ring_signature.ring_theory :=
begin
  intros ϕ h,
  repeat {cases h},
  { intros a b c, simp [add_assoc] },
  { intro a, simp }, -- add_zero
  { intro a, simp }, -- add_left_neg
  { intros a b, simp [add_comm] },
  { intros a b c, simp [mul_assoc] },
  { intro a, simp [mul_one] },
  { intros a b, simp [mul_comm] },
  { intros a b c, simp [add_mul] }
end
```

Conversely, given a model of the theory of rings we can supply an instance of a ring to the carrier type. I supply a lemma for each piece of data going into a `comm_ring`. As an example, we look at `add_comm`.

```
/- First show that add_comm is in ring_theory -/
lemma add_comm_in_ring_theory : add_comm ∈ ring_theory :=
begin apply_rules [set.mem_insert, set.mem_insert_of_mem] end
```

Since `ring_theory` was just built as `{_,_,...,_}` (syntax sugar for insert, insert, ..., singleton), it suffices just to iteratively try a couple of lemmas for membership of such a construction.

```
lemma add_comm (a b : M) (h : M ⊨ ring_signature.ring_theory) : a + b = b + a :=
begin
  /- M ⊨ ring_theory -> M ⊨ add_comm -/
```

```
  have hId : M ⊨ ring_signature.add_comm := h ring_signature.add_comm_in_ring_theory,
  /- M ⊨ add_comm -> add_comm b a -/
  have hab := hId b a,
  simpa [hab]
end
```

There is some definitional and internal simplification happening in here, but like before, for the most part `lean` recognizes that realizing the sentence `add_comm` is the same as having an instance of `add_comm`.

```
def comm_ring (h : M ⊨ ring_signature.ring_theory) : comm_ring M :=
{
  add            := add,
  add_assoc      := add_assoc h,
  zero           := zero,
  zero_add       := zero_add h,
  add_zero       := add_zero h,
  neg            := neg,
  add_left_neg   := left_neg h,
  add_comm       := add_comm h,
  mul            := mul,
  mul_assoc      := mul_assoc h,
  one            := one,
  one_mul        := one_mul h,
  mul_one        := mul_one h,
  left_distrib   := mul_add h,
  right_distrib  := add_mul h,
  mul_comm       := mul_comm h,
}
```

We make use of lean's type class inference system by making the hypothesis of modelling `ring_theory` an instance using `fact`.

```
instance models_ring_theory_to_comm_ring {M : Structure ring_signature}
  [h : fact (M ⊨ ring_signature.ring_theory)] : comm_ring M :=
models_ring_theory_to_comm_ring.comm_ring h.1
```

This way, we can supply an instance that any model of the theory of fields (as a `fact`) is a model of the theory of ring (as a `fact`), and is therefore a commutative ring. We can then extend this commutative ring to a field.

3.3 Fields

Our characterization of fields resembles the structure `is_field` more than the default `field` instance; they are equivalent.

```
structure is_field (R : Type u) [ring R] : Prop :=
(exists_pair_ne : ∃ (x y : R), x ≠ y)
(mul_comm : ∀ (x y : R), x * y = y * x)
(mul_inv_cancel : ∀ {a : R}, a ≠ 0 → ∃ b, a * b = 1)
```

The proof that any field forms a model of the theory of fields is straight forward: since fields are commutative rings, it is a model of `ring_theory` by our previous work; for the other two sentences we exploit `simp` and all the lemmas about fields that already exist in `mathlib`.

```
lemma realize_field_theory :
  Structure K ⊨ field_theory :=
begin
  intros ϕ h,
```

```
  cases h,
  {apply (comm_ring_to_model.realize_ring_theory K h)},
  repeat {cases h},
   { intro,
     simp only [fol.bd_or, models_ring_theory_to_comm_ring.realize_one,
       struc_to_ring_struc.func_map, fin.val_zero', realize_bounded_formula_not,
       struc_to_ring_struc.binaries_map, fin.val_eq_coe, dvector.last,
       realize_bounded_formula_ex, realize_bounded_term_bd_app,
       realize_bounded_formula, realize_bounded_term,
       fin.val_one, dvector.nth, models_ring_theory_to_comm_ring.realize_zero],
     apply is_field.mul_inv_cancel (K_is_field K) },
  { simp [fol.realize_sentence] },
  end
```

Going backwards is even easier. We prove that any model of `field_theory` is a model of `ring_theory` and therefore inherits a `comm_ring` instance. Given this instance of `comm_ring`, it then makes sense to ask for a proof of `is_field M`, which is straightforward:

```
variables {M : Structure ring_signature} [h : fact (M ⊨ field_theory)]

include h

instance ring_model : fact (M ⊨ ring_theory) :=
⟨ all_realize_sentence_of_subset h.1 ring_theory_sub_field_theory ⟩

lemma zero_ne_one : (0 : M) ≠ 1 :=
by { have h1 := h.1, have h2 := h1 non_triv_in_field_theory, simpa [h2] }

lemma mul_inv (a : M) (ha : a ≠ 0) : (∃ (b : M), a * b = 1) :=
by { have h1 := h.1, have hmulinv := h1 mul_inv_in_field_theory, by simpa using hmulinv a ha }

lemma is_field : is_field M :=
{ exists_pair_ne := ⟨ 0 , 1 , zero_ne_one ⟩,
  mul_comm := mul_comm,
  mul_inv_cancel := mul_inv }

noncomputable instance field : field M :=
is_field.to_field M is_field
```

3.4 Algebraically closed fields

Suppose we have an algebraically field `K`. We want to show that it is a model of the theory of algebraically closed fields, which given our work so far amounts to showing that for each natural number n we have that all generic monic polynomials of degree n have a root in `k`. Indeed using `is_alg_closed` we can obtain such a root for any polynomial, but this requires (internally) making a polynomial corresponding `gen_monic_poly n`. We first assume the existence of such a polynomial `P` and that evaluating such a polynomial at some value `x` is the same thing as realising `gen_monic_poly n` at (its coefficients and then) `x`.

```
/-- Algebraically closed fields model the theory ACF-/
lemma realize_ACF : Structure K ⊨ ACF :=
begin
  intros ϕ h,
  cases h,
  /- we have shown that K models field_theory -/
  { apply field_to.realize_field_theory _ h },
  { cases h with n hϕ,
    rw ← hϕ,
```

```
    /- goal is now to show that all generic monic polynomials of degree n have a root -/
    simp only [all_gen_monic_poly_has_root, realize_sentence_bd_alls,
      realize_bounded_formula_ex, realize_bounded_formula,
      models_ring_theory_to_comm_ring.realize_zero],
    intro as,
    have root := is_alg_closed.exists_root
      (polynomial.term_evaluated_at_coeffs as (gen_monic_poly n)) gen_monic_poly_non_const,
      -- the above is our polynomial P and a proof that it is non-constant
    cases root with x hx,
    rw polynomial.eval_term_evaluated_at_coeffs_eq_realize_bounded_term at hx,
    -- the above is the lemma that evaluating P at x is the same as realizing gen_monic_poly n at x
    exact ⟨ x , hx ⟩ },
  end
```

In order to interpret `gen_monic_poly n` as a polynomial, we first note that it is natural to consider n-variable terms in the language of rings as n-variable polynomials over $\mathbb{Z}$:

```
def mv_polynomial.term {n} :
  bounded_ring_term n → mv_polynomial (fin n) ℤ :=
@ring_term_rec n (λ _, mv_polynomial (fin n) ℤ)
  mv_polynomial.X /- variable x_ i -> X i-/
  0 /- zero -/
  1 /- one -/
  (λ _ p, - p) /- neg -/
  (λ _ _ p q, p + q) /- add -/
  (λ _ _ p q, p * q) /- mul -/
```

I designed a handy function called `ring_term_rec` that does "induction on terms in the language of rings", based on `bounded_term.rec` from the `flypitch` project. This says that in order to make a multi-variable polynomial in variables n over $\mathbb{Z}$ (`mv_polynomial (fin n) ℤ`) we can just case on the term. If the term is a variable `x_ i` for some $i < n$ then we interpret that as the polynomial $X_i \in \mathbb{Z}[X_0, \ldots, X_{n-1}]$. The only other way we can get terms is by applying function symbols to other terms, hence we interpret the symbols for zero and one as 0 and 1, the symbolic negation of a term by subtracting the inductively given polynomial for the term in the ring, and so on.

Then we use this to make an general algorithm that takes a term t in the language of rings with up to $n+1$ variables and a list of n coefficients from a ring A, and returns a polynomial in $A[X]$. This is designed to treat he first variable X_0 of the associated polynomial as the polynomial variable X, and use the list (`dvector`) of coefficients $[a_1, \ldots, a_n]$ to evaluate the variables $X_1, \ldots, X_n$.

```
def polynomial.term_evaluated_at_coeffs {n} (as : dvector A n) (t : bounded_ring_term n.succ) :
    polynomial A :=
/- First make a map σ : {0, ..., n} → {X, as.nth' 0, ..., as.nth n} ⊆ A[X] -/
let σ : fin n.succ → polynomial A :=
@fin.cases n (λ _, polynomial A) polynomial.X (λ i, polynomial.C (as.nth' i)) in
/- Then this induces a map mv_polynomial.eval σ : A[X_0, ..., X_n] → A[X] by evaluating coefficients -/
mv_polynomial.eval
  σ
  (mv_polynomial.term t)
/- We evaluate at the multivariable polynomial corresponding to the term t -/
```

It remains to show that this polynomial in $A[X]$ evaluated at some a_0 gives the same value in the ring as the original term, realized at the `dvector` $[a_0, \ldots, a_n]$. This follows from the following two facts:

- A term `t` realized at values $[a_0, \ldots, a_n]$ is equal to the polynomial `mv_polynomial.term t` evaluated at the values $[a_0, \ldots, a_n]$. I called this `realized_term_is_evaluated_poly` and has a quick proof using `ring_term_rec`.
- If a multi-variable polynomial is evaluated at $(X, a_1 \ldots, a_n)$ in $A[X]$, then the resulting polynomial is

evaluated at a_0, then this is the same as simply evaluating the multi-variable polynomial at $(a_0, \ldots, a_n)$. This has a rather uninteresting proof, which I called `mv_polynomial.eval_eq_poly_eval_mv_coeffs`.

Moving on to the converse, we assume we have a model `M` of the theory of algebraically closed fields, and a non-constant polynomial p with coefficients in the model (as a field, by our previous work). We want to show that p has a root.

```
variables {M : Structure ring_signature} [hM : fact (M ⊨ ACF)]

instance is_alg_closed : is_alg_closed M :=
begin
  apply is_alg_closed.of_exists_root_nat_degree,
  intros p hmonic hirr hdeg,
  sorry,
end
```

We can feed the coefficients of p to our model theoretic hypothesis, which will give us a root to `gen_monic_poly` realized at these coefficients, which I call `root`.

```
instance is_alg_closed : is_alg_closed M :=
begin
  apply is_alg_closed.of_exists_root_nat_degree,
  intros p hmonic hirr hdeg,
  simp only [...] at hM,
  obtain ⟨ _ , halg_closed ⟩ := hM.1,
  set n := polynomial.nat_degree p - 1 with hn,
  /- I call the coefficients xs -/
  set xs := dvector.of_fn (λ (i : fin (n + 1)), polynomial.coeff p i) with hxs
  obtain ⟨ root , hroot ⟩ := halg_closed n xs,
  use root, /- root should be the root of p -/
  convert hroot,
  sorry,
end
```

It suffices to show that `root` is the root of p. Given the hypotheses, this amounts to equating the (internal) algebraic goal and the model theoretic hypothesis `hroot` about `root`.

```
/- The goal (at 'convert hroot') -/
polynomial.eval root p = realize_bounded_term (root::xs) (gen_monic_poly n) dvector.nil
```

In order to do this we *could* use try to reconstruct p using our previous construction `polynomial.term_evaluated_at_coeffs`. However, unfortunately I have discovered that generally it can be more straightforward to simply develop each side of the argument (interanal completeness and soundness) seperately. I make use of a result in the library that writes a polynomial evaluated at a root as a sum indexed by its degree:

```
lemma eval_eq_finset_sum (p : R[X]) (x : R) :
  p.eval x = Σ i in range (p.nat_degree + 1), p.coeff i * x ^ i :=
/- See mathlib. -/
```

Then we can directly compare this to `gen_monic_poly` realized at the values `xs` and `root`. After providing `simp` with the appropriate lemmas (such as the assumption that `p` is monic), the goal reduces to

```
root ^ p.nat_degree + (finset.range p.nat_degree).sum (λ (x : ℕ), p.coeff x * root ^ x) =
    root ^ p.nat_degree + realize_bounded_term (root::dvector.of_fn (λ (i : fin (n + 1)), p.coeff ↑i))
     (gen_poly n) dvector.nil
```

The first monomial pops out on both sides, allowing us to cancel them with `congr`. It remains to find out how `gen_poly n` is realised. We extract this as a lemma, which we prove by induction on n, since `gen_poly` was built inductively. Each part is just a long `simp` proof which can be found in the source code.

```
lemma realize_gen_poly {n : ℕ} {root} {c : ℕ → M} :
realize_bounded_term
  (dvector.cons root (dvector.of_fn (λ (i : fin (n + 1)), c i)))
  (gen_poly n) dvector.nil =
(finset.range (n + 1)).sum (λ (x : ℕ), c x * root ^ x) :=
begin
  induction n with n hn,
  { simp ... },
  { simp ... }
end
```

3.5 Characteristic

We omit the details of similar proofs for characteristic as it is not as interesting as the other parts. Here are the lemmas we prove along the way, some of which are convenient to feed to `lean` as `instances`

```
instance models_ACF_p_to_models_ACF [hp : fact (nat.prime p)] [hM : fact (M ⊨ ACF_p hp.1)] : fact (M ⊨
    ACF) := sorry

instance models_ACF_0_to_models_ACF [hM : fact (M ⊨ ACF_0)] : fact (M ⊨ ACF) := sorry

lemma models_ACF_p_char_p [hp : fact (nat.prime p)] [hM : fact (M ⊨ ACF_p hp.1)] : char_p M p := sorry

lemma models_ACF_0_char_zero' [hM : fact (M ⊨ ACF_0)] : char_zero M := sorry
```

4 Internal completeness and soundness for Ax-Grothendieck

In this section we will introduce both Ax-Grothendieck and its model theoretic counterpart. We then investigate internal completeness and soundness for these statements.

> **Definition – Polynomial maps**
>
> Let K be a commutative ring and n a natural (we use K since we are only interested in the case when it is an algebraically closed field). Let $f : K^n \to K^n$ such that for each $a \in K^n$,
>
> $$f(a) = (f_1(a), \ldots, f_n(a))$$
>
> for $f_1, \ldots, f_n \in K[x_1, \ldots, x_n]$. Then we call f a polynomial map over K.
>
> For the sake of computation it is simpler to simply assert the data of the n polynomials directly:
>
> ```
> def poly_map (K : Type*) [comm_semiring K] (n : ℕ) : Type* :=
> fin n → mv_polynomial (fin n) K
> ```
>
> Then we can take the map on types/sets by evaluating each polynomial
>
> ```
> def eval : poly_map K n → (fin n → K) → (fin n → K) :=
> λ ps as k, mv_polynomial.eval as (ps k)
> ```

> **Proposition – Ax-Grothendieck**
>
> Any injective polynomial map over an algebraically closed field is surjective. In particular injective polynomial maps over $\mathbb{C}$ are surjective.
>
> ```
> theorem Ax_Groth {n : ℕ} {ps : poly_map K n} (hinj : function.injective (poly_map.eval ps))
> ```

```
    :
  function.surjective (poly_map.eval ps) := sorry
```

The key lemma to prove this is the Lefschetz principle, which says that ring theoretic statements are true in instances of algebraically closed fields if and only if they are true in all algebraically closed fields (assuming zero or large enough prime characteristic). Lefschetz will be stated and proven in a later section.

An overview of the proof of Ax-Grothendieck follows:

- We want to reduce the statement of Ax-Grothendieck to a model-theoretic one. Then we can apply the Lefschetz principle to reduce to the prime characteristic case.
- To express "for any polynomial map ..." model-theoretically, which amounts to somehow quantifying over all polynomials in n variables, we bound the degrees of all the polynomials, i.e. asking instead "for any polynomial map consisting of polynomials with degree at most d". Then we can write the polynomial as a sum of its monomials, with the coefficients as bounded variables.
- We express injectivity and surjectivity model-theoretically, and prove internal completeness and soundness for these statements.
- We apply Lefschetz, so that it suffices to prove Ax-Grothendieck for algebraic closures of a finite fields. This is quite straight forward.

4.1 Writing down polynomials

Our objective is to state Ax-Grothendieck model-theoretically. Let us assume we have an n-variable polynomial $p \in K[x_1, \ldots, x_n]$. We know that p can be written as a sum of its monomials, and the set of monomials `monom_deg_le n d` is finite, depending on the degree d of the polynomial p. It can be indexed by

$$\texttt{monom\_deg\_le\_finset n d} := \left\{ f : \texttt{fin n} \to \mathbb{N} \mid \sum_{i<n} f i \leq d \right\}$$

Then we write

$$p = \sum_{f \;:\; \texttt{monom\_deg\_le n d}} p_f \prod_{i<n} x^{fi}$$

The typical approach to writing a sum like this in lean would be to tell lean that only finitely many of the p_f are non-zero ($p_\star$ is finitely supported - `finsupp`). However, the API built for this assumes that the underlying type in which the sum takes place is a commutative monoid, which is not the case here, as we will be expressing the above as a sum of terms in the language of rings. This type has addition and multiplication and so on, which we supplied as instances already, but these are neither commutative nor associative. Thus the workaround here was to use `list.sumr` (my own definition, similar to `list.sum`) instead, which will take a list of terms in the language of rings, and sum them together.

The below definition is meant to (re)construct polynomials as described above, using free variables to represent the coefficients of some polynomial. This can then be used to express injectivity and surjectivity.

```
def poly_indexed_by_monoms (n d s p c : ℕ)
  (hndsc : (monom_deg_le n d).length + s ≤ c)
  (hnpc : n + p ≤ c) :
  bounded_ring_term c :=
list.sumr
(list.map
  (
    λ f : (fin n → ℕ),
```

```
    let
      x_js : bounded_ring_term c :=
      x_ ⟨((monom_deg_le n d).index_of' f + s) , ... ⟩,
      x_ip (i : fin n) : bounded_ring_term c :=
      x_ ⟨ (i : ℕ) + p , ... ⟩
    in
    x_js * (n.non_comm_prod (λ i, npow_rec (f i) (x_ip i)))
  )
  (monom_deg_le n d)
)
```

To explain the above, we wish to express the ring term with c many free variables ("in context c")

$$\sum_{f \in \texttt{monom\_deg\_le n d}} x_{j+s} \prod_{0 \leq i < n} x_{i+p}^{f(i)}$$

- When we write `x_ < n , ... >` we are giving a natural n and a proof that n is less than the context/-variable bound c, which we omit here.
- `list.map` takes the list `monom_deg_le n d` (which is just `monom_deg_le_finset n d` as a list instead[1]) and gives us a list of terms looking like

$$x_{js} \prod_{i<n} (x_{ip} i)^{fi}$$

  one for each $f \in$ `monom_deg_le_finset n d`.
- Then `list.sumr` sums these terms together, producing a term in c many free variables representing a polynomial.
- To define `x_js` we take the index of f in the list that f came from and we add s at the end and take the variable $x_{\texttt{index f}+s}$.
- To make the product we use `non_comm_prod` (this makes products indexed by `fin n`, and works without commutativity or associativity conditions). For each $i < n$ we multiply together x_{i+p}.
- The purpose of adding s and p is to ensure we are not repeating variables in this expression. They give us control of where the variables begin and end.

  In the two situations where these polynomials are used p is taken to be either 0 or n; this makes the realizing variables $x_0, \ldots, x_{n-1}$ or $x_n, \ldots, x_{2n-1}$ represent evaluating the polynomials at values assigned to $x_0, \ldots, x_{2n-1}$.

  The value s in both instances taken to be $j \times |\texttt{monom\_deg\_le\_finset n d}| + 2n$, where j will represent the j-th polynomial (out of the n polynomials from `poly_map_data`). This ensures that the variables between different polynomials in our polynomial map don't overlap.

4.2 Injectivity and surjectivity

We can then express injectivity of a polynomial map.

```
def inj_formula (n d : ℕ) :
  bounded_ring_formula (n * (monom_deg_le n d).length) :=
let monom := (monom_deg_le n d).length in
-- for all pairs in the domain x_ ∈ K^n and ...
bd_alls' n _
$
```

[1] This uses the axiom of choice, in the form of `finset.to_list`.

```
-- ... y_ ∈ Kⁿ
bd_alls' n _
$
-- if at each pⱼ
(bd_big_and n
-- pⱼ x_ = pⱼ y_
  (λ j,
    (poly_indexed_by_monoms n d (j * monom + n + n) n _ _ ) -- note n
    ≃
    (poly_indexed_by_monoms n d (j * monom + n + n) 0 _ _ ) -- note 0
  )
)
-- then
⟹
-- at each 0 ≤ i < n,
(bd_big_and n ( λ i,
-- xᵢ = yᵢ (where yᵢ is written as x_{i+n+1})
  x_ ⟨ i + n , ... ⟩ ≃ x_ (⟨ i , ... ⟩)
))
```

To explain the above, suppose we have p the data of a polynomial map (i.e. for each $j < n$ we have p_j a polynomial). We wish to express "for all $x, y \in K^n$, if $px = py$ then $x = y$".

- `bd_alls' n` adds n many $\forall$s in front of the formula coming after. The first represents $x = (x_n, \ldots, x_{2n-1})$ and the second represents $y = (y_0, \ldots, y_{n-1}) = (x_0, \ldots, x_{n-1})$. We choose this ordering since when we quantify this expression we first introduce x, which is of a higher index.
- `bd_big_and n` takes n many formulas and places $\wedge$s between each of them. In particular it expresses $px = py$, by breaking this up into the data of "for each $j < n$, $p_j x = p_j y$", as well as $x = y$, by breaking this up into the data of "for each $i < n$, $x_{i+n} = x_i$"
- To write $p_j x$ and $p_j y$ we simply find the right variable indices to supply `poly_indexed_by_monoms`, and we ask for them to be equal.

Surjectivity is similar

```
def surj_formula (n d : ℕ) :
  bounded_ring_formula (n * (monom_deg_le n d).length) :=
let monom := (monom_deg_le n d).length in
-- for all x_ ∈ Kⁿ in the codomain
bd_alls' n _
$
-- there exists y_ ∈ Kⁿ in the domain such that
bd_exs' n _
$
-- at each 0 ≤ j < n
bd_big_and n
-- pⱼ y_ = xⱼ
λ j,
  poly_indexed_by_monoms n d (j * monom + n + n) 0 _
    inj_formula_aux0 inj_formula_aux1
  ≃
  x_ ⟨ j + n , ... ⟩
```

We wish to express "for all $x \in K^n$, there exists $y \in K^n$ such that $py = x$". Just like `bd_alls' n`, `bd_exs' n` adds n many $\exists$s in front of the formula coming after.

Now we are ready to express Ax-Grothendieck model theoretically and state internal soundness and completeness.

```
theorem realize_Ax_Groth_formula {n : ℕ} :
```

```
  (∀ d : ℕ, Structure K ⊨ Ax_Groth_formula n d)
  ↔
  (∀ (ps : poly_map K n),
    function.injective (poly_map.eval ps) → function.surjective (poly_map.eval ps)) :=
sorry
```

4.3 Completeness and soundness

It is important that the model theoretic statements of the above translate to our statements in lean. We finish by showing ‘`realize_Ax_Groth_formula`‘.

Injectivity (and surjectivity)

We only discuss completeness and soundness for injectivity. The same results for surjectivity are very similar. A closer inspection reveals that we actually need two results:

- (`realize_inj_formula_of_ring`) Given a ring A and a polynomial map, A (as a model of `ring_theory`) realizes the formula `inj_formula` evaluated at the coefficients of a polynomial map if and only if the polynomial map over A is injective.
- (`realize_inj_formula_of_model`) Given a model M of `ring_theory` and a (huge) list (`dvector`) of coefficients (representing n polynomials), M realizes the formula `inj_formula` evaluated at the coefficients of a polynomial map if and only if the polynomial map over M (as a field) is injective.

Clearly these are slightly different lemmas. They are necessary because of the data one has at hand depends on the direction one is working on. In more detail

```
lemma realize_inj_formula_of_ring
  {n d : ℕ}
  (ps : poly_map A n)
  (hdeg : ∀ (i : fin n), (ps i).total_degree ≤ d) :
  @realize_bounded_formula _ (struc_to_ring_struc.Structure A)
    _ _ (@poly_map.coeffs_dvector' A _ n d ps)
    (inj_formula n d) dvector.nil
  ↔
  function.injective (poly_map.eval ps) := sorry
```

The function `poly_map.coeffs_dvector'` takes the polynomial `ps` and finds all the coefficients of each polynomial in `ps` and converts that to a `dvector`. This is the only way we can use the data of `ps` in terms of realization. Conversely all we have access to on the side of ring structures will be a list of the coefficients, so we need

```
lemma realize_inj_formula_of_model
  {n d : ℕ} (coeffs : dvector (struc_to_ring_struc.Structure A)
    (n * (monom_deg_le n d).length)) :
  function.injective
    (poly_map.eval (λ i : fin n, mv_polynomial_of_coeffs (dvector.ith_chunk i coeffs)))
  ↔
  realize_bounded_formula coeffs (inj_formula n d) dvector.nil := sorry
```

Both of the above facts are simple to prove, but tiresome to work out.

Now that we have introduced completeness and soundness for Ax-Grothendieck we can move back and forth between algebra and model theory, so that we can do the following

$$\begin{tikzcd}
\text{(algebraic) locally finite Ax-G} \arrow[d, "\text{(internal) completeness}_p"] & & \\
\text{(model theoretic) locally finite Ax-G} \arrow[d, "\text{ACF}_p \text{ is complete}"'] & & \\
\text{(model theoretic) } \chi_p \text{ Ax-G} \arrow[r, "\text{int. soundness}_p"] \arrow[d, "\text{Lefschetz } \chi\text{-change}"] & \text{(algebraic) } \chi_p \text{ Ax-G} \arrow[dr] & \\
\text{(model theoretic) } \chi_0 \text{ Ax-G} \arrow[r, "\text{int. soundness}_0"'] & \text{(algebraic) } \chi_0 \text{ Ax-G} \arrow[r, "\text{case on } \chi"] & \text{(algebraic) Ax-G}
\end{tikzcd}$$

The reason that we need to split into cases depending on characteristic (the final two arrows above) is because ACF alone is not a complete theory, but adding in contraints on characteristic makes it so.

Hence the following parts remain: The locally finite part of the proof is given in the next section. The fact that ACF_p and ACF_0 are complete theories and the characteristic-change lemma are usually packaged together in the Lefschetz principle, which we work on in the section after that. The rest of the components of the proof are covered by our work so far.

5 The Locally Finite Case

Since Chris Hughes wrote the proof to this part of the project I will only explain the mathematics behind the proof and not talk about the `lean` formalization of it.

> **Definition – Locally finite fields [2]**
> Let K be a field of characteristic p a prime. Then the following are equivalent definitions for K being a *locally finite field*:
>
> 1. The minimal subfield generated by any finite subset of K is finite.
> 2. $\mathbb{F}_p \to K$ is algebraic.
> 3. K embeds into an algebraic closure of $\mathbb{F}_p$.
>
> The important example for us of a locally finite field is an algebraic closure of $\mathbb{F}_p$. By the following theorem, this is a model of ACF_p satisfying Ax-Grothendieck.

Proof. 1. $\Rightarrow$ 2. Let $a \in K$. Then $\mathbb{F}_p(a)$ is the minimal subfield generated by a, and is finite by assumption. Finite field extensions are algebraic a is algebraic over $\mathbb{F}_p$.

2. $\Rightarrow$ 1. We show by induction that K is locally finite. Let S be a finite subset of K. If S is empty then $\mathbb{F}_p(S) = \mathbb{F}_p$ and so it is finite. If $S = T \cup s$ and $\mathbb{F}_p(T)$ is finite, then $s \in K$ is algebraic so by some basic field theory we can take the quotient by the minimal polynomial of s giving

$$\mathbb{F}_p(T)[x]/\min(s, \mathbb{F}_p(T)) \cong \mathbb{F}_p(S)$$

The left hand side is finite because it is a finite dimensional vector space over a finite field. Hence K is locally finite.

2. $\Leftrightarrow$ 3. These are basic properties of algebraic closures. $\square$

Proposition – Ax-Grothendieck for locally finite fields

Let L be a locally finite field. Then any injective polynomial map $f : L^n \to L^n$ is surjective.

Proof. Let $b = (b_1, \ldots, b_n) \in L^n$. Writing $f = (f_1, \ldots, f_n)$ for $f_i \in L[x_1, \ldots, x_n]$ we can find $A \subseteq L$, the set of all the coefficients of all of the f_i when written out in monomials. $A \cup \{b_1, \ldots, b_n\}$ is finite and L is locally finite, so the subfield K generated by it is also finite.

The restriction $f\big|_{K^n}$ is injective and has image inside K^n since each polynomial has coefficients in K and is evaluated at an element of K^n. An injective endomorphism of a finite set is surjective, hence $b \in K^n = f(K^n)$. □

6 The Lefschetz Principle

Returning to model theory of algebraically closed fields. We begin by introducing the notion of a complete theory:

Definition – Complete theories

An L-theory T is *complete* when either of the following equivalent definitions hold:

- T deduces any sentence of its negation

```
def is_complete' (T : Theory L) : Prop :=
∀ (ϕ : sentence L), T ⊨ ϕ ∨ T ⊨ ∼ ϕ
```

- Sentences true in any model are deduced by the theory.

```
def is_complete'' (T : Theory L) : Prop :=
∀ (M : Structure L) (hM : nonempty M) (ϕ : sentence L), M ⊨ T → M ⊨ ϕ → T ⊨ ϕ
```

- All models of T satisfy the same sentences ("are elementarily equivalent").

Note that the definition `is_complete` from the flypitch project is stronger than these conditions, and is useful when constructing theories with nice properties[†]. However in practice there is no reason to throw that many sentences into our language, so we use the versions above.

[†]Personally, I prefer the word maximal consistent theory for their definition `is_complete`

Proof. The statement is

```
lemma is_complete''_iff_is_complete' {T : Theory L} :
  is_complete' T ↔ is_complete'' T := sorry
```

The forward direction involves casing on the hypothesis of $T \vDash \phi$ or $T \vDash \neg\phi$, in the first case we are done, and in the second we get a contradiction by ϕ being both true and false in our model M.

```
{ intros H M hM ϕ hMT hMϕ,
  cases H ϕ with hTϕ hTϕ,
  { exact hTϕ },
  {
    have hbot := hTϕ hM hMT,
    rw realize_sentence_not at hbot,
    exfalso,
    exact hbot hMϕ } },
```

On the other hand we need to case on whether T is consistent or not. When T is consistent we can show T deduces ϕ or its negation by checking in that model, otherwise T should deduce anything.

```
{ intros H ϕ,
  by_cases hM : ∃ M : Structure L, nonempty M ∧ M ⊨ T,
  {
    rcases hM with ⟨ M , hM0 , hMT ⟩,
    by_cases hMϕ : M ⊨ ϕ,
    { left, exact H M hM0 ϕ hMT hMϕ },
    {
      right,
      rw ← realize_sentence_not at hMϕ,
      exact H M hM0 _ hMT hMϕ} },
  { left,
    intros M hM0 hMT,
    exfalso,
    apply hM ⟨ M , hM0 , hMT ⟩} }
```

□

> **Proposition – Lefschetz principle**
> Let ϕ be a sentence in the language of rings. Then the following are equivalent:
>
> 1. Some model of ACF_0 satisfies ϕ. (If you like $\mathbb{C} \models \phi$.)
> 2. $\mathrm{ACF}_0 \models \phi$
> 3. There exists $n \in \mathbb{N}$ such that for any prime p greater than n, $\mathrm{ACF}_p \models \phi$
> 4. There exists $n \in \mathbb{N}$ such that for any prime p greater than n, some model of ACF_p satisfies ϕ.
>
> The first and last equivalences are due to the theories ACF_p being complete for any p (0 or prime).

To prove the above we need the following

- Vaught's test for showing a theory is complete (this does the first and last equivalences and is needed in the middle equivalence)
- The compactness theorem for the middle equivalence.

In this section we will introduce these notions properly and how they are used. Vaught's test will be proven in a later section. The compactness theorem will not be proven (it was part of the flypitch project).

6.1 Vaught's test

Another way of expressing that a theory T is complete is to ask for models of T to satisfy the same sentences (that they are elementarily equivalent). In particular it is known that isomorphic models satisfy the same sentences.

> **Definition – Categoricity**
> Given a language L and a cardinal κ, an L-theory T is called κ-categorical if any two models of T of size κ are isomorphic.
>
> ```
> def categorical (κ : cardinal) (T : Theory L) :=
> ∀ (M N : Structure L) (hM : M ⊨ T) (hN : N ⊨ T), #M = κ → #N = κ → nonempty (M ≃[L] N)
> ```

Vaught's test says that categoricity is a useful condition for showing a theory is complete. Another condition we will need is that there are only infinite models to the theory

```
def only_infinite (T : Theory L) : Prop := ∀ (M : Model T), infinite M.1
```

Proposition – Vaught's Test

Let L be a language and T be a consistent theory in the language L with only infinite models, such that it is κ-categorical for some large enough cardinal κ (see below for details). Then T is a complete theory.

```
lemma is_complete'_of_only_infinite_of_categorical
  [is_algebraic L] {T : Theory L} (M : Structure L) (hM : M ⊨ T)
  (hinf : only_infinite T) {κ : cardinal}
  (hκ : ∀ n, #(L.functions n) ≤ κ) (hωκ : ω ≤ κ) (hcat : categorical κ T) :
  is_complete' T := sorry
```

This may differ slightly to the statement in other sources; the reason for the choice of these (stronger than usual) hypotheses will be discussed in the section dedicated it its proof.

We apply Vaught's test in our case to show that the theory of algebraically closed fields of a fixed characteristic is complete.

Proposition

ACF_0 is complete and for any prime p, ACF_p is complete.

Proof. The two proofs are similar, so we focus on the characteristic 0 case. According to Vaught's test, we first need to show that ACF_0 is consistent, which we can do my simply giving a model: the algebraic closure of $\mathbb{Q}$. (For ACF_p we take the algebraic closure of $\mathbb{F}_p$.) We already have all the tools to make such a model:

- Mathlib has definitions of the rationals `rat` and finite fields `zmod`.
- (I lift them to an arbitrary universe level for generality.)
- Mathlib already has a definition of algebraic closure `algebraic_closure`.
- We showed that any algebraically closed field is a model of ACF and that characteristic n fields are models of ACF_n.

```
def algebraic_closure_of_rat :
  Structure ring_signature :=
Rings.struc_to_ring_struc.Structure algebraic_closure.of_ulift_rat

instance algebraic_closure_of_rat_models_ACF : fact (algebraic_closure_of_rat ⊨ ACF) :=
by {split, classical, apply is_alg_closed_to.realize_ACF }

instance : char_zero algebraic_closure_of_rat := ...

theorem algebraic_closure_of_rat_models_ACF₀ :
  algebraic_closure_of_rat ⊨ ACF₀ :=
models_ACF₀_iff.2 ring_char.eq_zero
```

The next thing to show is that any model of ACF_0 is infinite. This is true since any algebraically closed field is infinite (I give a proof of this in `Rings.ToMathlib.algebraic_closure`; it is just considering the roots of the separable polynomial $x^n - 1$ for each $0 < n$):

```
lemma only_infinite_ACF : only_infinite ACF :=
  by { intro M, haveI : fact (M.1 ⊨ ACF) := ⟨ M.2 ⟩, exact is_alg_closed.infinite }
```

We need categoricity, for a large cardinal. We choose this to be the continuum $\mathfrak{c}$, the cardinality of $\mathbb{C}$. For each natural there are only finitely many function symbols of that arity in the language of rings, and of course $\omega \leq \mathfrak{c}$. It remains to show that any two algebraically closed fields of characteristic 0 of size $\mathfrak{c}$ are isomorphic. This was proven by Chris Hughes, and is now available in `mathlib`.

```
lemma categorical_ACF₀ {κ} (hκ : ω < κ) : fol.categorical κ ACF₀ :=
begin
  intros M N hM hN hMκ hNκ,
  haveI : fact (M ⊨ ACF₀) := ⟨ hM ⟩, haveI : fact (N ⊨ ACF₀) := ⟨ hN ⟩,
  split,
  apply equiv_of_ring_equiv,
  apply classical.choice,
  apply ring_equiv_of_cardinal_eq_of_char_zero, -- credit to Chris Hughes
  repeat { apply_instance },
  repeat { cc },
end
```

Putting the above together we have

```
theorem is_complete'_ACF₀ : is_complete' ACF₀ :=
is_complete'_of_only_infinite_of_categorical
    instances.algebraic_closure_of_rat
    instances.algebraic_closure_of_rat_models_ACF₀ -- algebraic closure of ℚ is a model of ACF₀
    (only_infinite_subset ACF_subset_ACF₀ only_infinite_ACF) -- alg closed fields are infinite
    -- pick the cardinal κ := 𝔠
    card_functions_omega_le_continuum
    omega_le_continuum
    (categorical_ACF₀ omega_lt_continuum)
```

□

6.2 Compactness

6.3 Proving Lefschetz

One way of stating compactness is the idea that proofs are finite. If T is an L-theory and ϕ is an L-sentence then $T \vDash \phi$ implies there is some finite subtheory $\Delta \subseteq T$ such that $\Delta \subseteq \phi$.

> **Definition – Finitely consistent**
> An L-theory T is finitely consistent if

7 Vaught's test

8 Compactness

References

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

[2] Stack exchange - locally finite fields. https://math.stackexchange.com/questions/633473/locally-finite-field.

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