Ax-Grothendieck and Lean

Joseph Hua

June 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 a finite field, we notice that surjectivity and injectivity coincide in this case. Ax-Grothendieck states that over any algebraically closed field, injectivity of a polynomial map implies surjectivity, and the proof we will give is model theoretic, roughly saying "we may reduce to the finite field case".

This project formalizes this proof of Ax-Grothendieck in `lean`[1]. My github repository for this project is accessible at `github.com/Jlh18/ModelTheoryInLean8`.

I work on a fork of the `flypitch` project [1], which built the framework for basic model theory and proved the independence of the continuum hypothesis. I have made many updates to this fork to make it compatible with more modern versions of `mathlib`. My original work all lies in the folder `src/Rings`, even though a lot of the code (such as the proof of Vaught's test) is general model theory.

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].

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 collection of *function symbols* of *arity* n for the language. For some function symbol f of arity 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 collection of *relation symbols* of *arity* n for the language. For some relation symbol r of arity n we might write $r \hookrightarrow A^n$ to denote r with its arity.

[1]As far as I am aware this is the first attempt to formalize Ax-Grothendieck in proof verification software.

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 arity n.

In more detail: in type theory when we write `a:A` we mean `a` is something of *type* `A`. With caution, we can draw an analogy with the set theoretic notion $a \in A$. Note that in the above definitions, the two types `functions n` and `relations n` are things of type `Type u`, where `Type u` is a collection of all types at level `u`. "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 more generality we can make the language of groups and theory of groups, abelian groups, cyclic groups and so on.*

In the appendix we provide the example of the language with just a single binary relation. This can be used to define the theory of graphs by taking edges as the binary relation, to define the theory of partial orders with $<$ as the binary relation, to define the theory of equivalence relations with $\sim$ as the binary relation, and to define the theory ZFC *with $\in$ as the binary relation.*

Another interesting example is fixing a ring A and making the language of modules over A, with which we can define the theory of modules. In the same vein, we can consider the language of actions from a particular monoid M or a group G.

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 *inductive types*[†] - types that are 'freely' generated by their constructors, which are listed after each bar '`|`'. For example, the definition of `ring_consts` reads "the only things of type `ring_consts` are `ring_consts.zero` and `ring_consts.one`".

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. The syntax below is casing (induction) on the naturals,

```
/-- 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)
```

[†]In the appendix we give more examples of inductive types

- The natural numbers are defined as inductive types
- Lists are defined as inductive types

We soon introduce further inductive types such as terms in a language and formulas in a language.

We use languages to express logical assertions, such as "any (at most) 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 (assertions about rings), structures ("pre-rings") and models (ring), and the relation between structures and formulas (whether the ring satisfies this assertion).

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

2.2 Terms and formulas

Definition – Terms

Let `L = (functions, relations)` be a language. To make a (bounded) *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 of type `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`. `bd_func mul` is a preterm missing 2 inputs; applying `mul` to `x_ 1` gives us a preterm `bd_app (bd_func mul) (x_ 1)` missing 1 input. Then applying this to `bd_func zero` gives a term

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

We will introduce nice notation in `lean` to make this more legible.

Remark. There is an unfortunate terminology clash 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; "a term of type `A`". In this document we say "something of type `A`" to avoid confusion.

Terms in inductively defined types are (as mentioned before) freely generated symbols using the contructors. Similarly, 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 are only missing logical connectives:

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

This will be expressed as a formula.

Definition – Formulas

Let L be a language. A (classical first order bounded) L-*preformula* in L with (up to) n *free* variables can be built in the following ways:

| $\perp$ 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 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, but $\forall x : A, x = 0$ *is* an assersion about rings.

```
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.

Remark – Induction principles. Each inductive type is automatically given a mapping out property (or induction principle or recursor) [1], and this is central for any constructions and proofs involving terms and formulas. Preterms and preformulas are quite elaborate inductive types, due to `bd_app`, `bd_apprel` and `bd_all`, and much more complicated than `nat` and `list T`.[2]

The induction principle for terms states that to prove a statement about bounded preterms missing `any number of inputs`, it suffices to prove the statement for the variables, to prove the statement for `bd_func f`, and to prove the statement for `bd_app t s` given proofs for the statement for preterms `t` and `s` respectively.

This induction requires working with preterms missing all l inputs at once. This is because if `t` is a preterm with n variables and $l+1$ missing inputs, then `bd_app t s` is a preterm with n variables and l missing inputs; we needed information from `bounded_preterm n (l + 1)` to deduce information for something of type `bounded_preterm n l`.

The induction principle for formulas is similar, with `bd_falsum` and `bd_equal` behaving like `bd_var`, `bd_rel` behaving like `bd_func`, and `bd_apprel` behaving like `bd_app`. For the rest of the induction, one must prove the statement about `bd_imp f1 f2` given proofs of the statement for preformulas `f1` and `f2`, and prove the statement about `bd_all f` given a proof of the statement for `f`.

This induction requires not only to work with preformulas missing all l inputs at once, but also to work with preformulas in all n variables at once. This is because if `f` is a preformula with $n+1$ variables missing l inputs then then `bd_all f` is a preformula with n variables missing l inputs; we needed information from `bounded_preformula (n + 1) l` to deduce information for something of type `bounded_preformula (n + 1) l`.

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 ⟩
```

[1]Due to lean's type theory having a universe of propositions `Prop`, there is a slight difference between recursion and induction, the former mapping into *types* and the latter mapping into *propositions*. However, for the purposes of understanding they can be identified.

[2]The recursors for the naturals and lists are explained in the appendix. The difference in wording there should reflect that they are recursors rather than induction principles.

```
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 the above fixes which order addition and multiplication work in (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 ⟩
```

We can now write down the sentences that describe rings with ease.

```
  /-- 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.

2.4 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 for the ring and the function symbols to actually become maps from the ring to itself.

Definition – Structures

Given a language L, an 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 of type carrier. Note that the domain of this function for constant symbols is an empty tuples, hence it is a constant map into carrier, or simply a constant of type 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 (L : Language) :=
(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 defines dvector A n to be n-ary tuples of terms in A. This is replaced with fin in the mathlib setup of model theory.

Note that Structure L, like Language, is itself a mathematical structure. This is sensible, since Structure is meant to generalize algebraic and relational mathematical structures such as rings, modules, and graphs.

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 $<$ is a structure that models of the theory of partial orders but not the theory of equivalence relations.

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 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) as something in our ring. The term

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

Should be a map sending 4-tuples in the ring to a single 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 (or 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 recursion 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 k-th axis when we view `M` as 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.

This completes our goal of conveying "any (at most) degree two polynomial in our ring has a root". Whilst terms are interpreted as maps, formulas are interpreted as propositions, asking if the structure `M` satisfies the proposition.

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 ⊨` 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

A ring should be exactly a structure in the language of rings satisfying the sentences that axiomatize rings. In this subsection we organize algebraic definitions 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 – Theories and models

Given a language `L`, a set of sentences in the language is a theory in that language.

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

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.

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

The theory of rings is just the set of the sentences describing a ring. A model of the theory of rings should be exactly the data of a ring. We will show this in the next section.

```
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 internal semantics and the ambient mathematics, which we will informally call "internal completeness and soundness". For example, we need to show that a model of a ring is actually a ring, according to the `mathlib` definition of a ring. Showing that models of structures are equivalent to their "actual" mathematical counterparts and that model-theoretically stated theorems are equivalent to their counterparts is arguably the most important part of the project: it shows that model theory actually produces results that are usable in other areas of maths.

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 $\mathrm{ACF}_{(p)}$.

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 on paper, 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 relevant facts and tips:

- 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 properties in the type `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 recogizes 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* 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 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

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

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}.

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

- 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 $j \times |$`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_ ∈ Kⁿ and ...
bd_alls' n _
$
-- ... 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 ∃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 Internal 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 $b = (b_1, \dots, b_n) \in L^n$. Writing $f = (f_1, \dots, f_n)$ for $f_i \in L[x_1, \dots, x_n]$ we can find $A \subseteq L$, the set of all the coefficients of all of the f_i when written out in monomials. $A \cup \{b_1, \dots, b_n\}$ 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 $b \in K^n = f(K^n)$. $\square$

It is important to note that surjectivity does not imply injectivity for locally finite fields. An example:

$$P = x^2 + x + 1 : \overline{\mathbb{F}}_2 \to \overline{\mathbb{F}}_2$$

is a polynomial map between locally finite fields. This is surjective since the algebraic closure has all roots of all polynomials over it. It is *not injective* since this polynomial is separable in characteristic two (its derivative is a unit).

The reason any imitation of the above proof will fail in this direction is that the restriction of a surjective polynomial map is not in general surjective. Let $a \in \overline{F}_2$ be a root of $x^2 + x + 1$. Then

$$P|_{\mathbb{F}(a)} : \mathbb{F}(a) \to \mathbb{F}(a)$$

takes $0, 1 \mapsto 1$ and $a, a+1 \mapsto 0$. Note that $\mathbb{F}(a)$ is size 4 so we have shown it is neither injective nor surjective.

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 ACF_n is complete (Vaught's test)

We want to show that ACF_n is complete. 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 (recalling the definition of an L-morphism).

```
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. We check that this holds for ACF_n and uncountable cardinals.

Proposition – Categoricity for ACF_n
If two algebraically closed fields have the same *uncountable* cardinality then they are (non-canonically) isomorphic.

```
lemma ring_equiv_of_cardinal_eq_of_char_eq
  {K L : Type u} [hKf : field K] [hLf : field L]
  (hKalg : is_alg_closed K) (hLalg : is_alg_closed L)
  (p : ℕ) [char_p K p] [char_p L p]
  (hKω : ω < #K) (hKL : #K = #L) : nonempty (K ≃+* L) := sorry
```

Hence ACF_n is κ-categorical for any uncountable cardinal κ.

Proof. This is proven by Chris Hughes and is now part of mathlib. We outline the argument:

Let $\mathbb{F}$ be the minimal field in K and L, which is either finite or $\mathbb{Q}$ and is the same field since they are of the same characteristic.

There exist transcendence bases for K and L respectively, which we can call s and t. Since K and L are both uncountable, the transcendence bases must be of the same cardinality as the fields.

$$\#K = \omega + \#s = \#s \quad \text{and} \quad \#L = \omega + \#t = \#t$$

Then t and s biject, hence we have ring isomorphisms

$$K \cong \mathbb{F}(s) \cong \mathbb{F}(t) \cong L$$

Then we can apply this to show categoricity:

```
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, -- the char 0 version of what we showed above
  repeat { apply_instance },
  repeat { cc },
end
```

□

Another condition we will need for Vaught's test is that there are only infinite models to the theory

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

This will hold in our case since algebraically closed fields are infinite. We are now in a position to state Vaught's test.

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 details behind the choice of these (stronger than usual) hypotheses for Vaught's Test and Upwards Löwenheim-Skolem will be discussed in the section dedicated it their proofs.

We apply Vaught's test in our case to show that the theory of algebraically closed fields of a fixed characteristic is complete. However, before we do so we need a field theory lemma.

> **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 a large cardinal for categoricity. We choose this to be the continuum $\mathfrak{c}$, the cardinality of $\mathbb{C}$. It is large enough since for each natural there are only finitely many function symbols of that arity in the language of rings, and of course $\omega \leq \mathfrak{c}$.

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

One way of stating compactness is the idea that proofs are finite.

Proposition – Compactness (in terms of deduction)
If T is an L-theory and f is an L-sentence then T deduces ϕ if and only if there is some finite subtheory of T that deduces f.

```
theorem compactness {L : Language} {T : Theory L} {f : sentence L} :
  T ⊨ f ↔ ∃ fs : finset (sentence L), (↑fs : Theory L) ⊨ (f : sentence L) ∧ ↑fs ⊆ T :=
sorry
```

Confusingly, this can be found in a file called `completeness.lean`. The backwards direction of this is obvious since any model of T automatically is a model of a finite subset.

There is an alternative formulation of compactness which we do not use for Lefschetz, but is important as a tool for showing that a theory is consistent. The reader may choose to come back to it when it is referred to later on.

Proposition – Compactness (in terms of consistency)
If T is an L-theory then T is consistent if and only if each finite subtheory of T is consistent.

```
theorem compactness' {L} {T : Theory L} : is_consistent T ↔
  ∀ fs : finset (sentence L), ↑fs ⊆ T → is_consistent (↑fs : Theory L) :=
```

Often the term "finitely consistent" is used to describe the latter case.

I prove the second statement in `Rings.ToMathlib.completeness.lean` (using a lemma from `flypitch` made for the first). The proof I give below is *not exactly* this proof, since I wish to avoid first order logic syntax ($\vdash$), which is the default layer of definitions used in the `flypitch` library, and just argue using model theory ($\models$). However the essence of the proof is the same.

Proposition
The two formulations of compactness are equivalent.

Proof. ($\Rightarrow$) Clearly if T is consistent with a model $\mathcal{M}$ then $\mathcal{M}$ is also a model of any subtheory of T.

For the converse we prove the contrapositive. Suppose T is inconsistent, then $T \models \bot$, since proving this requires assuming a model of T. The first version of compactness implies there is a finite subset of T that deduces $\bot$. This subset cannot be consistent, as any model will satisfy $\bot$.

($\Leftarrow$) Clearly if a finite subtheory of T deduces a sentence ϕ then any model of T is a model of the subtheory, hence also satisfies ϕ.

For the converse we again prove the contrapositive. *Note that for a theory Δ and a sentence ϕ we have $\Delta \not\models \phi$ if and only if $\Delta \cup \{\neg\phi\}$ is consistent.* We make use of this fact: Suppose all finite subtheories of T do not deduce a sentence ϕ. Then for any finite subtheory $\Delta \subseteq T$, we have $\Delta \not\models \phi$ and so $\Delta \cup \neg\phi$ is consistent. Then $T \cup \{\neg\phi\}$ is a finitely consistent theory, hence is consistent by the second version of compactness. Hence $T \not\models \phi$. □

6.3 Proving Lefschetz

We are now ready to prove the Lefschetz principle. We begin by showing that if ACF_0 deduces a ring sentence ϕ then ACF_p deduces ϕ for large p. We prove it seperately because it will be used in the converse!

```
theorem characteristic_change_left (ϕ : sentence ring_signature) :
ACF₀ ⊨ ϕ → ∃ (n : ℕ), ∀ {p : ℕ} (hp : nat.prime p), n < p → ACFₚ hp ⊨ ϕ := sorry
```

Proof. We apply compactness: if ACF_0 deduces ϕ then we must have a finite subtheory of ACF_0 that deduces ϕ. In particular since ACF_0 consisted of the axioms for ACF plus $p+1 \neq 0$ for each $p \in \mathbb{N}$ we know that only finitely many such formulas are needed to deduce ϕ. Hence our n should be the maximum p such that $p+1 \neq 0$ is in our finite subtheory, plus 1.

```
begin
  rw compactness,
  intro hsatis,
  obtain ⟨ fs , hsatis , hsub ⟩ := hsatis,
  classical,
  obtain ⟨ fsACF , fsrange , hunion, hACF , hrange ⟩ :=
    finset.subset_union_elim hsub,
  set fsnat : finset ℕ := finset.preimage fsrange plus_one_ne_zero
      (set.inj_on_of_injective injective_plus_one_ne_zero _) with hfsnat,
  use fsnat.sup id + 1,
```

In the above `fs` is our finite subtheory, `fsrange` is the part consisting of the formulas $p+1 \neq 0$, and the other part from ACF is called `fsACF`.

Let us then suppose that we have a prime that is larger than this n. By design, it should be that ACF_p deduces all the formulas from our finite subtheory, hence ACF_p should deduce ϕ. Supposing that M is a model of ACF_p, it suffices that M deduces ϕ. Since `fs` = `fsACF` ∪ `fsrange` deduces ϕ it suffices that M deduces `fsrange` (it deduces ACF so it deduces any subset of ACF).

```
  intros p hp hlt M hMx hmodel,
  haveI : fact (M ⊨ ACF) := ⟨ (models_ACFₚ_iff'.mp hmodel).2 ⟩,
  have hchar := (@models_ACFₚ_iff _ _ _inst_1 _).1 hmodel,
  apply hsatis hMx,
  rw [← hunion, finset.coe_union, all_realize_sentence_union],
  split,
  { apply all_realize_sentence_of_subset _ hACF,
    exact all_realize_sentence_of_subset hmodel ACF_subset_ACFₚ},
  {sorry},
```

It suffices to show that for each q in `fsnat` (the list of q such that $q+1 \neq 0$ is in the subtheory `fs`), M satisfies $q+1 \neq 0$. We can then conclude that this is true since M is characteristic p, and $q < p$. □

Now for the whole theorem:

```
theorem characteristic_change (ϕ : sentence ring_signature) :
ACF₀ ⊨ ϕ ↔ (∃ (n : ℕ), ∀ {p : ℕ} (hp : nat.prime p), n < p → ACFₚ hp ⊨ ϕ) := sorry
```

Proof. It remains to prove the converse. We know that ACF_0 is complete, so either $\mathrm{ACF}_0 \vDash \phi$ or $\mathrm{ACF}_0 \vDash \neg\phi$, and it suffices to refute the latter case.

```
begin
  split,
  { apply characteristic_change_left },
  {
    intro hn,
    cases is_complete'_ACF₀ ϕ with hsatis hsatis,
    { exact hsatis },
    { sorry },
```

We can apply the forward direction of Lefschetz, and have that ACF_p deduces $\neg\phi$ for large p. We instantiate these lower bounds, and take any prime p that is larger than their maximum.

```
    { have hm := characteristic_change_left (∼ ϕ) hsatis,
      cases hn with n hn,
      cases hm with m hm,
      obtain ⟨ p , hle , hp ⟩ := nat.exists_infinite_primes (max n m).succ,
```

We take the algebraic closure Ω of $\mathbb{F}_p$ as a model of ACF_p, and since p is suitably large, we have that $\Omega \vDash \phi$ and $\Omega \vDash \neg\phi$, which is a contradiction.

```
      have hnp : n < p :=
        lt_of_lt_of_le (nat.lt_succ_of_le (le_max_left _ _)) hle,
      have hmp : m < p :=
        lt_of_lt_of_le (nat.lt_succ_of_le (le_max_right _ _)) hle,
      have hS := instances.algebraic_closure_of_zmod_models_ACFₚ hp, -- Ω ⊨ ACFₚ
      specialize @hn p hp hnp _ ⟨ 0 ⟩ hS,
      specialize @hm p hp hmp _ ⟨ 0 ⟩ hS,
      simp only [realize_sentence_not] at hm,
      exfalso,
      apply hm hn } },
end
```

□

7 Vaught's test and Upwards Löwenheim-Skolem

In this section we go back to general model theory, with the goal of proving Vaught's test. However, the proof of Vaught's test relies on a (a variant of) the Upwards Löwenheim-Skolem Theorem. It says the following:

Proposition – Upwards Löwenheim-Skolem

Suppose L is an algebraic language and T is an L-theory. If κ is a sufficiently large cardinal and T has an infinite model, then T has a model of size κ.

```
theorem has_sized_model_of_has_infinite_model [is_algebraic L] {T : Theory L} {κ :
    cardinal}
  (hκ : ∀ n, #(L.functions n) ≤ κ) (hωκ : ω ≤ κ) :
  (∃ M : Structure L, nonempty M ∧ M ⊨ T ∧ infinite M) →
  ∃ M : Structure L, nonempty M ∧ M ⊨ T ∧ #M = κ := sorry
```

This is often stated in terms of starting with an L-structure, and extending it to a larger L-structure, hence the word "upward" in the name. This can be done using the above by taking T to be the set of sentences satisfied by the structure.

7.1 Proof of Vaught's Test

We first apply Upwards Löwenheim-Skolem to prove Vaught's test. Recall the statement:

```
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
```

Proof. The proof is by contradiction. Suppose T is not complete; this gives us a formula ϕ such that

$$T \nvDash \phi \quad \text{and} \quad T \nvDash \neg\phi$$

which in turn (after unfolding the definition of $T \nvDash \phi$) gives us two models M and N of T such that

$$M \nvDash \phi \quad \text{and} \quad N \nvDash \neg\phi$$

our aim is to adjust these to two models of T of cardinality κ so that they are isomorphic by categoricity, but satisfy different sentences.

```
begin
  intro ϕ,
  by_contra hbot,
  simp only [not_or_distrib, not_ssatisfied] at hbot,
  obtain ⟨ ⟨ M , hM0 , hM ⟩ , ⟨ N , hN0 , hN ⟩ ⟩ := hbot,
```

We can adjust cardinality using Upwards Löwenheim-Skolem, obtaining models of cardinality κ. This is why we need T to only have infinite models.

```
  obtain ⟨ M' , hM'0 , hM' , hMcard ⟩ := has_sized_model_of_has_infinite_model hκ hωκ
    ⟨
      M , hM0 , hM ,
      hinf ⟨ M , all_realize_sentence_of_subset hM (set.subset_insert _ _) ⟩
    ⟩,
  obtain ⟨ N' , hN'0 , hN' , hNcard ⟩ := has_sized_model_of_has_infinite_model hκ hωκ
    ⟨
      N , hN0 , hN ,
      hinf ⟨ N , all_realize_sentence_of_subset hN (set.subset_insert _ _) ⟩
    ⟩,
```

By categoricity, M and N are isomorphic as L-structures. We supply a proof that isomorphic structures satisfy the same sentences in `Rings.ToMathlib.fol.lean`. It follows from a series of proofs by induction on terms and formulas.

```
  have hiso := hcat M' N'
    (all_realize_sentence_of_subset hM' (set.subset_insert _ _))
    (all_realize_sentence_of_subset hN' (set.subset_insert _ _)) hMcard hNcard,
  rw all_realize_sentence_insert at hM' hN',
  rw Language.equiv.realize_sentence _ (classical.choice hiso) at hN',
  exact hN'.1 hM'.1,
end
```

□

7.2 Upwards Löwenheim-Skolem

Our remaining goal is to prove Upwards Löwenheim-Skolem.

```
theorem has_sized_model_of_has_infinite_model [is_algebraic L] {T : Theory L} {κ : cardinal}
  (hκ : ∀ n, #(L.functions n) ≤ κ) (hωκ : ω ≤ κ) :
  (∃ M : Structure L, nonempty M ∧ M ⊨ T ∧ infinite M) →
  ∃ M : Structure L, nonempty M ∧ M ⊨ T ∧ #M = κ := sorry
```

The idea of the proof is that we want to design a model of the right size by making a language L_2 extending L and an L_2-theory T_2 extending T (extending in the sense that any L_2-model of T_2 reduces down to a L-model of T), such that the design of L_2 and T_2 guarantee that any model of T_2 is large enough. Meanwhile, we design an L_2-model `term_model` of T_2, by taking the type of all the L_2-terms, and quotienting by equality deduced by T_2 (this requires T_2 to be Henkin), guaranteeing that `term_model` is small enough - it will be bounded by the number of terms, and thus by the number of function symbols in the language.

7.2.1 Adding distinct constant symbols

Suppose we have a language L and a consistent theory T that has an infinite model M, as well as an infinite cardinal κ. Our first goal is to make a consistent theory T_κ in a language L_κ such that any model of T_κ is size at least κ.

The language extended to have κ many symbols: For design reasons it is convenient to work generally. We will do the following

1. Define the type of language morphisms `L1` $\to^L$ `L2`
2. Define the sum of two languages `L1.sum L2` and the language morphisms into the sum.
3. Define the language that has constant symbols indexed by a type α, called `of_constants` α. In our situation we will take α to be κ`.out`, which is a type of cardinality κ (by the axiom of choice)
4. Define the theory `distinct_constants` α in the language `of_constants` α that consists of `a` $\neq$ `b` for each pair of distinct terms `a b :` α.
5. Define the induced `L2`-theory from a morphism of languages `L1` $\to^L$ `L2` and an `L1`-theory
6. Make the sum of the languages L (from the above hypotheses) and `of_constants` α. We are interested in taking the union of the induced theory from `T` and the induced theory from `distinct_constants` α. We call this theory `union_add_distinct_constants T` α
7. Show that `union_add_distinct_constants T` α is consistent when T has an infinite model

(1) A morphism of languages consists of a map on function symbols and a map on relation symbols (for each arity).

```
structure Lhom (L1 L2 : Language) :=
(on_function : ∀{n}, L1.functions n → L2.functions n)
(on_relation : ∀{n}, L1.relations n → L2.relations n)
```

We denote this type by `L1` $\to^L$ `L2`.

(2) The sum of two languages takes two languages and makes the disjoint sum of the function symbols and relation symbols of each arity.

```
def sum (L1 L2 : Language) : Language :=
⟨λn, L1.functions n ⊕ L2.functions n, λ n, L1.relations n ⊕ L2.relations n⟩
```

The obvious morphisms into the sum are from the maps into the disjoint sum of types

```
def sum_inl {L L' : Language} : L →ᴸ L.sum L' :=
⟨λn, sum.inl, λ n, sum.inl⟩

def sum_inr {L L' : Language} : L' →ᴸ L.sum L' :=
⟨λn, sum.inr, λ n, sum.inr⟩
```

(3) `of_constants` α sets α as the set of function symbols of arity 0, and makes no other function or relation symbols.

```
def of_constants (α : Type*) : Language :=
{ functions := λ n, match n with | 0 := α | (n+1) := pempty end,
  relations := λ _, pempty }
```

(4) We first make a function from the product $\alpha \times \alpha$ to the set of sentences, that takes (x_1, x_2) and returns the sentence $x_1 \neq x_2$. Then the image of the complement of the diagonal in $\alpha \times \alpha$ is the theory we want.

```
def distinct_constants_aux (x : α × α) : sentence (Language.of_constants α) :=
∼ (bd_const x.fst ≃ bd_const x.snd)

def distinct_constants : Theory (Language.of_constants α) :=
set.image (distinct_constants_aux _) { x : α × α | x.fst ≠ x.snd }
```

(5) The theory induced by a language morphism is constructed by taking image of the induced map from `L1`-sentences to `L2`-sentences. This in turn is a special case of the induced map on bounded formulas, which is made by induction on bounded formulas and bounded terms. We don't go through the details of this; the code is from `flypitch` and can be found in `language_extension.lean`.

(6) The theory we are interested in is the latter of the following

```
def add_distinct_constants : Theory $ L.sum (Language.of_constants α) :=
Theory_induced Lhom.sum_inr $ distinct_constants _

def union_add_distinct_constants (T : Theory L) (α : Type u) :=
(Theory_induced Lhom.sum_inl T : Theory $ L.sum (of_constants α)) ∪ add_distinct_constants α
```

It combines the induced theories from the maps into the sum, starting with the theory T we want to build a κ-sized model for and adding κ many symbols to it.

(7) Suppose T is a theory with an infinite model M and α is a type. We want to show

```
lemma is_consistent_union_add_distinct_constants {T : Theory L} (α : Type u)
  {M : Structure L} (hMinf : infinite M) (hMT : M ⊨ T):
  is_consistent $ union_add_distinct_constants T α
```

Compactness tells us we only need to show that a finite subset of α is consistent. So we can take our original model M and realize finitely many distinct constant symbols from α in M as distinct elements, as M is infinite. To this end let's suppose we have

$$\texttt{Tfin} \cup \texttt{con\_fin} = \texttt{fs}$$

where `Tfin` and `con_fin` are respectively finite subsets of T and `add_distinct_constants`α.

```
  rw compactness',
  intros fs hfsTα,
  rw model_existence,
  obtain ⟨Tfin, con_fin, hfs, hTfin, h_con_fin⟩ := finset.subset_union_elim hfsTα,
```

We need to pick out all the constant symbols that appeared in `con_fin` (the details of which we will not go through). We call the set of these constant symbols α`_fin`, and note that it must be finite, hence has an injection into M (by choice).

```
  set αfin : finset α := constants_appearing_in (of_constants.preimage con_fin)
    with hαfin,
  let on_αfin : αfin ↪ M := classical.choice ((cardinal.le_def αfin M).1
    (le_of_lt $ cardinal.finset_lt_infinite hMinf)),
```

We can extend this to a full realization of M has a structure in the language `L.sum (of_constants` α). Since M is non-empty, we can send every symbol not in α`_fin` to some arbitrary element (by choice). To instantiate the goal with this structure, we make a general function

```
def sum_Structure : Structure (L.sum (of_constants α)) :=
{ carrier := S,
  fun_map := λ n f, sum.cases_on f (λ f, S.fun_map f) $ of_constants.fun_map c,
  rel_map := λ n r, sum.cases_on r (λ r, S.rel_map r) pempty.elim }
```

which takes a map interpreting the extra constant symbols and produces a structure in the extended language.

```
  have hM0 : nonempty M := infinite.nonempty _,
  set c : α → M :=
    λ x, dite (x ∈ αfin) (λ h, on_αfin ⟨x,h⟩) (λ _, classical.choice hM0) with hc,
  refine ⟨ Language.of_constants.sum_Structure c , hM0 , _ ⟩,
```

It remains to show that this structure is a model of the theory `fs`. That it models a the subset of T is tedious to show but clear[1].

```
  rw [← hfs, finset.coe_union, all_realize_sentence_union],
  split,
  { apply all_realize_sentence_of_subset _ hTfin,
    apply Language.of_constants.sum_Structure_Theory_induced hMT },
  { sorry },
```

That it models `con_fin` requires a lot of rewriting but boils down to the fact that the realization of constant symbols was an injection when restricted to the symbols from `con_fin`. We omit the rest of the code.

7.2.2 term_model

Recalling that our goal is to construct a model of a fixed cardinality, we can move on to designing our model. In `flypitch` this construction is called `term_model`. In brief, given an L-theory T, the structure consists of:

- The collection of L-terms with no variables (`closed_term L`) up to T-equality as the carrier type for the structure. Formally this is the quotient by the relation

$$t \sim s \Leftrightarrow T \vDash t = s$$

- To interpret function symbols, we need to take a symbol f of arity n and a `dvector` of closed terms (up to equivalence) and return (the equivalence class of) a closed term. The term we pick is naturally f applied to the previous n closed terms (using `bd_apps`). One must check that this is respects the equivalence relation.

[1]The lemmas used to show that `sum_Structure c` models the induced theory could be generalized to say that if the interpretation maps agree upon restriction to the smaller language then the extended structure will model the induced theory.

- Similarly to interpret relation symbols, we need to take a symbol r of arity n and a `dvector` of closed terms. The term we pick is r applied to the previous n closed terms (using `bd_apps_rel`).

Of course this construction is not always going to give a model of the theory, since there are theories that have no models. However, in suitable conditions we will have that for any L-sentence ϕ

$$T \vDash \phi \quad \text{if and only if} \quad \texttt{term\_model T} \vDash \phi$$

These conditions are

- T is a maximal complete theory (called `is_complete` in `flypitch`).
- The theory T is Henkin, or has the witness property, or L has enough constant symbols with respect to T:

```
def has_enough_constants (T : Theory L) :=
∃(C : Π(f : bounded_formula L 1), L.constants),
  ∀(f : bounded_formula L 1), T ⊨ ∃' f ⟹ f[bd_const (C f)/0]
```

  This says for each formula with one free-variable there is a constant symbol that would witness the existence of a term satisfying the formula in any model.

We demonstrate the role of these conditions in the following. To prove

$$T \vDash \phi \quad \text{if and only if} \quad \texttt{term\_model T} \vDash \phi$$

for all sentences ϕ, we induct on ϕ. For backwards direction on the $\forall$ case we are showing that

$$\texttt{term\_model T} \vDash \forall x, \phi \quad \text{implies} \quad T \vDash \forall x, \phi$$

Assume `term_model` $\vDash \forall x, \phi$. By **maximality** either $T \vDash \forall x, \phi$ or $T \vDash \exists x, \neg\phi$. It suffices to refute the latter case. Suppose $T \vDash \exists x, \neg\phi$. As T is **Henkin**, there is some constant symbol c such that $T \vDash \neg\phi_c$ where ϕ_c is the sentence with x replaced for c. Since T is **consistent** this implies $T \nvDash \phi_c$, and by the induction hypothesis `term_model` $\nvDash \phi_c$. However, c is realized as some `a : term_model`, thus by our assumption `term_model` $\vDash \phi(a)$, and a bit of work shows `term_model` $\vDash \phi_c$, a contradiction.

7.2.3 Henkinization

The above indicates we must extend to a further language and a further theory in the language, such that the extended theory is maximally consistent and Henkin. We start with a consistent theory T_κ (which in our situation is the theory with extra κ constant symbols). The first thing to do is to make it Henkin, ensuring it is still consistent, we call the language we extended to L_H and the new L_H-theory T_H. Secondly, we throw in enough formulas to make it maximally consistent, and call this new theory T_m. All of this is done in `flypitch` and can mostly be found in `henkin.lean`. An overview follows.

The second step can be done in either of the following ways:

- Since the theory T_H is consistent, it has a L_H-model, hence the set T_m of L_H-sentences satisfied by the L_H-model is a L_H-theory extending T_H. It is maximal by the law of the excluded middle: the model M either satisfies a formula or not, hence

$$M \vDash \phi \quad \text{or} \quad M \vDash \neg\phi$$

  T_m is consistent since M is a model.[1]

[1]Note that $T \nvDash \phi$ does not imply $T \vDash \neg\,\phi$ in general. This implication holds if and only if T is complete. Thus we needed to appeal to a model in the above.

- We can use Zorn's lemma: the set of consistent L_H-theories extending T_H is non-empty as T is in the set. Any chain of consistent L_H-theories extending T_H is bounded above by a consistent theory since we can take the union of them, and check consistency using compactness. One can check that this set theoretic maximality corresponds to the definition of a maximal consistent theory. Consistency is given for free by Zorn.

In the `flypitch` project, the setup of first order logic syntax rather than its semantics means the Zorn approach is most natural.

The first step requires recursively defined languages extending one another

$$L_0 \to^L L_1 \to^L \dots$$

and theories in each language such that the induced theories at each level are sub-theories of the next

$$\begin{tikzcd}
\mathtt{T}_0 \arrow[r, mapsto] & \mathtt{induce}\ \mathtt{T}_0 \subseteq \mathtt{T}_1 \\
\mathtt{Theory}\ \mathtt{L}_0 \arrow[r, "\mathtt{induce}"] & \mathtt{Theory}\ \mathtt{L}_1
\end{tikzcd}$$

Specifically, the inductive step is

```
inductive henkin_language_functions (L : Language.{u}) : ℕ → Type u
| inc : ∀ {n}, L.functions n → henkin_language_functions n
| wit : bounded_formula L 1 → henkin_language_functions 0
```

At each step we make a language L_{i+1} inheriting all the function symbols from L_i via `inc`, and for each L_i-formula ϕ with one free variable, we introduce a new constant symbol `wit` ϕ for that specific formula.

Then we can then make the new L_{i+1}-theory T_{i+1} by taking the induced theory of T_i and adding a new sentence $\exists\phi \Rightarrow \phi($`wit` $\phi)$ (as an induced L_{i+1}-sentence) for each L_i-formula ϕ with one free variable.

```
def wit_property {L : Language} (f : bounded_formula L 1) (c : L.constants) :
  sentence L := (∃'f) ⟹ f[bd_const c/0]

def henkin_theory_step {L} (T : Theory L) : Theory $ henkin_language_step L :=
Theory_induced henkin_language_inclusion T ∪
(λ f : bounded_formula L 1,
  wit_property (henkin_language_inclusion.on_bounded_formula f) (wit’ f)) ’’ (set.univ : set $
    bounded_formula L 1)
```

Each T_{i+1} is consistent since T_i is consistent; a model of T_i will be a model of the new theory. Indeed if `a : M` realizes $\exists\phi$ then we can interpret `wit` ϕ as `a`, satisfying the new sentences in T_{i+1}.

Hence we can take the colimit of these languages `L_infty` (which amounts to a union of the function symbols in set theory)

$$\begin{tikzcd}
\mathtt{L}_0 \arrow[r] \arrow[drr] & \mathtt{L}_1 \arrow[r] \arrow[dr] & \cdots \arrow[d, dotted] \\
 & & \mathtt{L\_infty}
\end{tikzcd}$$

and take the union of the induced theories in `L_infty` to be our desired theory `T_infty`.

- `T_infty` is consistent since it is finitely consistent (at each step the new theory T_{i+1} is consistent and since the theories form a chain, any finite subset will be a subset of some T_i).

- `T_infty` is Henkin since any `L_infty`-formula in one free variable is an induced formula from some L_i, which is witnessed by a constant symbol from L_{i+1} according to theory T_{i+1}, which is embedded in `T_infty`.

Both of these steps are combined together as

```
def completion_of_henkinization {L} {T : Theory L} (hT : is_consistent T) : Theory
    (henkin_language) := sorry
```

We are now ready to prove Upwards Löwenheim-Skolem sans the part about cardinality.

```
theorem has_sized_model_of_has_infinite_model [is_algebraic L] {T : Theory L} {κ : cardinal}
  (hκ : ∀ n, #(L.functions n) ≤ κ) (hωκ : ω ≤ κ) :
  (∃ M : Structure L, nonempty M ∧ M ⊨ T ∧ infinite M) →
  ∃ M : Structure L, nonempty M ∧ M ⊨ T ∧ #M = κ :=
begin
  rintro ⟨ M , hM0, hMT, hMinf ⟩,
```

Supposing T is consistent, we can come up with a model M. We can then add κ many constant symbols to form L_κ and ensure they are all distinct in theory T_κ, which is consistent by our work above.

```
  set Tκ := union_add_distinct_constants T κ.out,
  have hTκ_consis := is_consistent_union_add_distinct_constants κ.out hMinf hMT,
```

We Henkinize T_κ then take the maximal consistent L_2-theory T_2 extending that (where $L_2 :=$ `henkin_language`). We know that this is consistent, maximal and Henkin, hence `term_model T2` satisfies exactly the formulas that appear in T_2. However, we need to take the `reduct` of `term_model T2`, since we only want an L-structure which models the L-theory T. The reduct simply takes the original carrier set and realizes symbols as the realization of their images in the extended language. This is denoted `M[[` $\iota : L_0 \to^L L_1$ `]]`, where M is an L_1-structure. We first take the reduct to L_κ, then down to L.

```
  set T2 := completion_of_henkinization hTκ_consis,
  use (term_model T2)[[ henkin_language_over ]]
    [[(Lhom.sum_inl : L →ᴸ L.sum (of_constants κ.out))]],
```

It remains to show that this reduct is non-empty and a model of T, which follows from general theory about reducts and API for `completion_of_henkinization` and `term_model`.

```
  split,
  -- the reduction of a non-empty model is non-empty
  { apply fol.nonempty_term_model, exact completion_of_henkinization_is_henkin _, },
  split,
  -- this reduction models T
  { apply Lhom.reduct_Theory_induced Lhom.sum.is_injective_inl,
    have h := reduct_of_complete_henkinization_models_T hTκ_consis,
    simp only [all_realize_sentence_union] at h,
    exact h.1 },
  sorry,
```

The final goal is finding the cardinality of `term_model`, which we explore in the next subsection.

7.2.4 Cardinality of `term_model`

Our goal is

```
⊢ # (term_model T2[[henkin_language_over]][[Lhom.sum_inl]]) = κ
```

which says the cardinality of the carrier type of the reduct of `term_model T2` to L is κ. We first show $(\geq)$

```
⊢ # κ ≤ (term_model T2[[henkin_language_over]][[Lhom.sum_inl]])
```

To show this, it suffices to show

- The carrier type of any `(Language.of_constants α)`-model of `distinct_constants α` is at least size $\#\alpha$
- The reduct of `term_model` to `Language.of_constants κ.out` is a model of `distinct_constants κ.out`
- The carrier type in the goal (the reduct of `term_model` to `L`) equals to the carrier type of the reduct of `term_model` to `Language.of_constants α`

The first point follows from our definition of `distinct_constants`: Suppose M is a `(Language.of_constants α)`-model of `distinct_constants α`. Then to show that $\#\alpha \leq \#M$ it suffices to show that the function taking any $a : \alpha$ to its realization in M is an injection.

```
lemma all_realize_sentence_distinct_constants (M : Structure _) (hM : M ⊨ distinct_constants α
    ) : #α ≤ #M :=
begin
  apply @cardinal.mk_le_of_injective _ _ (λ a, M.constants a),
  intros x y hfxy,
```

Let two terms x and $y : \alpha$ be equal upon realization in M, and suppose for a contradiction $x \neq y$. Then by definition the sentence $x \neq y$ is in of the theory `distinct_constants` $/al$, so x and y are not equal upon realization in M, a model of `distinct_constants`.

```
  by_contra' hxy,
  rw all_realize_sentence_image at hM,
  apply hM ⟨x,y⟩ hxy,
  simp only [Structure.constants] at hfxy,
  simp [bd_const, hfxy],
end
```

The second point follows from the fact that for any injective morphism of languages, the reduct of models of induced theories are models of the original theories.

The third point is just by simplification, which I have extracted for clarity. Putting the three parts together we have the inequality

```
    have hle : #κ.out ≤ #((term_model T2)[[henkin_language_over]]
              [[(Lhom.sum_inr : _ →ᴸ L.sum (of_constants κ.out))]]),
    { apply all_realize_sentence_distinct_constants,
      apply Lhom.reduct_Theory_induced Lhom.sum.is_injective_inr,
      have h := reduct_of_complete_henkinization_models_T hTκ_consis,
      simp only [all_realize_sentence_union] at h,
      exact h.2 },
    { simp only [fol.Lhom.reduct_coe, cardinal.mk_out] at hle ⊢,
      exact hle }
```

Now we show $(\leq)$.

```
⊢ # (term_model T2[[henkin_language_over]][[Lhom.sum_inl]]) ≤ κ
```

This will require opening up the definition of `term_model`. We know that `term_model T2` is a quotient of the type of closed terms in the language `henkin_language`, thus

```
lemma card_le_closed_term : #(term_model T) ≤ #(closed_term L) :=
cardinal.mk_le_of_surjective quotient.surjective_quotient_mk'
```

We see that we must investigate the cardinality of closed terms, or more generally terms and formulas. Since intuitively induction on terms and formulas produces well-founded trees stemming from formula (and relation) symbols, we should be able to bound `bounded_preterm L n l` and `bounded_formula L n` by the collection of function symbols in L. More precisely,

```
lemma bounded_preterm_le_functions {l} : #(bounded_preterm L n l) ≤
  max (cardinal.sum (λ n : ulift.{u} (ℕ), #(L.functions n.down))) ω := sorry

lemma bounded_formula_le_functions [is_algebraic L] {n} : #(bounded_formula L n) ≤
  max (cardinal.sum (λ n : ulift.{u} ℕ, #(L.functions n.down))) ω := sorry
```

We will prove these facts in a later section. For now, we conclude

```
lemma card_le_functions : #(term_model T) ≤
  max (cardinal.sum (λ n : ulift.{u} (ℕ), #(L.functions n.down))) ω :=
calc #(term_model T)
      ≤ #(closed_term L) : card_le_closed_term T
  ... ≤ max (cardinal.sum (λ n : ulift.{u} ℕ, #(L.functions n.down))) ω :
    cardinal.bounded_preterm_le_functions _
```

We can then extract the condition for which `term_model` is less than or equal to an infinite cardinal by simple cardinal arithmetic: it suffices that for each natural n, the number of function symbols with arity n is bounded by κ.

```
lemma card_le_cardinal {κ : cardinal.{u}} (hωκ : ω ≤ κ)
  (hκ : ∀ n : ulift.{u} ℕ, #(L.functions n.down) ≤ κ) : #(term_model T) ≤ κ :=
begin
  apply le_trans (card_le_functions T),
  apply max_le _ hωκ,
  apply le_trans (cardinal.sum_le_sup (λ n : ulift.{u} ℕ, #(L.functions n.down))),
  apply le_trans (cardinal.mul_le_max _ _),
  apply max_le _ hωκ,
  apply max_le,
  { simp [hωκ] },
  { rw cardinal.sup_le, exact hκ },
end
```

Now we can continue with our proof:

```
  apply term_model.card_le_cardinal T2 hωκ,
  intro n,
```

Our goal looks like

```
⊢ # (henkin_language.functions n.down) ≤ κ
```

We must investigate how many function symbols we have added during Henkinization. Since Henkinization is an inductive process adding L-formulas-with-one-free-variable many constant symbols at each step, this must be at most the collection of all function symbols or ω. Again we extract a lemma, saying that if an infinite cardinal κ bounds the function symbols in L above then κ bounds the function symbols of the henkinization of L above as well.

```
lemma henkin_language_le_cardinal [is_algebraic L] {T : Theory L}
  {hconsis : is_consistent T} (hωκ : ω ≤ κ)
  (hLκ : ∀ n, # (L.functions n) ≤ κ) (n : ℕ) :
  # ((@henkin_language _ _ hconsis).functions n) ≤ κ := sorry
```

We also prove this lemma in the next subsection. Note that we assume the language is algebraic for simplicity. This condition can be dropped but saves a bit of work for our use case. Indeed in our use case, the sum of algebraic languages is algebraic, L is assumed to be algebraic, and `of_constants κ.out` is clearly algebraic.

Proceeding with the proof, we simply need to show that for each natural m the language that we Henkinized has at most κ many function symbols with arity m. Since the sum of languages takes the disjoint sum of function symbols, the cardinality of the sum of function symbols is just the sum of the cardinalities of function symbols from each language.

```
  apply henkin_language_le_cardinal hωκ,
  { intro m,  -- the bound on function symbols
    simp only [Language.sum, cardinal.mk_sum, cardinal.lift_id],
```

The goal is now

```
⊢ # (L.functions m) + # ((of_constants κ.out).functions m) ≤ κ
```

By cardinal arithmetic it suffices to show that both parts of the sum are bounded by κ. The left is bounded by κ by assumption. The right is equal to κ when $m = 0$ by definition of `of_constants`, and otherwise is empty. Hence the sum is bounded by κ.

```
    apply le_trans (cardinal.add_le_max _ _),
    apply max_le _ hωκ,
    apply max_le,
    { apply hκ },
    { cases m,
      { simp [of_constants] },
      { simp [of_constants] } }
```

Hence we have completed the proof of Upwards Löwenheim-Skolem.

7.3 Cardinality lemmas

7.3.1 Terms

In this subsection we prove that the number of preterms is bounded by the number of function symbols in the language.

```
lemma bounded_preterm_le_functions {l} : #(bounded_preterm L n l) ≤
  max (cardinal.sum (λ n : ulift.{u} (ℕ), #(L.functions n.down))) ω := sorry
```

There should be many approaches to this problem. Mine was to note that preterms can be interpreted the collection of lists of preterm symbols that satisfy certain rules. Then the list of all these symbols can be easily bounded above. For example, the term `0 + (1 * x_0)` will be written as a list of symbols (the natural numbers for app will be explained later)

```
  [app,10]
    ++ ([app,1]
      ++ [func +]
      ++ ([app,4] ++ ([app,1] ++ [func *] ++ [func 1]) ++ [var 0]))
    ++ [func 0]
```

The **preterm symbols** can be made as an inductive type

```
inductive preterm_symbol (L : Language) : Type u
| nat : ℕ → preterm_symbol
| var : Π {l}, fin l → preterm_symbol
| func : Π {l}, L.functions l → preterm_symbol
| app : preterm_symbol
```

Then we **inject** `bounded_preterm L n l` into the collection of lists of preterm symbols.

```
def preterm_symbol_of_preterm {n} : ∀ {l},
  bounded_preterm L n l → list (preterm_symbol L)
| _ (&k)         := [ preterm_symbol.var k ]
| l (bd_func f)  := [ preterm_symbol.func f ]
| l (bd_app t s) := [ preterm_symbol.app,
  preterm_symbol.nat (preterm_symbol_of_preterm t).length ]
  ++ preterm_symbol_of_preterm t ++ preterm_symbol_of_preterm s
```

The choice of each list is designed to capture all the pieces of data that went into constructing the term. If the term was built as a variable `&k` then we only need to include the data that it was built as a variable symbol and that it used bounded natural k, so we take the list consisting of only the preterm symbol `[ preterm_symbol.var k ]`. The case for a function symbol is similar. More interestingly, when the preterm is built from applying a preterm `t : bounded_preterm L n (l + 1)` to a preterm `s : bounded_preterm L n 0`, we preserve the data of t and s by appending their inductively given lists to the end of everything else we need. It turns out that preserving the length of the list from t is important for showing injectivity.

To **show injectivity** of the above we induct on L-terms x and y. There are 9 cases to work on since there are 3 cases for x and y respectively.

```
lemma preterm_symbol_of_preterm_injective {l} :
  function.injective (@preterm_symbol_of_preterm L n l) :=
begin
  intros x,
  induction x with k _ _ _ tx sx htx hsx,
  { intro y,
    cases y,
    { intro h, simp only [...] at h, subst h },
    { intro h, cases h },
    { intro h, cases h } },
  { intro y,
    cases y,
    { intro h, cases h },
    { intro h, simp only [...] at h, subst h },
    { intro h, cases h } },
  { intro y,
    cases y with _ _ _ _ ty sy,
    { intro h, cases h },
    { intro h, cases h },
    { intro h, simp only [...] at h,
      obtain ⟨ ht , hs ⟩ := list.append_inj h.2 h.1,
      congr, { exact htx ht }, { exact hsx hs } } },
end
```

The cases where x and y are not built by the same constructor are easy to eliminate, since `no_confusion` for lists tells us two equal lists must have equal elements in the lists, and `no_confusion` for `preterm_symbol` tells us two equal preterm symbols must have come from the same constructor, which yields a contradiction in each case. This argument is hidden by the tactics `intro h, cases h`, where h is the assumption that x and y make equal lists of preterm symbols.

The remaining cases: when both are variable symbols or both are function symbols then we are assuming two lists with a single element are equal, since the elements are the same constructor applied to some variable, those variables must be equal by `no_confusion` for `preterm_symbol`. We thus have that $x = y$. In the case when x and y are both applications, we can use `no_confusion` for lists and apply injectivity of `list.append` to deduce each part of the list is equal and apply the induction hypothesis. Injectivity of `list.append` uses equality of lengths of the sublists, which is why we included that data in our definition of `preterm_symbol_of_preterm`.

Now that we have an injection into `list (preterm_symbol L)` we should **compute the cardinality** of `preterm_symbol L`, which will determine the cardinality of lists of them. We make a type equivalent to `preterm_symbol L`:

```
def preterm_symbol_equiv_fin_sum_formula_sum_nat :
  (preterm_symbol L) ≃
    (Σ l : ulift.{u} ℕ, ulift.{u} (fin l.down)) ⊕ (Σ l : ulift.{u} ℕ, L.functions l.down) ⊕
    ℕ := ...
```

This equivalence of types is obvious. Equivalent types have the same cardinality, so we can just compute the cardinality of the latter, for which there is plenty of API.

We are ready to complete the lemma. By the injection above we have the first inequality:

```
lemma bounded_preterm_le_functions {l} : #(bounded_preterm L n l) ≤
  max (cardinal.sum (λ n : ulift.{u} (ℕ), #(L.functions n.down))) ω :=
calc #(bounded_preterm L n l) ≤ # (list (preterm_symbol L)) :
    cardinal.mk_le_of_injective (@preterm_symbol_of_preterm_injective L n l)
```

For an infinite type α, $\#\alpha = \#\texttt{list}\ \alpha$. Then replacing the cardinality along the equivalence above, and going through some simple cardinal arithmetic proves the final inequality.

```
  ... = # (preterm_symbol L) : cardinal.mk_list_eq_mk (preterm_symbol L)
  ... ≤ max (cardinal.sum (λ n : ulift.{u} (ℕ), #(L.functions n.down))) ω :
begin
  rw cardinal.mk_congr (preterm_symbol_equiv_fin_sum_formula_sum_nat L),
  simp only [...],
  apply le_trans (cardinal.add_le_max _ _) (max_le (max_le _ _) (le_max_right _ _)),
  { apply le_max_of_le_right,
    apply le_trans (cardinal.sum_le_sup.{u} (λ (i : ulift.{u} ℕ), (i.down : cardinal.{u}))),
    apply le_trans (cardinal.mul_le_max _ _) (max_le (max_le _ _) (le_of_eq rfl)),
    { simp },
    { rw cardinal.sup_le, intro i, apply le_of_lt, rw cardinal.lt_omega, simp, }
  },
  { apply le_trans (cardinal.add_le_max _ _) (max_le (max_le _ _) (le_max_right _ _)),
    { simp },
    { exact le_max_right _ _ } }
end
```

7.3.2 Formulas

We make a similar construction for formulas, where we bound the number of formulas by the number of terms. We then utilize previous work bounding the number of terms by the number of function symbols to improve the bound:

```
lemma bounded_formula_le_bounded_term [is_algebraic L] {n} :
  #(bounded_formula L n) ≤ max (cardinal.sum (λ n : ulift.{u} ℕ, #(bounded_term L n.down))) ω
    := sorry

lemma bounded_formula_le_functions [is_algebraic L] {n} :
  #(bounded_formula L n) ≤ max (cardinal.sum (λ n : ulift.{u} ℕ, #(L.functions n.down))) ω :=
begin
  apply le_trans (bounded_formula_le_bounded_term L),
  apply max_le _ (le_max_right _ _),
  apply le_trans (cardinal.sum_le_sup _),
  simp only [cardinal.mk_denumerable],
  apply le_trans (cardinal.mul_le_max _ _),
  apply max_le _ (le_max_right _ _),
  apply max_le (le_max_right _ _),
  rw cardinal.sup_le,
  intro i,
  apply bounded_preterm_le_functions,
end
```

The **formula symbols** we need are

```
inductive formula_symbol (L : Language.{u}) : Type u
| bot : formula_symbol
| eq : formula_symbol
| imp : formula_symbol
| all : formula_symbol
| term : Π (l : ℕ), bounded_term L l → formula_symbol
| nat : ℕ → formula_symbol
```

The structure of induction on formulas is more interesting than that of terms, since the constructor `bounded_preformula.bd_all` takes preformulas with $n+1$ free variables and converts them to preformulas with n free variables. Thus we must keep track of the data of the number of free variables when we write a formula as a list of symbols. We do so when we **inject** formulas into lists of formula symbols.

```
def formula_symbol_of_formula [is_algebraic L] {n} :
  bounded_formula L n → list (formula_symbol L) :=
bounded_formula.rec2
  (λ l, [formula_symbol.nat l, formula_symbol.bot]) -- ⊥
  (λ l t s, [ formula_symbol.nat l, formula_symbol.eq ,
    formula_symbol.term l t , formula_symbol.term l s ]) -- t ≃ s
  (λ _ _ r, false.elim $ Language.is_algebraic.empty_relations _ r) -- bd_rel
  (λ l ϕ ψ lϕ lψ, (formula_symbol.nat l) :: (formula_symbol.nat (list.length lϕ))
    :: (formula_symbol.nat (list.length lψ)) :: formula_symbol.imp :: lϕ.append lψ ) -- ϕ ⟹
      ψ
  (λ l ϕ lϕ, (formula_symbol.nat l) :: formula_symbol.all :: lϕ) -- ∀ ϕ
```

Here we have taken advantage of an induction lemma made for `bounded_formula` rather than `bounded_preformula`. Note that the assumption `is_algebraic` here means that we need not worry about relation symbols. A generalization of this work would have to include the relation symbols in the final upper bound.

To show that this assignment is **injective**, we need to be careful. For the induction to work on the `bd_all` case, we need the inducion hypothesis to include the statement about formulas with $n+1$ many free variables. (This is possible since we are not inducting on the naturals, but on formulas.) We quickly find that we are in the territory of `heq`, since we need to ask for two terms of definitionally different types `bounded_formula L n` and `bounded_formula L m` to be equal. Thus we state it as follows:

```
lemma formula_symbol_of_preformula_injective' [is_algebraic L] {n} : ∀ (x : bounded_formula L
    n)
  {m} (y : bounded_formula L m),
  formula_symbol_of_formula x = formula_symbol_of_formula y → x == y := sorry

lemma formula_symbol_of_preformula_injective [is_algebraic L] {n}:
  function.injective (@formula_symbol_of_formula L _ n) :=
begin
  intros x y hxy,
  have h := formula_symbol_of_preformula_injective' x y hxy,
  subst h,
end
```

There are $6 * 6 = 36$ cases to check for the first lemma, of which 30 are quite easily resolved by applications of `no_confusion`. The rest of the 6 are resolved by careful handling of the induction hypothesis and heq.

We find **an equivalent type**, whose cardinality is easy to compute:

```
def formula_symbol_equiv_bounded_term_sum_nat :
  (formula_symbol L) ≃ ((Σ l : ulift.{u} ℕ, bounded_term L l.down) ⊕ ulift.{u} ℕ) := sorry
```

Finally we conclude our computation. By our injection into formula symbols,

```
lemma bounded_formula_le_bounded_term [is_algebraic L] {n} :
  #(bounded_formula L n) ≤ max (cardinal.sum (λ n : ulift.{u} ℕ, #(bounded_term L n.down))) ω
     :=
calc #(bounded_formula L n) ≤ # (list (formula_symbol L)) :
    cardinal.mk_le_of_injective (formula_symbol_of_preformula_injective)
```

Then since the type of formula symbols is infinite, the size of lists is equal to its original size, which is equal to the size of the equivalent type above.

```
  ... = # (formula_symbol L) : cardinal.mk_list_eq_mk _
  ... = _ : cardinal.mk_congr formula_symbol_equiv_bounded_term_sum_nat
```

The rest is simple cardinal arithmetic.

```
  ... ≤ max (cardinal.sum (λ n : ulift.{u} ℕ, #(bounded_term L n.down))) ω : by {
  simp only [le_refl, and_true, cardinal.mk_denumerable, cardinal.mk_sum, cardinal.lift_omega,
    cardinal.mk_sigma, cardinal.lift_id],
  apply le_trans (cardinal.add_le_max _ _) (max_le (max_le _ _) (le_max_right _ _)),
  { exact le_max_left _ _ },
  { exact le_max_right _ _ } }
```

7.3.3 Henkinization

We are showing

```
lemma henkin_language_le_cardinal [is_algebraic L] {T : Theory L}
  {hconsis : is_consistent T} (hωκ : ω ≤ κ)
  (hLκ : ∀ n, # (L.functions n) ≤ κ) (n : ℕ) :
  #((@henkin_language _ _ hconsis).functions n) ≤ κ := sorry
```

In `flypitch` they design a general process of taking the colimit of a collection (directed diagram) of languages, which specialises to making `henkin_language`. To bound the cardinality of the colimit of languages we just need that

- The function symbols are some quotient of a coproduct of languages.
- The coproduct of languages is formalized as a sigma type, whose cardinality can be computed as a sum of cardinalities.

We make a general lemma, which says if cardinal κ bounds the function symbols of each language in the directed diagram then κ also bounds the function symbols in the colimit of the directed diagram.

```
lemma colimit_language_le_cardinal {F : colimit.directed_diagram_language ℕ'}
  (n : ℕ) (h : ∀ i : ℕ, # ((F.obj i).functions n) ≤ κ) :
  # ((colimit.colimit_language F).functions n) ≤ κ :=
begin
  apply le_trans cardinal.mk_quotient_le,
```

The above says that since the function symbols are a quotient, and the cardinality of a quotient type is bounded by the original type it suffices that the original type (the coproduct of function symbols) is bounded by κ.

```
  dsimp only [colimit.coproduct_of_directed_diagram],
  rw cardinal.mk_sigma,
```

Since the coproduct is defined as a sigma type, we simply take the cardinality of the sigma type, which is a sum. The rest is cardinal arithmetic.

```
  rw cardinal.sum_nat,
  apply le_trans (cardinal.sum_le_sup _),
  simp only [cardinal.mk_denumerable],
  apply le_trans (cardinal.mul_le_max _ _) (max_le (max_le hωκ _) hωκ),
  rw cardinal.sup_le,
  intro i,
  cases i,
  apply h,
end
```

To apply this lemma to our situation, we will need to show that at each inductive Henkinization step, the language has function symbols bounded by κ. We prove this as well:

```
lemma henkin_language_chain_obj_card [is_algebraic L] {T : Theory L}
  (hconsis : is_consistent T)
  (hLκ : ∀ n, # (L.functions n) ≤ κ) (i : ℕ) :
  ∀ (n : ℕ), # (((@henkin_language_chain L).obj i).functions n) ≤ κ :=
```

If we induct on i, we can use the way in which each step in the Henkinization was built. When $i = 0$ the language is just L itself, which has function symbols bounded by assumption.

```
  simp only [henkin_language_chain, henkin_language_chain_objects],
  induction i with i hi,
  { apply hLκ },
```

For the inductive step, we should also case on n since we only add new constant symbols at each step.

```
  { intro n,
    cases n with n,
    { rw cardinal.mk_congr (@henkin_language_functions_zero (@henkin_language_chain_objects L
     i)),
      simp only [cardinal.mk_sum, cardinal.lift_id],
      apply le_trans (cardinal.add_le_max _ _),
      refine max_le (max_le (hi _) _) hωκ,
      apply bounded_formula_card_le hωκ hi, },
    { rw cardinal.mk_congr (@henkin_language_functions_succ (@henkin_language_chain_objects L
     i) n),
      apply hi } }
end
```

To explain the above:

- The constant symbols of any successive language (`henkin_language_chain_object i.succ`) biject with constant symbols from the previous language together with formulas with a single variable from the previous language:

  ```
  henkin_language_functions (henkin_language_chain_objects i) 0 ≃
    (henkin_language_chain_objects i).functions 0 ⊕ bounded_formula
     (henkin_language_chain_objects i) 1
  ```

  Hence it suffices to bound the sum of the two cardinalities by κ. By the induction hypothesis `hi` we have the bound for The constant symbols from the previous language. By our previous work `bounded_formula_card_le` we also have a bound on the number of formulas in the previous language, given a bound on the number of function symbols by the induction hypothesis `hi` again.

- The function symbols with arity $n+1$ of any successive language biject with the the function symbols with arity $n+1$ from the previous language:

  ```
  henkin_language_functions (henkin_language_chain_objects i) (n + 1) ≃
    (henkin_language_chain_objects i).functions (n + 1)
  ```

  Hence by rewriting our goal to be the cardinality of this type we can conclude using the induction hypothesis `hi`.

Using the above lemmas we prove the main result:

```
lemma henkin_language_le_cardinal [is_algebraic L] {T : Theory L}
  {hconsis : is_consistent T}
  (hLκ : ∀ n, # (L.functions n) ≤ κ) (n : ℕ) :
  # ((@henkin_language _ _ hconsis).functions n) ≤ κ :=
begin
  apply colimit_language_le_cardinal hωκ,
  intro i,
  apply henkin_language_chain_obj_card hωκ hconsis hLκ,
end
```

8 Appendix

8.1 Relational languages

Definition – Language of binary relations
Let the following be the language of binary relations:

- There are no function symbols.
- The only relation symbol is $<$, which has arity 2.

For variables x and y of type P, we write $x < y$ as notation for $< (x, y)$.

Definition – Graph theory
The theory of simple directed graphs is the language of binary relations is given by the single formula

- Non-reflexivity - $\forall x, \neg (x < x)$

Here the sort symbol is viewed as the set of vertices, and the relation symbol viewed as an edge between vertices. The fact that any model of this theory is a simple graph follows from non-reflexivity and proposition extensionality.

The theory of simple undirected graphs includes the formula above and also

- Symmetry - $\forall x\, y, x < y \to y < x$

Definition – Order theories
The theory of partial orders in the language of binary relations is given by the two formulas

- Non-reflexivity - $\forall x, \neg (x < x)$
- Transitivity - $\forall x\, y\, z, x < y \land y < z \Rightarrow x < z$

The theory of linear orders is the theory of partial orders plus

- Linearity - $\forall x\, y, x < y \lor x = y \lor y < x$

Definition – Theory of equivalence relations
The theory of equivalence relations in the language of binary relations is given by the three formulas

- Reflexivity - $\forall x, x < x$
- Symmetry - $\forall x\, y\, z, x < y \Rightarrow y < x$
- Transitivity - $\forall x\, y\, z, x < y \land y < z \Rightarrow x < z$

ZFC can also be described as a theory in the language of binary relations. The above theories all contain finitely many sentences, but one would need infinitely many sentences to make ZFC, since each axiom schema includes infinitely many sentences.

8.2 Actions

Definition – Monoid and group actions
Fix M a monoid. Let the following be the language of M-sets:

- For each monoid element a from M we have a function symbol ρ_a of arity 1.

- There are no relation symbols.

In the language of M-sets we let the theory of left monoid actions consist of the sentences

- Respecting multiplication - $\{\forall x, \rho_a(\rho_b x) = \rho_{ab} \,|\, a, b \in M\}$
- Respecting the identity - $\forall x, \rho_e x = x$, where e is the identity in M

The above automatically defines G-sets and left G actions for a group G.

Definition – Modules

Fix A a ring. Let the language of modules over A be the sum of the language A-sets and the language of groups. (We already made the language of groups and the theory of abelian groups whilst making the language of rings and theory of commutative rings.)

In the langauge of modules we make the theory of left modules by taking the union of the following

- The (induced) theory of left monoid actions from the language of A-sets.
- The (induced) theory of abelian groups from the language of groups.
- Ring and module distributivity:

$$\{\forall x\, y, \rho_a(x+y) = \rho_a x + \rho_a y \,|\, a \in A\} \cup \{\forall x, \rho_{a+b} x = \rho_a x + \rho_b x \,|\, a, b \in A\}$$

8.3 Inductive types

Definition – The naturals

The naturals convey the data of their induction principle, which is embodied in the following.

```
inductive nat
| zero : nat
| succ (n : nat) : nat
```

This definition of the naturals reads "the only things of type `nat` are `nat.zero` and `nat.succ n`, for some `n` of type `nat`".

The recursor for the naturals says to make a map out of the naturals into some type `A` (i.e. recursively make things of type `A`), it suffices to send `nat.zero` to something of type `A` and, given the image of `n`, to send `nat.succ n` to something of type `A`.

Definition – Lists

Given a type `T`, we can build lists of things of type `A`.

```
inductive list (T : Type u)
| nil : list
| cons (hd : T) (tl : list) : list
```

This definition of lists on a type `T` reads "the only things of type `list T` are `list.nil` and `list.cons hd tl`, for some `hd` of type `T` and some `tl` of type `list T`". The first constructor is the empty list and the second gives a way of making a new list by adding `hd` at the beginning of an existing list.

The recursor for lists says to make a map out of lists on a type into some type `A` (i.e. recurse on lists), it suffices to send the empty list to something of type `A` and, given the image of a list `tl`, to send `list.cons hd tl` to something of type `A`.

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.