Formalization of Fraïssé limits in Lean

Gabin Kolly

Born 30th August 1999 in Fribourg, Switzerland

February 13, 2025

Master's Thesis Mathematics

Advisor: Prof. Hieronymi

Second Advisor: Prof. van Doorn

Mathematisches Institut

Mathematisch-Naturwissenschaftliche Fakultät der

Rheinischen Friedrich-Wilhelms-Universität Bonn

Acknowledgments

I would like to thank Aaron Anderson for his precious guidance during the project, especially design decisions on how to implement the proof of the back-and-forth.
I'm grateful to Prof. Floris van Doorn for his assistance when I was completely stuck on a terrible recursive construction.
I appreciated Mario Carneiro and Kyle Miller taking the time to answer my questions on the Lean Zulip forum.
Finally, this work would not have been possible without the supervision, advice and encouragement from my advisor Prof. Philipp Hieronymi.

Introduction

In modern mathematics, studying embeddings between structures within a given class often yields deeper insights into the properties of that class. For some well-behaved classes of structures, restricting attention to finitely generated structures reveals particularly nice properties concerning embeddings. Such classes are referred to as Fraïssé classes, and they are characterized by their essential equivalence to the set of substructures of a countable homogeneous structure. This structure, known as the Fraïssé limit of the class, can be viewed as the most generic or universal object within the class.

This thesis presents progress in implementing key results about Fraïssé limits in Mathlib, the primary mathematical library for the Lean theorem prover. Fraïssé limits were first introduced by Roland Fraïssé, a French mathematical logician [Hod97]. We establish the existence of Fraïssé limits for Fraïssé classes and prove their uniqueness up to equivalence. Additionally, we develop the *back-and-forth method*, a fundamental tool in model theory often used to construct isomorphisms between countable structures.

All the non-formal proofs in this thesis are standard and could be found in most presentations on the subject, for example in Hodges' *A Shorter Model Theory* [Hod97].

Interactive theorem provers

Automated and interactive theorem provers have gained significant popularity in recent years, driven by multiple factors. As noted by Massot [Mas21], formalizing mathematics provides substantial benefits in terms of rigor and verification. The complexity of modern proofs continues to grow, with some famous cases—such as the classification of finite simple groups and Fermat's Last Theorem—reaching a size and intricacy that make it nearly impossible for any single mathematician to fully comprehend them. Computer-verified proofs offer greater assurance of correctness for such results.

Massot further argues that proof assistants are not only virtuous tools for ensuring rigor but can also be highly practical aids during the proof process. They help users keep track of the current goal, available assumptions, and intermediate results. Additionally, if a subtle change is made to a definition, the software can pinpoint where earlier parts of the proof may no longer hold. Of course, for the moment it is harder to use interactive theorem provers than to do mathematics in the classical way, but one could hope that

advances in the development of these proof assistants would offer more advantages than burden. In the future, such tools could even tailor the presentation of a proof to suit the reader's level of familiarity with the subject, translating formal arguments into varying levels of detail. Furthermore, interactive theorem provers provide a structured framework that is well-suited for training artificial intelligence to do mathematics, as demonstrated by recent advances in this domain [Alp24].

All the formal proofs in this work were carried out using Lean 4. Lean 4 is a functional programming language and interactive theorem prover based on dependent type theory. One of its standout features is the ability to write custom automation scripts, called tactics, to streamline parts of the proof. The first version of Lean, developed by Leonardo de Moura, was released in 2015, and Lean 4, the successor to Lean 3, was published in 2021 [MU21]. Lean has been used to formalize many significant and complex proofs. A particularly notable project is the *Liquid Tensor Experiment*, in which a team spent two years formalizing a key lemma from the recent and highly technical work of Scholze and Clausen [Com22].

Mathlib

Mathlib is a collaborative, open-source project aimed at creating a unified library of pure mathematics in Lean, with the goal of formalizing foundational results across most domains [Com20] [Mat21]. Currently, the library is most developed in areas such as algebra and analysis, while less progress has been made in fields like algebraic and geometric topology, classical geometry, and more applied disciplines such as probability and numerical analysis. As of January 2025, Mathlib contains over 92,000 definitions and 179,000 theorems and has had contributions from more than 370 developers [Mat25].

Unlike many libraries for interactive theorem provers, Mathlib emphasizes classical mathematics over constructive mathematics. It is designed with a high level of modularity and compatibility between its components, ensuring maintainability, scalability, and the ability to serve as a common framework and language for projects in Lean. Another distinctive feature of Mathlib is its decentralized governance model: decisions are made collaboratively by the community rather than being centralized in a small group. This approach aligns with the breadth and depth of mathematics, where researchers often specialize in only a small subset of the field.

Contributions to Mathlib

One objective of this thesis is to contribute code to Mathlib, expanding its coverage of basic model theory. This requires adhering to Mathlib's conventions on style, naming, and theorem organization, while ensuring the code meets the library's standards for usability and maintainability. As of January 2024, the formalization of the back-and-forth method and the proof of the uniqueness of Fraïssé limits have been already integrated to Mathlib, with the help of Aaron Anderson [KA24]. The proof of the existence of Fraïssé limits and the characterization of the Fraïssé limit of finite simple graphs are for the moment proposed as pull requests for the library and need to be reviewed by others in

the community [Kol24] [Kol25].

Previous works

Few theorems from model theory have been formalized with proof assistants. Proofs of Gödel's completeness [JJS12] [Fro21] [Ber07] and incompleteness theorems [Pau13] [O'C05] [Tea23], the compactness theorem [SdLAR24], the Löwenheim-Skolem theorems [Ber07] and Łoś's theorem [Mat21] have been done in multiple proof assistants. All of these theorems are also in Mathlib, except the incompleteness theorems. The subjects of categoricity, stability, and relations with geometry have been mostly unexplored. In Lean, the *Flypitch project* [HvD19] [HvD20], done by Han and von Doorn, proved the independence of the continuum hypothesis. The Flypitch project is particularly important for this work, since they made some of the basic definitions and theorems from first order logic and model theory which were added to Mathlib and used in this work.

Contents

Introduction 1

Conventions 5

1 Implementation of Model Theory in Lean 6
- 1.1 Language . . . 6
- 1.2 Structures . . . 6
- 1.3 Embeddings and equivalences . . . 6
- 1.4 Substructures . . . 7
- 1.5 Direct limits . . . 9
- 1.6 Equivalences between substructures . . . 11

2 Back-And-Forth 12
- 2.1 Limit of a directed system of partial equivalences . . . 13
- 2.2 Proof of the back-and-forth . . . 16

3 Fraïssé Limits 20
- 3.1 Bundled structures . . . 20
- 3.2 Embeddings between equal structures . . . 21
- 3.3 Definition of Fraïssé limits . . . 22
- 3.4 Cardinality of FGEquiv . . . 27
- 3.5 Fraïssé limits exist . . . 29
- 3.6 Fraïssé limits are unique . . . 42
- 3.7 Fraïssé limit of finite graphs . . . 45

4 Future works 54

Bibliography 55

Conventions

In this paper, we will use the following conventions:

- L will always designate a language.
- M and N will be L-structures.
- Since we will never have two different languages at the same time, we will say "structure" instead of "L-structure".

All the snippets of code will be mostly the same as what is present in the code files. There will be minor changes that help comprehension and make the links with the mathematical proofs clearer, for example change of names of variables. We will also remove technical modifiers, for example `protected` or `private` keywords before theorems, since they are not important for this discussion and are related to the organization of the Mathlib library. When the code is not written by the author of the thesis, it will be said before the code. Its source will always be Mathlib [Mat21] or a branch of Mathlib, which will be indicated by a citation, and we will also indicate where in Mathlib the file containing the code is located by writing "Code in X/Y" to indicate the file 'Mathlib/X/Y.lean'. When the code is written by someone else, comments by the author will sometimes be added in the code, preceded by "author:".

1 Implementation of Model Theory in Lean

We will present the formalization of basic definitions from model theory that we will need. The way languages and structures are formalized has been developed by Jesse Michael Han and Floris van Doorn during the Flypitch project [HvD20], and Aaron Anderson has written the definitions and results about substructures and direct limits that we will present in this section for Mathlib. Only the subsection about equivalence between substructures is original work.

1.1 Language

A language in Lean is defined as a structure with two functions, giving for each natural number a type of symbols for the functions of that arity, and a type of symbols for the relations of that arity.

Language

```
-- Code in ModelTheory/Basic, written by Han and van Doorn [AHvD]
structure Language where
  /-- For every arity, a `Type*` of functions of that arity -/
  Functions : ℕ → Type u
  /-- For every arity, a `Type*` of relations of that arity -/
  Relations : ℕ → Type v
```

For the rest of this paper, the variable L will always be a language.

1.2 Structures

We can define an L-structure on a type M as interpretations of the function and relation symbols as functions and relations on M:

Structure

```
-- Code in ModelTheory/Basic, written by Han and van Doorn [AHvD]
class Structure where
  /-- Interpretation of the function symbols -/
  funMap : ∀ {n}, L.Functions n → (Fin n → M) → M
  /-- Interpretation of the relation symbols -/
  RelMap : ∀ {n}, L.Relations n → (Fin n → M) → Prop
```

The type `Fin n → M` is the type of n-tuples of elements of M, so at each symbol of functions, funMap associates a symbol of function to a function from n-tuples to M, i.e. an n-ary function. Similarly, RelMap associates to any relation symbol a function which takes as input an n-tuple and returns an element of Prop, so either True or False.

1.3 Embeddings and equivalences

Embedding are structures (in the programming sense) with a function and propositions saying that it is injective and commutes with the functions and relations.

Embedding

```
-- Code in ModelTheory/Basic, written by Han and van Doorn [AHvD]
structure Embedding extends M ↪ N where
  map_fun' : ∀ {n} (f : L.Functions n) (x), toFun (funMap f x) = funMap f
    (toFun ∘ x)
  map_rel' : ∀ {n} (r : L.Relations n) (x), RelMap r (toFun ∘ x) ↔ RelMap r x

scoped[FirstOrder] notation:25 A " ↪[" L "] " B =>
    FirstOrder.Language.Embedding L A B
```

The last line allows us to write `M ↪[L] N` for the type of embeddings from M to N. This structure is an extension of the non-model theoretic embedding structure, written `M ↪ N`, which is the type of injective functions from M to N, which is why the proposition that the function is not injective is not present in the definition above.

We also have equivalences as bijective functions which commutes with functions and relations:

Equiv

```
-- Code in ModelTheory/Basic, written by Han and van Doorn [AHvD]
structure Equiv extends M ≃ N where
  map_fun' : ∀ {n} (f : L.Functions n) (x), toFun (funMap f x) = funMap f
    (toFun ∘ x)
  map_rel' : ∀ {n} (r : L.Relations n) (x), RelMap r (toFun ∘ x) ↔ RelMap r x

scoped[FirstOrder] notation:25 A " ≃[" L "] " B => FirstOrder.Language.Equiv L
    A B
```

The last line allows us to write the type of equivalences between M and N as `M ≃[L] N`. This structure extends the structure of non-model theoretic equivalences, written `M ≃ N`, which is the type of bijective functions.

Both embeddings and equivalences have a FunLike instance, which allows us to apply them to elements directly, like functions, and write `f m` for the image of m by the underlying function of f, for f an embedding or an equivalence:

```
variable (f : M ↪[L] N) (m : M)
#check f m -- Lean message: f m : N, confirming that f m is of type N
```

However, to compose them, we don't write $f \circ g$, but f.comp g, otherwise Lean composes them as functions and outputs a function.

1.4 Substructures

We will also need the definition of substructures. The type of substructures of M is a structure containing a subset of M and a proof that it is closed under all the interpretations of the function symbols:

Substructure

```
-- Code in ModelTheory/Substructures, written by Anderson [AKc]
```

```
def ClosedUnder : Prop :=
  ∀ x : Fin n → M, (∀ i : Fin n, x i ∈ s) → funMap f x ∈ s

structure Substructure where
  carrier : Set M
  fun_mem : ∀ {n}, ∀ f : L.Functions n, ClosedUnder f carrier
```

They have a SetLike instance, so Lean can coerce a substructure to a subset if needed, for example for a substructure S and an element m, we can write $m \in S$, to signify that m is contained in the substructure. A set S of a type T can be coerced to a type, which will be the type of pairs of elements of T with a Prop that they are in S, so any substructure has also a coercion to a type, and we can write `m : S`. In this case, m can be understood as a pair of an element of M (the whole structure) and a proposition saying that it is contained in S. We can define an instance of structure on each substructure, more precisely on the coercion of this substructure to a type, so that a substructure of a structure is automatically considered as a structure by Lean. This is essential to work comfortably with them:

Induced structure on substructure

```
-- Code written by Anderson [AKc]
instance inducedStructure {S : L.Substructure M} : L.Structure S where
  funMap {_} f x := ⟨funMap f fun i => x i, S.fun_mem f (fun i => x i) fun i
    => (x i).2⟩
  RelMap {_} r x := RelMap r fun i => (x i : M)
```

There is a partial order on substructures induced by the partial order on subsets, and this partial order is a complete lattice, meaning that we can take the infimum and supremum of sets of substructures. For a subset S of M, we have a function which returns the substructure generated by s, defined here as the infimum of substructures containing s:

Generated substructure

```
-- Code in ModelTheory/Substructures, written by Anderson [AKc]
/-- The ‘L.Substructure‘ generated by a set. -/
def closure : LowerAdjoint ((↑) : L.Substructure M → Set M) :=
  ⟨fun s => sInf { S | s ⊆ S }, fun _ _ =>
    ⟨Set.Subset.trans fun _x hx => mem_sInf.2 fun _S hS => hS hx, fun h =>
    sInf_le h⟩⟩
```

We have a bit more information in fact: the definition also tells us that the closure forms a Galois connection with the map sending a substructure to its underlying subset. We can define a countably, respectively finitely generated substructure as a substructure which is equal to the closure of a countable, respectively finite set:

Countably and finitely generated substructures

```
-- Code in ModelTheory/FinitelyGenerated, written by Anderson [AKb]
/-- A substructure of ‘M‘ is countably generated if it is the closure of a
    countable subset of ‘M‘.-/
```

```
def CG (N : L.Substructure M) : Prop :=
  ∃ S : Set M, S.Countable ∧ closure L S = N

/-- A substructure of `M` is finitely generated if it is the closure of a
    finite subset of `M`. -/
def FG (N : L.Substructure M) : Prop :=
  ∃ S : Finset M, closure L S = N
```

We write $\top$ for the substructure corresponding to the whole structure, and a structure is finitely, resp. countably generated if $\top$ is finitely, resp. countably generated:

Countably and finitely generated structures

```
-- Code in ModelTheory/FinitelyGenerated, written by Anderson [AKb]
/-- A structure is countably generated if it is the closure of a countable
    subset. -/
class CG : Prop where
  out : (⊤ : L.Substructure M).CG

/-- A structure is finitely generated if it is the closure of a finite subset.
    -/
class FG : Prop where
  out : (⊤ : L.Substructure M).FG
```

There are four other definitions that we will use: the preimage of a substructure along a homomorphism, its image along a homomorphism, its inclusion in the whole structure, and for two substructures A, B such that $A \leq B$, the inclusion $A \hookrightarrow B$:

Important functions on substructures

```
-- Code in ModelTheory/Substructures, written by Anderson [AKc]
/-- The preimage of a substructure along a homomorphism is a substructure. -/
def comap (φ : M →[L] N) (S : L.Substructure N) : L.Substructure M := ...

/-- The image of a substructure along a homomorphism is a substructure. -/
def map (φ : M →[L] N) (S : L.Substructure M) : L.Substructure N := ...

/-- The natural embedding of an `L.Substructure` of `M` into `M`. -/
def subtype (S : L.Substructure M) : S ↪[L] M := ...

/-- The embedding associated to an inclusion of substructures. -/
def inclusion {S T : L.Substructure M} (h : S ≤ T) : S ↪[L] T := ...
```

1.5 Direct limits

If we have a directed system of structures, we can take its direct limit. We define a directed system of structures as a set S of structures, with a partial order on it, and for each $M \leq N$, we have an embedding $f_{MN} : M \hookrightarrow N$, so that for any $M \leq N \leq O$, we have $f_{MO} = F_{NO} \circ F_{MN}$. We also ask that the partial order be directed, i.e. any two elements have an upper bound. In Lean, the setting looks like this:

Setting for direct limits

```
variable {ι : Type v} [Preorder ι] -- The set of indices, with a preorder on it.
variable {G : ι → Type w}
-- Author: This is the definition of a directed system in Mathlib, which could
    be interpreted as a functor from a preorder to some category. Confusingly,
    the definition does not require the preorder to be directed, probably
    because it was defined originally for systems of modules in algebra, and
    in this case it is not necessary for the system to be directed to be able
    to define a limit.
class DirectedSystem (f : ∀ i j, i ≤ j → G i → G j) : Prop where
  map_self' : ∀ i x h, f i i h x = x
  map_map' : ∀ {i j k} (hij hjk x), f j k hjk (f i j hij x) = f i k (le_trans
    hij hjk) x

variable {L : Language} [∀ i, L.Structure (G i)]
variable (f : ∀ i j, i ≤ j → G i ↪[L] G j)
-- Author: We write `[DirectedSystem G fun i j h => f i j h]` and not
    `[DirectedSystem G f]`, because here Lean wants a function that outputs a
    function, not an embedding, and f outputs an embedding, so we replace it
    by the function which to i j h associates `f i j h`. Of course `f i j h`
    is still an embedding, but Lean coerces it automatically to the underlying
    function, so this is equivalent to `[DirectedSystem G (fun i j h ↦ (f i j
    h).toFun)]`
variable [IsDirected ι (· ≤ ·)] [DirectedSystem G fun i j h => f i j h]
```

So instead of a directed set of structures, we have a directed type ι, a map G associating to any index a type, and any type in the image of G has an instance of structure. This is because it is generally much more convenient to index structures with a set of indices that is already ordered, for example if you construct recursively a sequence of structures. We define an equivalence relation on the disjoint union of all the structures, generated by the maps between the structures, so if $f_{i,j}(a) = b$ for $a \in S\ i$, $b \in S\ j$, then $a \sim b$. The direct limit is defined as the quotient of the disjoint union with respect to this equivalence relation:

Direct limit

```
-- Code in ModelTheory/DirectLimit, written by Anderson [AKa]
/-- The direct limit of a directed system is the structures glued together
    along the embeddings. -/
def DirectLimit [DirectedSystem G fun i j h => f i j h] [IsDirected ι (· ≤ ·)]
    :=
  Quotient (DirectLimit.setoid G f)
```

We also define a structure on it. There are two important maps related to a direct limit: lifts that we get from its universal property, and embeddings from any indexed structure to the direct limit:

```
-- Code in ModelTheory/DirectLimit, written by Anderson [AKa]
/-- The canonical map from a component to the direct limit. -/
def of (i : ι) : G i ↪[L] DirectLimit G f where
```

```
  -- Author: ‘Sigma.mk‘ takes the image in the disjoint union, and ‘⟦ · ⟧‘
     sends to the quotient.
  toFun := fun a => ⟦.mk f i a⟧
  ...

-- Author: The ‘of‘ maps commute when composed with the maps between the
    components.
theorem of_f {i j : ι} {hij : i ≤ j} {x : G i} : of L ι G f j (f i j hij x) =
    of L ι G f i x := ...

/-- The universal property of the direct limit: maps from the components to
    another module that respect the directed system structure (i.e. make some
    diagram commute) give rise to a unique map out of the direct limit. -/
def lift (g : ∀ i, G i ↪[L] P) (Hg : ∀ i j hij x, g j (f i j hij x) = g i x) :
    DirectLimit G f ↪[L] P where
  ...

-- Author: ‘lift‘ verifies the universal property.
theorem lift_of {i} (x : G i) : lift L ι G f g Hg (of L ι G f i x) = g i x :=
```

1.6 Equivalences between substructures

From now on, the code has been written by the author, except when we mention otherwise. To be able to write a simple proof of the back-and-forth method in Lean, we need a practical way to deal with equivalences between substructures. So in this work we defined a new type of equivalences between substructures, and developed an API to work with them. We call equivalences between substructures of two structures M and N partial equivalences between M and N.

Partial equivalences

```
-- Code in ModelTheory/PartialEquiv [AKWa]
structure PartialEquiv where
  /-- The substructure which is the domain of the equivalence. -/
  dom : L.Substructure M
  /-- The substructure which is the codomain of the equivalence. -/
  cod : L.Substructure N
  /-- The equivalence between the two subdomains. -/
  toEquiv : dom ≃[L] cod

scoped[FirstOrder] notation:25 M " ≃ₚ[" L "] " N =>
  FirstOrder.Language.PartialEquiv L M N
```

It is simply a structure with a substructure of M, a substructure of N, and an equivalence between them. We could have a shorter definition, with only a substructure A of M, and an embedding $f : A \hookrightarrow N$, but we often need to access both the domain and codomain of a PartialEquiv, and the inverse of a partial equivalence is easier to define and work with. We could also have defined it as a subset of $M \times N$ with some properties, but most

important definitions, like the domain and codomain, would be more complicated to define. We can use the notation $M \simeq_p[L] N$ to designate the type of partial equivalences between M and N. The partial order on partial equivalences was defined as follows:

Partial order on partial equivalences

```
-- Code in ModelTheory/PartialEquiv [AKWa]
instance : LE (M ≃ₚ[L] N) :=
  ⟨fun f g ↦ ∃ h : f.dom ≤ g.dom,
    (subtype _).comp (g.toEquiv.toEmbedding.comp (Substructure.inclusion h)) =
      (subtype _).comp f.toEquiv.toEmbedding⟩
```

The first part is of course that the domain of f is contained in the domain of g, but saying that $f(x) = g(x)$ for all x in the domain of f is a bit tricky, because f and g don't have the same codomain as functions. Their codomains are substructures, not the full structure. So we compose with additional maps so that we have both sides with the same domain and codomain, giving

$$(f.dom \hookrightarrow g.dom \overset{g}{\simeq} g.cod \hookrightarrow M) = (f.dom \overset{f}{\simeq} f.cod \hookrightarrow M)$$

Definition 1.1 (Mapping of a partial equivalence)**.** For $g : A \simeq B$ an equivalence between substructures of M, and $f : M \hookrightarrow N$, we define the mapping of g through f as

$$f \circ g \circ (f^{-1} \restriction f(A)) : f(A) \simeq f(B)$$

In Lean:

Mapping of a partial equivalence

```
-- Code in ModelTheory/PartialEquiv [Kol24]
/-- Map of a self-PartialEquiv through an embedding. -/
def map (f : M ↪[L] N) (g : M ≃ₚ[L] M) : N ≃ₚ[L] N where
  dom := g.dom.map f.toHom
  cod := g.cod.map f.toHom
  -- ‘f.substructureEquivMap A‘ is the equivalence between ‘A‘ and ‘f(A)‘, so
    the restriction of ‘f‘ to ‘A‘. The notation ‘<|‘ is equivalent to putting
    everything after between parentheses, it is used sometimes to make the
    code cleaner instead of having a lot of parentheses.
  toEquiv := (f.substructureEquivMap g.cod).comp <|
    g.toEquiv.comp (f.substructureEquivMap g.dom).symm
```

We will write `g.map f` for the mapping of g through f.

2 Back-And-Forth

A basic technique in model theory is the back-and-forth method, and we formalized it in Lean in the course of this work, since it is used to show both the existence and uniqueness of Fraïssé limits. The main theorem can be formulated as follows:

Theorem 2.1 (Back-And-Forth)**.** For two countable structures M and N, $A \subseteq M$ and $B \subseteq N$ finitely generated substructures, and $f : A \simeq B$ an equivalence, if any equivalence between finitely generated substructures of M and N can be extended to have any element of M in its domain or any element of N in its image, then there is an equivalence $g : M \simeq N$ extending f.

Proof. Let $M = \{m_1, m_2, \ldots\}$ and $N = \{n_1, n_2, \ldots\}$ be enumerations of M and N, we define recursively a sequence $f_0 := f, f_1, \ldots$ of equivalences between finitely generated substructures in the following way: we get f_{k+1} by extending f_k so that it has m_{k+1} in its domain and n_{k+1} in its image, and then restricting it to the substructure generated by the domain of f_k, m_{k+1}, and the preimage of n_{k+1}, so that its domain and image are still finitely generated. The union of these maps is still an equivalence, and it has M as domain and N as image. □

In the proof of the back-and-forth, we will construct the equivalence as the limit of partial equivalences. The proof of the back-and-forth was adapted in part from David Wärn's approach to countable dense linear orders which was already present in Mathlib, which was a specialization of the back-and-forth for linear orders. We will also prove a very similar result, a sort of half back-and-forth. We will use it when we'll characterize the Fraïssé limit of finite simple graphs:

Theorem 2.2. For a countable structures M and a structure N, $A \subseteq M$ and $B \subseteq N$ finitely generated substructures, and $f : A \simeq B$ an equivalence, if any equivalence between finitely generated substructures of M and N can be extended to have any element in M in its domain, then there is an embedding $g : M \hookrightarrow N$ extending f.

2.1 Limit of a directed system of partial equivalences

During the proof of the back-and-forth, we will construct an increasing sequence of partial equivalences, and we will take its limit, so we need to define what the limit of a directed system of partial equivalences is. There are two lemmas about direct limits that were needed for the definition. The first one is:

Lemma 2.3. Let M be a structure, and S a directed system of substructures of M. Then

$$\varinjlim S \simeq \bigcup S$$

The proof is quite simple: since each substructure A in S has an embedding $A \hookrightarrow \bigcup S$, and these embeddings commute with the inclusions between substructures, you get an embedding from the direct limit by the universal property of the direct limit, and it is surjective, therefore it is an equivalence. Here is how we write this in Lean:

```
-- Code in ModelTheory/DirectLimit, written by Anderson [AKWa]
noncomputable def Equiv_iSup :
-- Author: We have to write ‘(iSup S : L.Substructure M)‘ and not just ‘iSup
    S‘, because otherwise Lean gets confused and cannot find what the type of
    the output is supposed to be, for reasons unclear to the author. ‘iSup‘ is
    of type ‘iSup.{u, v} {α : Type u} {ι : Sort v} [SupSet α] (s : ι → α) : α
    ‘, so the end type and its ‘SupSet‘ instance are implicit, and apparently
    it is too much uncertainty for Lean here, even though it should be able to
    deduce that α is ‘L.Substructure M‘ since it is the codomain of ‘S‘.
    Similarly, if we write ‘DirectLimit S‘, Lean gets confused, because it
    wants a function which associates to each element of ι a type, not a
    substructure, and it wants an L-structure instance on this type.
    DirectLimit (fun i ↦ S i) (fun _ _ h ↦ Substructure.inclusion
    (S.monotone h)) ≃[L]
    (iSup S : L.Substructure M) := by
  -- Author: ‘liftInclusion‘ is the embedding from the direct limit into ‘M‘
    and this lemma says that its image is contained in the supremum.
  have liftInclusion_in_sup : ∀ x, liftInclusion S x ∈ (⨆ i, S i) := by
    simp only [← rangeLiftInclusion, Hom.mem_range, Embedding.coe_toHom]
    intro x; use x
  -- Author: This is the restriction of ‘liftInclusion‘ so that its codomain
    is the supremum, instead of being the whole structure.
  let F := Embedding.codRestrict (⨆ i, S i) _ liftInclusion_in_sup
  have F_surj : Function.Surjective F := by
    rintro ⟨m, hm⟩
    rw [← rangeLiftInclusion, Hom.mem_range] at hm
    rcases hm with ⟨a, _⟩; use a
    simpa only [F, Embedding.codRestrict_apply', Subtype.mk.injEq]
  -- Author: From F, and the fact that it is injective and surjective, we get
    an equivalence.
  exact ⟨Equiv.ofBijective F ⟨F.injective, F_surj⟩, F.map_fun', F.map_rel'⟩
```

We will need another result: two isomorphic systems have isomorphic direct limits. More precisely:

Lemma 2.4. For two systems G and G' of structures and embeddings indexed by ι, and for each index i, an equivalence $g_i : G\ i \simeq G'\ i$ such that everything commutes, then the direct limit of G is isomorphic to the direct limit of G'.

Here is the proof in Lean:

Limits of isomorphic systems

```
-- Code in ModelTheory/DirectLimit [AKa]
variable (g : ∀ i, G i ≃[L] G' i)
/-- The isomorphism between limits of isomorphic systems. -/
noncomputable def equiv_lift (H_commuting : ∀ i j hij x, g j (f i j hij x) =
    f' i j hij (g i x)) : DirectLimit G f ≃[L] DirectLimit G' f' := by
```

```
  -- Each component ‘G i‘ has an embedding ‘U i‘ to the direct limit of ‘G’‘.
  let U i : G i ↪[L] DirectLimit G' f' := (of L _ G' f' i).comp (g
    i).toEmbedding

  -- ‘F‘ is the lift that we get from all the ‘U i‘ maps. We just need to
    prove that it is surjective.
  let F : DirectLimit G f ↪[L] DirectLimit G' f' := lift L _ G f U <| by
    intro _ _ _ _
    simp only [U, Embedding.comp_apply, Equiv.coe_toEmbedding, H_commuting,
    of_f]

  have surj_f : Function.Surjective F := by
    intro x
    -- We have some element ‘x‘ in the direct limit of ‘G’‘, and we need to
    find a preimage.
    -- This just says that ‘x‘ is represented by an element ‘pre_x‘ in some
    component ‘G' i‘.
    rcases x with ⟨i, pre_x⟩
    -- We take the preimage by ‘g i‘, and then we map it to the direct limit
    of ‘G‘.
    use of L _ G f i ((g i).symm pre_x)
    -- We simplify everything to show that this element is indeed sent to ‘x‘
    by ‘F‘.
    simp only [F, U, lift_of, Embedding.comp_apply, Equiv.coe_toEmbedding,
    Equiv.apply_symm_apply]
    rfl

  -- From a surjective embedding, we can get an equivalence.
  exact ⟨Equiv.ofBijective F ⟨F.injective, surj_f⟩, F.map_fun', F.map_rel'⟩
```

We apply lemmas 2.3 and 2.4 to define the limit of a directed system of partial equivalences:

Direct limit of PartialEquivs

```
-- Code in ModelTheory/PartialEquiv [AKWa]
variable {ι : Type*} [Preorder ι] [Nonempty ι] [IsDirected ι (· ≤ ·)]
-- ‘A →o B‘ is the type of monotone maps from ‘A‘ to ‘B‘.
variable (S : ι →o M ≃ₚ[L] N)

/-- The limit of a directed system of PartialEquivs. -/
noncomputable def partialEquivLimit : M ≃ₚ[L] N where
  dom := iSup (fun i ↦ (S i).dom)
  cod := iSup (fun i ↦ (S i).cod)
  toEquiv :=
    (Equiv_iSup {
      toFun := (fun i ↦ (S i).cod)
      monotone' := monotone_cod.comp S.monotone}
    ).comp
```

```
((DirectLimit.equiv_lift L ι (fun i ↦ (S i).dom)
  (fun _ _ hij ↦ Substructure.inclusion (dom_le_dom (S.monotone hij)))
  (fun i ↦ (S i).cod)
  (fun _ _ hij ↦ Substructure.inclusion (cod_le_cod (S.monotone hij)))
  (fun i ↦ (S i).toEquiv)
  (fun _ _ hij _ ↦ toEquiv_inclusion_apply (S.monotone hij) _)
).comp
  (Equiv_iSup {
    toFun := (fun i ↦ (S i).dom)
    monotone' := monotone_dom.comp S.monotone}).symm)
```

So the equivalence is defined as the composition

$$\bigcup_i (S\ i).\mathrm{dom} \simeq \lim_i (S\ i).\mathrm{dom} \simeq \lim_i (S\ i).\mathrm{cod} \simeq \bigcup_i (S\ i).\mathrm{cod}$$

We get the first and third equivalences by lemma 2.3, and the second equivalence by lemma 2.4. It has the crucial property that it extends any partial equivalence in the system:

```
theorem le_partialEquivLimit (i : ι) : S i ≤ partialEquivLimit S := ...
```

In particular, its domain is greater than the domain of each component, and its codomain is greater than the codomain of each component.

2.2 Proof of the back-and-forth

We define FGEquiv the type of partial equivalences between finitely generated substructures:

FGEquiv

```
-- Code in ModelTheory/PartialEquiv [AKWa]
/-- The type of equivalences between finitely generated substructures. -/
abbrev FGEquiv := {f : M ≃ₚ[L] N // f.dom.FG}
```

For any `f : L.FGEquiv M N`, we write `f.val` for the underlying partial equivalence, `f.val.dom` for its domain and `f.val.cod` for its codomain.

We will also use this definition:

Definition 2.5 (Extension pair)**.** Two structures M and N form an extension pair if for any equivalence f between finitely generated substructures of M and N, and any element m in M, f can be extended to contain m in its domain.

In Lean [AKWa]:

Extension pair

```
/-- Two structures `M` and `N` form an extension pair if the domain of any
    finitely-generated map from `M` to `N` can be extended to include any
    element of `M`. -/
def IsExtensionPair : Prop := ∀ (f : L.FGEquiv M N) (m : M), ∃ g, m ∈ g.1.dom ∧
    f ≤ g
```

Therefore, in the case of the back-and-forth, we have that M and N form an extension pair, and also that N and M form an extension pair (this is not a symmetric relation). Another way to formulate this property is to say that if M and N form an extension pair, then for any $m \in M$, we have that the FGEquivs with m in their domains form a cofinal set in FGEquiv, i.e. they contain arbitrarily large elements.

definedAtLeft and definedAtRight

```
-- Code in ModelTheory/PartialEquiv [AKWa]
/-- The cofinal set of finite equivalences with a given element in their
    domain. -/
def definedAtLeft
    (h : L.IsExtensionPair M N) (m : M) : Order.Cofinal (FGEquiv L M N) where
  carrier := {f | m ∈ f.val.dom}
  mem_gt := fun f => h f m

/-- The cofinal set of finite equivalences with a given element in their
    codomain. -/
def definedAtRight
    (h : L.IsExtensionPair N M) (n : N) : Order.Cofinal (FGEquiv L M N) where
  carrier := {f | n ∈ f.val.cod}
  mem_gt := fun f => h.cod f n
```

This is a useful reformulation since there was already a result about cofinal sets in mathlib that will be quite handy for the proof [Wä]:

```
-- Code in Order/Ideal, written by David Warn
/-- Given a starting point, and a countable family of cofinal sets,
  this is an increasing sequence that intersects each cofinal set. -/
noncomputable def sequenceOfCofinals : ℕ → P := ...
```

In our case, intersecting a cofinal set of the form `definedAtLeft _ m` would mean that m is in the domain, and intersecting a cofinal set of the form `definedAtRight _ n` would mean that n is in the codomain. And since M and N are countably generated, we only need to intersect countably many such sets, therefore we can apply this result. We have everything to prove the back-and-forth:

Proof of Theorem 2.2

```
-- Code in ModelTheory/PartialEquiv [AKWa]
/-- For a countably generated structure `M` and a structure `N`, if any
    partial equivalence between finitely generated substructures can be
    extended to any element in the domain, then there exists an embedding of
    `M` in `N`. -/
theorem embedding_from_cg (M_cg : Structure.CG L M) (g : L.FGEquiv M N)
```

```
  (H : L.IsExtensionPair M N) :
  ∃ f : M ↪[L] N, g ≤ f.toPartialEquiv := by

-- We get a countable set `X` which generates `M`.
rcases M_cg with ⟨X, _, X_gen⟩

-- Here we get that `X` is `Encodable`, which is the constructive version of
  being `Countable`. However since we're working in classical logic, it is
  equivalent to `Countable` and we get it for free. We need it because
  `sequenceOfCofinals` asks for it.
have _ : Encodable (↑X : Type _) := @Encodable.ofCountable _ (by simpa only
  [countable_coe_iff])

-- `D` associates each element `x` of `X` with the cofinal set of FGEequivs
  with `x` in their domains.
let D : X → Order.Cofinal (FGEquiv L M N) := fun x ↦ H.definedAtLeft x

-- Using `sequenceOfCofinals`, we define an increasing sequence `S` of
  partial equivalences that will intersect with all `D x`.
let S : ℕ →o M ≃ₚ[L] N :=
  ⟨Subtype.val ∘ (Order.sequenceOfCofinals g D),
    (Subtype.mono_coe _).comp (Order.sequenceOfCofinals.monotone _ _)⟩

-- `F` is the limit of `S`, and should have `M` as its domain, as we will
  prove next.
let F := DirectLimit.partialEquivLimit S

have _ : X ⊆ F.dom := by
  intro x hx
  have := Order.sequenceOfCofinals.encode_mem g D ⟨x, hx⟩
  exact dom_le_dom
    (le_partialEquivLimit S (Encodable.encode (⟨x, hx⟩ : X) + 1)) this

-- This says that indeed the domain of `F` is `M`.
have isTop : F.dom = ⊤ := by rwa [← top_le_iff, ← X_gen,
  Substructure.closure_le]

-- From a partial equivalence whose domain is the whole structure, we can
  get an embedding.
exact ⟨toEmbeddingOfEqTop isTop,

-- We still need to prove that this equivalence extends `g`. We conclude
  from the lemma that the direct limit of a system of partial equivalences
  extends any partial equivalence in the system, and `g` is the starting
  point of the system.
      by convert (le_partialEquivLimit S 0); apply
  Embedding.toPartialEquiv_toEmbedding⟩
```

The proof of the back-and-forth is mostly the same, except that we have to verify

properties for the codomain too.

Proof of Theorem 2.1

```
-- Code in ModelTheory/PartialEquiv [AKWa]
/-- For two countably generated structure ‘M‘ and ‘N‘, if any PartialEquiv
between finitely generated substructures can be extended to any element in the
    domain and to
any element in the codomain, then there exists an equivalence between ‘M‘ and
    ‘N‘. -/
theorem equiv_between_cg (M_cg : Structure.CG L M) (N_cg : Structure.CG L N)
    (g : L.FGEquiv M N)
    (ext_dom : L.IsExtensionPair M N)
    (ext_cod : L.IsExtensionPair N M) :
    ∃ f : M ≃[L] N, g ≤ f.toEmbedding.toPartialEquiv := by

  -- We get a countable set ‘X‘ generating ‘M‘.
  rcases M_cg with ⟨X, X_count, X_gen⟩
  -- We get a countable set ‘Y‘ generating ‘N‘
  rcases N_cg with ⟨Y, Y_count, Y_gen⟩

  have _ : Encodable (↑X : Type _) := @Encodable.ofCountable _ (by simpa only
    [countable_coe_iff])
  have _ : Encodable (↑Y : Type _) := @Encodable.ofCountable _ (by simpa only
    [countable_coe_iff])

  -- ‘D‘ has domain the disjoint union of ‘X‘ and ‘Y‘, and associates to each
    element the corresponding cofinal set.
  let D : Sum X Y → Order.Cofinal (FGEquiv L M N) := fun p ↦
    Sum.recOn p (fun x ↦ ext_dom.definedAtLeft x) (fun y ↦
    ext_cod.definedAtRight y)

  -- ‘S‘ is an increasing sequence intersecting all cofinal sets ‘D x‘ and ‘D
    y‘.
  let S : ℕ →o M ≃ₚ[L] N :=
    ⟨Subtype.val ∘ (Order.sequenceOfCofinals g D),
      (Subtype.mono_coe _).comp (Order.sequenceOfCofinals.monotone _ _)⟩

  -- ‘F‘ is the limit of ‘S‘, it should have ‘M‘ as domain and ‘N‘ as
    codomain, as we will prove next.
  let F := @DirectLimit.partialEquivLimit L M N _ _ ℕ _ _ _ S
  have _ : X ⊆ F.dom := by
    intro x hx
    have := Order.sequenceOfCofinals.encode_mem g D (Sum.inl ⟨x, hx⟩)
    exact dom_le_dom
      (le_partialEquivLimit S (Encodable.encode (Sum.inl (⟨x, hx⟩ : X)) + 1))
    this
  have _ : Y ⊆ F.cod := by
    intro y hy
    have := Order.sequenceOfCofinals.encode_mem g D (Sum.inr ⟨y, hy⟩)
```

```
    exact cod_le_cod
      (le_partialEquivLimit S (Encodable.encode (Sum.inr (⟨y, hy⟩ : Y)) + 1))
    this

  -- This says that the domain of `F` is all of `M`.
  have dom_top : F.dom = ⊤ := by rwa [← top_le_iff, ← X_gen,
    Substructure.closure_le]
  -- This says that the codomain of `F` is all of `N`.
  have cod_top : F.cod = ⊤ := by rwa [← top_le_iff, ← Y_gen,
    Substructure.closure_le]

  -- From a partial equivalence whose domain and codomain are the whole
    structures, we can get an equivalence.
  refine ⟨toEquivOfEqTop dom_top cod_top, ?_⟩

  -- We still need to prove that this equivalence extends `g`. We conclude
    from the lemma that the direct limit of a system of partial equivalences
    extends any partial equivalence in the system, and `g` is the starting
    point of the system.
  convert le_partialEquivLimit S 0
  rw [toEquivOfEqTop_toEmbedding]
  apply Embedding.toPartialEquiv_toEmbedding
```

3 Fraïssé Limits

In this section, we will suppose that in the language L, there are only countably many function symbols. Therefore, a structure is countable if and only if it is countably generated.

3.1 Bundled structures

For this part, we will talk about classes of structures having an embedding in some structure, and for that, it will be helpful to always pair each type with the structure it is supposed to have. Remember that until now, we were working with types for which there was also an instance of structure, but here it is more comfortable to have a type corresponding to pairs of one type and one structure instance on it. We write this type as `Bundled.{w} L.Structure`. Here, `w` indicates in which universe we are working, but it will not be important for the rest of the paper. You can understand this type as the type of pairs `(M : Type w) × L.Structure M`. Lean will coerce it automatically to a type when needed, and remember that this type has an instance of structure. We can also coerce a bundled structure M to a type explicitly by writing `↑M`. A class of structures will have the type `Set (Bundled.{w} L.Structure)`.

3.2 Embeddings between equal structures

Multiple times when we will prove the existence of Fraïssé limits, we will have structures that we can prove are equal, but that are not definitionally equal. In Lean, and in other proof assistants as well, we distinguish two elements being definitionally equal and propositionally equal. They are definitionally equal if the internal system of Lean can see that they have the same definition (up to unfolding of some notations). Two elements x and y are propositionally equal if you can prove that they are equal, meaning that you can construct an element of the type `x = y`, and this is a weaker property. So for example, you could be in this situation:

```
variable {M N P Q : Bundled.{w} L.Structure}
variable (h : M = N)
variable (f : P ↪[L] M) (g : N ↪[L] Q)
-- Trying to compose ‘f‘ and ‘g‘:
#check g.comp f

/-
Lean error: application type mismatch
  g.comp f
argument
  f
has type
  ↑P ↪[L] ↑M : Type w
but is expected to have type
  ?m.16673 ↪[L] ↑N : Type (max ?u.16660 w)
-/
```

Although we know that M and N are propositionally equal, they are not definitionally equal, so the type `M ↪[L] Q` is not the same as `N ↪[L] Q` (but we could prove that they are propositionally equal), and Lean won't accept an element of one type when it expects an element of the other type. To bridge this gap, by induction on the equality type we define a function that gives us an embedding between two structures when we know that they are equal and that should behave similarly to the identity function:

Embeddings between equal structures

```
-- Code in ModelTheory/Bundled [Kol24]
variable {M N P : Bundled.{w} L.Structure}
/-- Embedding between equal structures.-/
def ofEq (h : M = N) : M ↪[L] N := by
  cases h
  exact refl L M

-- If the equality is of type ‘M = M‘, ‘ofEq‘ is simply the identity map.
@[simp]
theorem ofEq_refl : ofEq (Eq.refl M) = refl L M := rfl

-- Two ‘ofEq‘ compose well with each other, being another map of the form
    ‘ofEq _‘.
```

```
@[simp]
theorem ofEq_comp (h : M = N) (h' : N = P) :
    (ofEq h').comp (ofEq h) = ofEq (h.trans h') := by
  cases h
  cases h'
  rfl
```

Then, if we are in the same way as before, we can now compose the two maps as follows:

```
variable (h : M = N)
variable (f : P ↪[L] M) (g : N ↪[L] Q)

#check g.comp ((ofEq h).comp f)
-- Lean message: g.comp ((ofEq h).comp f) : ↑P ↪[L] ↑Q
```

3.3 Definition of Fraïssé limits

The definitions of Fraïssé classes and Fraïssé classes were already implemented by Aaron Anderson in Mathlib before this work. Here is how it has been done:

Definition 3.1 (Age of a structure)**.** The age of a structure M is the class of finitely generated structures who embed in M.

In Lean:

Age of a structure

```
-- Code in ModelTheory/Fraisse, written by Anderson [AKWb]
/-- The age of a structure `M` is the class of finitely-generated structures
    that embed into it. -/
def age (M : Type w) [L.Structure M] : Set (Bundled.{w} L.Structure) :=
  {N | Structure.FG L N ∧ Nonempty (N ↪[L] M)}
```

Definition 3.2 (Hereditary property)**.** A class K of finitely-generated structures has the hereditary property if for all $M \in K$, all finitely-generated structures which embed into M are also in K.

Another way of formulating that is that a class has the hereditary property if it contains the age of all structures in it. In Lean:

Hereditary property

```
-- Code in ModelTheory/Fraisse, written by Anderson [AKWb]
variable (K : Set (Bundled.{w} L.Structure))
```

```
/-- A class ‘K‘ has the hereditary property when all finitely-generated
    structures that embed into
  structures in ‘K‘ are also in ‘K‘. -/
def Hereditary : Prop :=
  ∀ M : Bundled.{w} L.Structure, M ∈ K → L.age M ⊆ K
```

Definition 3.3 (Joint embedding property)**.** A class K of structures has the joint embedding property if for any $M, N \in K$, there is a structure $P \in K$ such that both M and N embed into P.

In Lean:

Joint embedding property

```
-- Code in ModelTheory/Fraisse, written by Anderson [AKWb]
/-- A class ‘K‘ has the joint embedding property when for every ‘M‘, ‘N‘ in
    ‘K‘, there is another structure in ‘K‘ into which both ‘M‘ and ‘N‘ embed.
    -/
def JointEmbedding : Prop :=
  DirectedOn (fun M N : Bundled.{w} L.Structure => Nonempty (M ↪[L] N)) K
-- Author: ‘DirectedOn‘ has type ‘DirectedOn.{u} {α : Type u} (r : α → α →
    Prop) (s : Set α) : Prop‘, and returns True if for any ‘a, b‘ in ‘s‘,
    there exists a ‘c‘ in ‘s‘ such that both ‘r a c‘ and ‘r b c‘ equal ‘True‘.
```

Definition 3.4 (Essentially countable)**.** A class K of structures is essentially countable if it contains only countably many structures up to equivalence.

In Lean:

Essentially countable

```
-- Code in ModelTheory/Fraisse, written by Anderson [AKWb]
/-- The equivalence relation on bundled ‘L.Structure‘s indicating that they
    are isomorphic. -/
instance equivSetoid : Setoid (Bundled.{w} L.Structure) where
  r M N := Nonempty (M ≃[L] N)
  iseqv := ...

-- Author: For a function ‘(f : X → Y)‘, and a set ‘(A : set X)‘, we can
    write ‘f '' A‘ for the image of ‘A‘ by ‘f‘.
-- Author: If there is an instance of an equivalence relation on a type ‘X‘,
    then Quotient.mk' is the quotient map, therefore the definition is saying
    that the image of ‘K‘ in the quotient with respect to the equivalence
    relation of being equivalent as L-structures is countable.
def is_essentially_countable : Prop := (Quotient.mk' '' K).Countable
```

The non-empty classes verifying these properties are exactly the classes corresponding to ages of countable structures.

Proposition 3.5. Let K be a non-empty class of finitely generated structures. Then K has the hereditary and joint embedding property and is essentially countable, if and only if there exists a countable structure M such that $K = \text{Age } M$.

Proof. The 'if' part is clear, so we will show the 'only if' part. Suppose that K has the hereditary and joint embedding property, and is essentially countable. Let $M_1, M_2, \dots$ be a sequence of structures in K such that any structure in K is equivalent to some M_i. Then, we can define a sequence

$$M_1 =: N_1 \hookrightarrow N_2 \hookrightarrow \dots$$

of structures using the joint property, by taking N_{n+1} a structure in K in which both M_{n+1} and N_n embed. We define M as the direct limit of this sequence, and $f_i : N_i \hookrightarrow M$ the canonical embeddings. Any finite set $A \subseteq M$ as a preimage $A' \subseteq N_i$ for some integer i, and the map of closure A' through f_i is equal to closure A, therefore any finitely generated substructure of M has a preimage in some component, and we get

$$\text{Age } M = \bigcup_{i \in \mathbb{N}} \text{Age } N_i = K$$

□

The result we use at the end of the proof can be generalized: the age of a direct limit is the union of the age of each component.

Lemma 3.6 (Age of a direct limit)**.** For G a directed system indexed by ι,

$$\text{Age}\,(\lim\ G) = \bigcup_{i \in \iota} \text{Age } G_i$$

This lemma was already present in Mathlib and we will need it to prove the existence of Fraïssé limits:

Age of a direct limit

```
-- Code written by Anderson, in ModelTheory/Fraisse [AKWb]
/-- The age of a direct limit of structures is the union of the ages of the
    structures. -/
theorem age_directLimit {ι : Type w} [Preorder ι] [IsDirected ι (· ≤ ·)]
    [Nonempty ι]
    (G : ι → Type max w w') [∀ i, L.Structure (G i)] (f : ∀ i j, i ≤ j → G i
    ↪[L] G j)
    [DirectedSystem G fun i j h => f i j h] : L.age (DirectLimit G f) = ⋃ i : ι
    , L.age (G i) := ...
```

Definition 3.7 (Amalgamation property)**.** A class K of structures has the amalgamation property if for any $M, N, P \in K$, $f : M \hookrightarrow N$, $g : M \hookrightarrow P$, there is $Q \in K$ and two embeddings $f' : N \hookrightarrow Q$, $g' : P \hookrightarrow Q$, such that

$$f' \circ f = g' \circ g$$

In Lean:

Amalgamation property

```
-- Code in ModelTheory/Fraisse, written by Anderson [AKWb]
/-- A class ‘K‘ has the amalgamation property when for any pair of embeddings
    of a structure ‘M‘ in ‘K‘ into other structures in ‘K‘, those two
    structures can be embedded into a fourth structure in ‘K‘ such that the
    resulting square of embeddings commutes. -/
def Amalgamation : Prop :=
  ∀ (M N P : Bundled.{w} L.Structure) (MN : M ↪[L] N) (MP : M ↪[L] P),
    M ∈ K → N ∈ K → P ∈ K → ∃ (Q : Bundled.{w} L.Structure) (NQ : N ↪[L]
    Q) (PQ : P ↪[L] Q),
      Q ∈ K ∧ NQ.comp MN = PQ.comp MP
```

Definition 3.8. A structure M is ultrahomogeneous if any equivalence f between finitely generated substructures of M can be extended to an automorphism of M.

We could also formulate this property by saying that for any substructure S of M, and any embedding $f : S \hookrightarrow M$, there exists an equivalence $F : M \simeq M$ such that

$$F \circ \mathrm{incl}_S = f$$

with $\mathrm{incl}_S : S \hookrightarrow M$ being simply the inclusion, and this is how it was written in Mathlib:

ultrahomogeneity

```
-- Code in ModelTheory/Fraisse, written by Anderson [AKWb]
/-- A structure ‘M‘ is ultrahomogeneous if every embedding of a finitely
    generated substructure into ‘M‘ extends to an automorphism of ‘M‘. -/
def IsUltrahomogeneous : Prop :=
  ∀ (S : L.Substructure M) (_ : S.FG) (f : S ↪[L] M),
    ∃ g : M ≃[L] M, f = g.toEmbedding.comp S.subtype
```

The age of a structure M doesn't always have the amalgamation property, but it does if M is ultrahomogeneous:

Lemma 3.9. If a structure M is ultrahomogeneous, then Age M has the amalga-

mation property.

Proof. Let's suppose we have $f : A \hookrightarrow B$, $g : A \hookrightarrow C$ for some $A, B, C \in \text{Age } M$. We can see B and C as finitely generated substructures of M, and $g \circ f^{-1}$ as an equivalence between finitely generated substructures in B and C. Using the ultrahomogenous property, we can get an automorphism $F : M \simeq M$ extending $g \circ f^{-1}$. We define

$$D := \text{closure}\,(F(B) \cup C)$$

and defining $f' : B \hookrightarrow D$ as the restriction of F to B, and $g' : C \hookrightarrow D$ simply as the inclusion, since f' restricted to $f(A)$ is equal to F restricted to $f(A)$ which is equal to $g \circ f^{-1}$, we indeed get

$$f' \circ f = (g \circ f^{-1}) \circ f = g = g' \circ g$$

□

This is not an if and only if: for example, $\mathbb{N}$ as a linear order is not ultrahomogeneous, since the partial equivalence sending 2 to 1 cannot be extended to an automorphism, but its age contains all finite linear orders and has the amalgamation property. However, there is an ultrahomogeneous linear order with the same age: $\mathbb{Q}$, which is the Fraïssé limit of the class of finite linear orders, as we will see.

Definition 3.10 (Fraïssé class)**.** A non-empty class K of finitely generated structures is Fraïssé if K has the hereditary, joint embedding, amalgamation properties and is essentially countable.

Definition 3.11 (Fraïssé limit)**.** For a class K of structures, a structure M is a Fraïssé limit of K if $K = \text{Age } M$ and M is countable and ultrahomogeneous.

In Lean:

Fraïssé class and Fraïssé limit

```
-- Code in ModelTheory/Fraisse, written by Anderson [AKWb]
/-- A Fraisse class is a nonempty, isomorphism-invariant, essentially
    countable class of structures satisfying the hereditary, joint embedding,
    and amalgamation properties. -/
class IsFraisse : Prop where
  is_nonempty : K.Nonempty
  FG : ∀ M : Bundled.{w} L.Structure, M ∈ K → Structure.FG L M
  is_essentially_countable : (Quotient.mk' '' K).Countable
  hereditary : Hereditary K
  jointEmbedding : JointEmbedding K
  amalgamation : Amalgamation K
```

```
/-- A structure ‘M‘ is a Fraisse limit for a class ‘K‘ if it is countable,
    ultrahomogeneous, and has age ‘K‘. -/
structure IsFraisseLimit [Countable M] : Prop where
  ultrahomogeneous : IsUltrahomogeneous L M
  age : L.age M = K
```

From Proposition 3.5 and lemma 3.9, we know that the age of a countable ultrahomogeneous structure is a Fraïssé class. The rest of this section will be devoted to prove that any Fraïssé class has a Fraïssé limit, and that this Fraïssé limit is unique up to equivalence.

3.4 Cardinality of FGEquiv

During the recursive construction of the directed system whose limit will be the Fraïssé limit, we will extend a partial equivalence at each step, therefore we need to make sure that there are not too many of them:

Lemma 3.12. Let M be a countable structure. Then there are only countably many equivalences between finitely generated substructures of M.

Proof. M has only countably many finitely generated substructures, and for each finitely generated substructure A, there are only countably many homomorphisms from A to M, since each map is uniquely characterized by its restriction on the finite generating set. □

So we need two intermediate results to prove this in Lean: that a countable structure has countably many finitely generated substructures, and that there are only countably many homomorphisms between a finitely generated structure and a countable structure.

Countably many finitely generated substructures

```
-- Code in ModelTheory/FinitelyGenerated [Kol24]
theorem Substructure.countable_fg_substructures_of_countable [Countable M] :
    Countable { S : L.Substructure M // S.FG } := by
  -- We define a function ‘g‘ sending each finitely generated substructure to
    a finite set generating it.
  let g : { S : L.Substructure M // S.FG } → Finset M :=
    fun S ↦ Exists.choose S.prop
  -- We show that ‘g‘ is injective.
  have g_inj : Function.Injective g := by
    intro S S' h
    -- We have ‘g S = g S'‘, and we need to show ‘S = S'‘. The next line
    reduces the goal to having only to prove that they are equal as
    substructures, not as finitely generate substructures.
    apply Subtype.eq
    -- We replace ‘S‘ and ‘S'‘ by the closure of ‘g S‘ and the closure of ‘g
    S'‘, and we conclude from the equality ‘g S = g S'‘.
```

```
    rw [(Exists.choose_spec S.prop).symm, (Exists.choose_spec S'.prop).symm]
    exact congr_arg ((closure L) ∘ Finset.toSet) h
  -- It is already proven in Mathlib that a countable type has only countably
    many finite subsets, and we gave an injective map to the type of finite
    subsets, therefore we can conclude that the type of finitely generated
    substructures is countable.
  exact Function.Embedding.countable ⟨g, g_inj⟩
```

Countably many homomorphisms and embeddings

```
-- Code in ModelTheory/FinitelyGenerated [Kol24]
theorem FG.countable_hom (N : Type*) [L.Structure N] [Countable N] (h : FG L
    M) :
    Countable (M →[L] N) := by
  -- We get a finite set ‘S‘ generating ‘M‘.
  let ⟨S, finite_S, closure_S⟩ := fg_iff.1 h
  -- We define a function ‘g‘ sending each homomorphism to its restriction to
    ‘S‘.
  let g : (M →[L] N) → (S → N) :=
    fun f ↦ f ∘ (↑)
  -- We use the lemma ‘Hom.eq_of_eqOn_dense‘ saying that two homomorphisms
    that are equal on a generating set must be equal, to conclude that ‘g‘ is
    injective.
  have g_inj : Function.Injective g := by
    intro f f' h
    apply Hom.eq_of_eqOn_dense closure_S
    intro x x_in_S
    exact congr_fun h ⟨x, x_in_S⟩
  -- Lean already knows that ‘S‘ as a set is finite, but we need to tell
    explicitely that ‘S‘ as a type is finite.
  have : Finite ↑S := (S.finite_coe_iff).2 finite_S
  -- The fact that there are only countably many functions between a finite
    type and a countable type is already present in Mathlib, so we can
    conclude that the type ‘M →[L] N‘ is countable too.
  exact Function.Embedding.countable ⟨g, g_inj⟩
```

And finally, we can prove lemma 3.12 in Lean:

```
-- Code in ModelTheory/FinitelyGenerated [Kol24]
theorem countable_self_fgequiv_of_countable [Countable M] :
    Countable (L.FGEquiv M M) := by
  -- We define a function ‘g‘ associating to each ‘FGEquiv‘ ‘f‘ the pair
    consisting of its domain and the underlying equivalence composed with the
    inclusion in ‘M‘.
  let g : L.FGEquiv M M →
      Σ U : { S : L.Substructure M // S.FG }, U.val →[L] M :=
    fun f ↦ ⟨⟨f.val.dom, f.prop⟩, (subtype _).toHom.comp f.val.toEquiv.toHom⟩

  -- We show that ‘g‘ is injective.
  have g_inj : Function.Injective g := by
```

```
    intro f f' h
    -- We have ‘g f = g f'‘, and we want to conclude that ‘f = f'‘. The next
    line reduces the goal to having to show that ‘f‘ and ‘f'‘ are equal as
    partial equivalences instead of as ‘FGEquiv‘.
    apply Subtype.eq
    -- Since ‘g f = g f'‘, we can deduce ‘f.dom = f'.dom‘, and we want to ask
    Lean to replace every instance of ‘f.dom‘ by ‘f'.dom‘. However, for Lean
    to be able to do that, we first need to decompose ‘f‘ to a combination of
    its subparts: two substructures, an equivalence between them, and the fact
    that its domain is finitely generated.
    let ⟨⟨dom_f, cod_f, equiv_f⟩, f_fin⟩ := f
    cases congr_arg (·.1) h
    -- Two partial equivalences are equal if they have equal domains, and if
    the image of any ‘x‘ in their domain by the composition of the inclusion
    in the whole structure with themselves is the same. We use this result,
    and the fact that ‘g f = g f'‘, to conclude.
    apply PartialEquiv.ext (by rfl)
    simp only [g, Sigma.mk.inj_iff, heq_eq_eq, true_and] at h
    exact fun x hx ↦ congr_fun (congr_arg (↑) h) ⟨x, hx⟩

  -- We need to tell explicitely to Lean that for any element of the subtype
    of substructures that are finitely generated, the underlying substructure
    is finitely generated.
  have : ∀ U : { S : L.Substructure M // S.FG }, Structure.FG L U.val :=
    fun U ↦ (U.val.fg_iff_structure_fg.1 U.prop)
  -- We can conclude. Lean knows that if I have a countable type ‘T‘, and a
    function ‘F‘ associating to each ‘t : T‘ a countable structure, then the
    type ‘Σ t : T, F t‘ of pairs of an element ‘t : T‘ and an element ‘a : F
    t‘ is countable. It also knows that there are only countably many finitely
    generated substructures, and for finitely generated substructure,
    countably many homomorphisms to ‘M‘, and is capable of combining
    everything automatically to conclude that
  -- ‘Σ U : { S : L.Substructure M // S.FG }, U.val →[L] M‘
  -- is countable. This is the power of type class inference in Lean: it is
    capable of chaining instances automatically.
  exact Function.Embedding.countable ⟨g, g_inj⟩
```

3.5 Fraïssé limits exist

Countable ultrahomogeneous structures have a nice characterization:

Lemma 3.13. A countable structure M is ultrahomogeneous if and only if it forms an extension pair with itself.

Proof. If M is ultrahomogeneous, any equivalence between finitely generated substructures can be extended to an automorphism, so it can be extended to contain any element in its domain. We get the other direction from Theorem 2.1. □

We proved this result in Lean in the following way:

```
-- Code in ModelTheory/Fraisse [Kol24]
/-- A countably generated structure is ultrahomogeneous if and only if any
    equivalence between finitely generated substructures can be extended to
    any element in the domain.-/
theorem isUltrahomogeneous_iff_IsExtensionPair (M_CG : CG L M) :
    L.IsUltrahomogeneous M ↔
    L.IsExtensionPair M M := by
  constructor

  -- Forward implication
  · intro M_homog ⟨f, f_FG⟩ m
    -- We have a partial equivalence `f` between finitely generated
    substructures, and an element `m : M`. We need to define an extension of
    `f` which contains `m` in its domain.
    -- We define `S` as the substructure generated by the domain of `f` and
    `m`.
    let S := f.dom ⊔ closure L {m}
    have dom_le_S : f.dom ≤ S := le_sup_left

    -- From ultrahomogeneity, we get an automorphism `F` extending `f`.
    let ⟨F, hF⟩ := M_homog _ f_FG f.toEmbedding

    -- `f'` is the restriction of `F` to `S`. This is the one we will use, and
    we need to convince Lean that it has the right properties.
    let f' := F.toEmbedding.toPartialEquiv.domRestrict (A := S) (fun {|x|} a ↦
    trivial)
    use ⟨f', FG.sup f_FG (fg_closure_singleton m)⟩

    -- Its domain is `S` which contains `m`, and its domain contains the
    domain of `f`, by the properties of the supremum.
    refine ⟨le_sup_right (b := closure L {m}) (subset_closure (mem_singleton
    m)), ⟨dom_le_S, ?_⟩⟩

    -- We still need to show that `f'` extends `f`. For this, it is sufficient
    to show that elements in the domain of `f` have the the same image by both
    maps.
    ext
    simp only [Embedding.comp_apply, Equiv.coe_toEmbedding, coeSubtype, ←
    f.toEmbedding_apply, hF]
    rfl

  -- Backward implication
  · intro h S S_FG f
    -- We have a substructure `S` and an embedding `f : S ↪[L] M`. We need to
    get an automorphism extending `f`.
    -- We apply back-and-forth to get an automorphism `g`
    let ⟨g, ⟨dom_le_dom, eq⟩⟩ :=
      equiv_between_cg M_CG M_CG ⟨⟨S, f.toHom.range, f.equivRange⟩, S_FG⟩ h h
```

```
    use g

    -- The rest is just to convince Lean that ‘g‘ indeed extends ‘f‘.
    simp only [Embedding.subtype_equivRange] at eq
    rw [← eq]
    ext
    rfl
```

The following definition will be quite useful:

Definition 3.14 (Fully extendable)**.** Let f be an equivalence between substructures of M, and $g : M \hookrightarrow N$ an embedding. Then f is fully extendable through g if there exists an equivalence $f' : A \simeq B$ between substructures of N such that f' extends the mapping of f through g and $g(M) \subseteq A$.

In Lean:

Is fully extendable through

```
-- Code in ModelTheory/PartialEquivs [Kol24]
/-- A partial equivalence ‘f‘ between substructures of ‘M‘ is fully extendable
    through an embedding ‘g‘ if there is partial equivalence between
    substructures of the codomain of ‘g‘ which extends the map of ‘f‘ and
    whose domain contains the image of ‘M‘.-/
def is_fully_extendable_through (f : M ≃ₚ[L] M) (g : M ↪[L] N) : Prop :=
  ∃ f', f.map g ≤ f' ∧ g.toHom.range ≤ f'.dom
```

Fraïssé classes have the following nice property:

Lemma 3.15. Let K be a Fraïssé class, $M \in K$, $A, B \subseteq M$ finitely generated substructures, and $f : A \simeq B$, then there exists $N \in K$ and $g : M \hookrightarrow N$ such that f is fully extendable through g.

Proof. Applying the amalgamation property to $incl_A : A \hookrightarrow M$ and $incl_B \circ f : A \hookrightarrow M$, we get a structure $N \in K$ and two embeddings $g, g' : M \hookrightarrow N$ such that

$$g' = g \circ f$$

We define $f' := g' \circ g^{-1}$. Then $g(M)$ is in the domain of f', and for any $a \in A$, we have

$$f' \circ g(a) = g'(a) = g \circ f(a)$$

therefore f' extends the mapping of f through g. □

We write this lemma in Lean as:

```
-- Code in ModelTheory/Fraisse [Kol24]
theorem can_extend_FGEquiv (S : K) (f : S ≃ₚ[L] S) (f_fg : f.dom.FG) :
    ∃ T : K, ∃ incl : S ↪[L] T, f.is_fully_extendable_through incl := ...
```

We can now prove the existence of Fraïssé limits:

Theorem 3.16 (Existence of Fraïssé limits)**.** Any Fraïssé class has a Fraïssé limit.

Proof. Let K be a Fraïssé class. Let $M_0, M_1, \ldots$ be an essentially surjective sequence of structures in K, meaning that any structure $N \in K$ is equivalent to some M_i. By lemma 3.12, for any $M : K$, there are only countably many equivalences between finitely generated substructures of M, so we can fix a sequence $f(M,0), f(M,1), \ldots$ of all the equivalences between finitely generated substructures in M. We also fix two maps $P, Q : \mathbb{N} \to \mathbb{N}$ with the property that $n \mapsto (P(n), Q(n))$ is surjective on $\mathbb{N}^2$ and $P(n), Q(n) \leq n$. We then define a sequence of structures and embeddings $N_0 \hookrightarrow N_1 \hookrightarrow N_2 \hookrightarrow \ldots$ recursively in the following way: We initialize with $N_0 := M_0$. For the inductive step, we first get f_i' a partial equivalence on N_i by mapping $f(N_{P(i)}, Q(i))$ through the sequence of embeddings

$$N_{P(i)} \hookrightarrow N_{P(i)+1} \hookrightarrow \ldots \hookrightarrow N_i.$$

We get N_{i+1} and $g_i : N_i \hookrightarrow N_{i+1}$ by first applying the lemma 3.15 on f_i' and then using the join property with M_{i+1}. Note that it means that f_i' is fully extendable by g_i, and since f_i' is the mapping of $f(N_{P(i)}, Q(i))$, this partial equivalence is fully extendable by the composition of embeddings

$$N_{P(i)} \hookrightarrow N_{P(i)+1} \hookrightarrow \ldots \hookrightarrow N_{i+1}.$$

We claim that the direct limit N of this sequence is a Fraïssé limit. Since M_i has an embedding into the component N_i, it has an embedding into N, so $K \subseteq \operatorname{Age} N$, and $\operatorname{Age} N = \bigcup_i \operatorname{Age} N_i \subseteq K$, so $K = \operatorname{Age} N$, and N is the direct limit of countably many countable structures, so it is countable. We still need to show that it forms an extension pair with itself. Let $f : A \simeq B$ be an equivalence between finitely generated substructures, and $m \in M$, and $S \subseteq A$ and $T \subseteq B$ finite generating sets. Then there is some $i \in \mathbb{N}$ such that $S \cup T \cup \{m\}$ has a preimage in N_i, therefore A and B also have a preimage in N_i, and thus f must also have a preimage $f(N_i, j)$ in N_i. Let $k \in \mathbb{N}$ be a natural number such that $P(k) = i$ and $Q(k) = j$. We used lemma 3.15 to define N_{k+1}, so we know that there is a partial equivalence in N_{k+1} whose domain contains the image of N_i and which extends the mapping of $f(N_i, j)$. In particular, its domain contains the preimage of m and its mapping to N is an extension of f, so we can conclude that N is ultrahomogeneous. □

In a set theoretical setting, instead of constructing a sequence of embeddings $N_0 \hookrightarrow N_1 \hookrightarrow \ldots$, we generally directly suppose that it is an inclusion of sets $N_0 \subseteq N_1 \subseteq \ldots$, which makes the proof cleaner, since there is no need to talk about mappings and

preimages, the direct limit being simply the union. But when working with types, there is no notion of "type inclusion" akin to the notion of "set inclusion". Another thing that makes this proof less straightforward to formalize in Lean is that to define the induction step N_{i+1}, we need to have access to all previous steps N_j and maps between them. Writing the type of this construction is awkward:

```
-- This would not work since you don't have access to the maps between them.
def system : ℕ → K := ...

-- Another idea would be to directly define the type as a sequence of
    embeddings between elements of `K`, but without the information that these
    maps are composable, it is impossible to write the definition of the
    inductive step.
def system : ℕ → (A : K) × (B : K) × (A ↪[L] B) := ...

-- Lean allows mutual recursive definitions, and you can write multiple
    definitions depending on each other, but these definitions cannot depend
    on each other in their types, so the following is not accepted by Lean:
mutual
  def system : ℕ → K := ...

  def system_maps : (n : ℕ) → (system n ↪[L] system (n+1)) := ...
end
```

In the end, we used the following type that contains all the information needed, before being able to write cleanly the sequence of structures:

```
-- Code in ModelTheory/Fraisse [Kol24]
/-- recursive construction containing all the information to define `system`
    and `maps_system`. The left handside of the image gives a sequence of
    structures whose limit will be the Fraisse limit. The right handside
    stores all the previous structures in the sequence, and maps from them to
    the new structure.-/
noncomputable def init_system : (n : ℕ) → (A : K) × (ℕ → (B : K) × (B ↪[L]
    A))
```

We want `init_system` to have the following properties:

- `(init_system n).1` is N_n.
- for $m < n$, `(init_system n).2 m` is the pair composed of N_m and the composition of maps $f_{n-1} \circ \dots f_m : N_m \hookrightarrow N_n$.
- `(init_system n).2 n` is (N_n, id_{N_n}).

We don't care what `(init_system n).2 m` is for $m > n$, since we won't use it during the rest of the proof, but in this case it will simply be the pair (N_n, id_{N_n}). To write the definition, we use the three following functions:

extend_and_join

```
-- Code in ModelTheory/Fraisse [Kol24]
/-- Extends a `FGEquiv`, then joins another structure.-/
noncomputable def extend_and_join (B : K) {A : K} {f : A ≃ₚ[L] A} (f_fg :
    f.dom.FG) :
    (C : K) × (A ↪[L] C) := ...

/-- An essentially surjective sequence of L.structures in a Fraisse class. -/
noncomputable def ess_surj_sequence (n : ℕ) : K := ...

/-- A surjective sequence of `FGEquiv`.-/
noncomputable def sequence_FGEquiv (A : K) (n : ℕ) : FGEquiv L A A := ...
```

And the surjective function $\mathbb{N} \to \mathbb{N}^2$ we will use is this function that was already present in Mathlib and is in fact bijective:

pair and unpair

```
-- Code in Data/Nat/Unpair, written by Leonardo de Moura and Mario Carneiro
    [dMC]
/-- Unpairing function for the natural numbers. -/
def unpair (n : ℕ) : ℕ × ℕ :=
  let s := sqrt n
  if n - s * s < s then (n - s * s, s) else (s, n - s * s - s)

def pair (a b : ℕ) : ℕ :=
  if a < b then b * b + a else a * a + a + b

theorem pair_unpair (n : ℕ) : pair (unpair n).1 (unpair n).2 = n := ...

theorem unpair_pair (a b : ℕ) : unpair (pair a b) = (a, b) := ...

theorem unpair_left_le : ∀ n : ℕ, (unpair n).1 ≤ n := ...

theorem unpair_right_le (n : ℕ) : (unpair n).2 ≤ n := ...
```

We can finally write this complicated definition:

init_system and system

```
-- Code in ModelTheory/Fraisse [Kol24]
/-- recursive construction containing all the information to define `system`
    and `maps_system`.
The left handside of the image gives a sequence of structures whose limit will
    be the Fraisse limit.
The right handside stores all the previous structures in the sequence, and
    maps from them to the new structure.-/
noncomputable def init_system : (n : ℕ) → (A : K) × (ℕ → (B : K) × (B ↪[L]
    A))
  -- Initial value
  | 0 => ⟨ess_surj_sequence K_fraisse 0,
    fun _ => ⟨_, Embedding.refl L _⟩⟩
```

```
  -- Inductive step
  | n + 1 => by
    -- ‘p‘ is P(n) in the proof
    let p := (Nat.unpair n).1
    -- ‘q‘ is Q(n)
    let q := (Nat.unpair n).2
    let N_n := (init_system n).1
    let S_n := (init_system n).2
    let N_p := (S_n p).1
    -- ‘N_p_to_N_n‘ is the composition g_{n-1} ∘ ... ∘ g_p.
    let N_p_to_N_n : N_p ↪[L] N_n := (S_n p).2
    -- ‘f‘ is f(N_p, q) in the proof
    let ⟨f, f_fg⟩ := sequence_FGEquiv K_fraisse N_p q
    -- ‘N‘ is N_{n+1}, and ‘N_n_to_N‘ is g_n
    let ⟨N, N_n_to_N⟩ := extend_and_join K_fraisse
      (A := N_n) (B := ess_surj_sequence K_fraisse (n+1))
      (f := f.map N_p_to_N_n) (PartialEquiv.map_dom N_p_to_N_n f ▷ FG.map _ f_fg)
    exact ⟨N, fun m ↦ if m ≤ n then ⟨(S_n m).1, N_n_to_N.comp ((S_n m).2)⟩
                          else ⟨_, Embedding.refl L _⟩⟩

/-- Sequence of structures whose direct limit is the Fraisse limit.-/
noncomputable def system (n : ℕ) : K := (init_system K_fraisse n).1
```

As we have defined everything, we have the following property:

```
-- Code in ModelTheory/Fraisse [Kol24]
-- The property that the right side indeed serves as memory for the structures
    that appeared in the left side of ‘init_system‘
theorem system_eq {n : ℕ} {m : ℕ} (h : m ≤ n) :
    system K_fraisse m = ((init_system K_fraisse n).2 m).1 := ...

-- Same as the previous lemma, but the equality is as ‘L.Structure‘, not as
    elements of ‘K‘.
theorem system_eq_as_structures {n : ℕ} {m : ℕ} (h : m ≤ n) :
    (system K_fraisse m : Bundled.{w} L.Structure) = ((init_system K_fraisse
    n).2 m).1 := ...
```

But this is only a propositional equality, not a definitional equality, which means that `((init_system K_fraisse n).2 m).2` has type

```
init_system K_fraisse n).2 m).2 : ((init_system K_fraisse n).2 m).1 ↪[L]
    system K_fraisse n
```

and cannot directly be used directly to define the map `system K_fraisse m ↪[L] system K_fraisse n`. We simply composed it with the embedding we get from the equality.

maps_system

```
-- Code in ModelTheory/Fraisse [Kol24]
```

```
/-- Maps to have a directed system on the sequence given by `system`. -/
noncomputable def maps_system {m n : ℕ} (h : m ≤ n):
    system K_fraisse m ↪[L] system K_fraisse n :=
  ((init_system K_fraisse n).2 m).2.comp (Embedding.ofEq
    (system_eq_as_structures K_fraisse h))
```

Another problem we have is that the definition of `init_system` contains an "if-then-else", and we will again have problems of types that are equal propositionally but not definitionally and for which Lean needs help to simplify things:

```
variable (f : A ↪[L] C) (g : B ↪[L] C)
variable (h : P = True)
-- The following is ill-typed
#check (if P then (⟨A, f⟩ : (B : K) × (B ↪[L] C)) else ⟨B, g⟩).2 = f
/-
type mismatch
  f
has type
  ↑↑A ↪[L] ↑↑C : Type w
but is expected to have type
  ↑↑(if P then ⟨A, f⟩ else ⟨B, g⟩).fst ↪[L] ↑↑C : Type w
-/
```

So to simplify this expression, we need again to compose it with an embedding we get from the equality `(if P then (⟨A, f⟩ : (B : K) × (B ↪[L] C)) else ⟨B, g⟩).1 = A`:

```
-- Code in ModelTheory/Fraisse [Kol24]
theorem if_then_else_left_struct {A B C: K} {P : Prop} [Decidable P] (f : A ↪
    [L] C)
    (g : B ↪[L] C) (h : P = True) :
      (if P then (⟨A, f⟩ : (B : K) × (B ↪[L] C)) else ⟨B, g⟩).1 = A := ...

theorem if_then_else_left {A B C: K} {P : Prop} [Decidable P] (f : A ↪[L] C)
    (g : B ↪[L] C) (h : P = True) :
      (if P then (⟨A, f⟩ : (B : K) × (B ↪[L] C)) else ⟨B, g⟩).2 =
        f.comp (Embedding.ofEq (congr_arg _ (if_then_else_left_struct f g h)))
    := ...
```

The auxiliary result `if_then_else_left` may seem to transform a simple formula into a more complicated formula, but in fact the new form `f.comp (Embedding.ofEq _)` will be easier for Lean to deal with and to see that it is essentially equal to f. We have of course a similar result for the right hand-side:

```
-- Code in ModelTheory/Fraisse [Kol24]
theorem if_then_else_right_struct {A B C: K} {P : Prop} [Decidable P] (f : A ↪
    [L] C)
    (g : B ↪[L] C) (h : P = False) :
      (if P then (⟨A, f⟩ : (B : K) × (B ↪[L] C)) else ⟨B, g⟩).1 = B := ...

theorem if_then_else_right {A B C: K} {P : Prop} [Decidable P] (f : A ↪[L] C)
```

```
    (g : B ↪[L] C) (h : P = False) :
      (if P then (⟨A, f⟩ : (B : K) × (B ↪[L] C)) else ⟨B, g⟩).2 =
        g.comp (Embedding.ofEq (congr_arg _ (if_then_else_right_struct f g
    h))) := ...
```

We now have our sequence of structures and we need to prove three properties:

1. The maps given by `maps_system` commute.
2. Any structure in K embeds in `system n` for some n.
3. For any m, any `FGEquiv` f in `system m`, there is some $n > m$ such that the map `system m ↪ system n` extends f.

We get Property 1. by showing that for $m \leq n$ the map `system m ↪[L] system n+1` is equal to the composition `system m ↪ system n ↪ system n+1`, and then doing a simple proof by induction.

Property 1

```
-- Code in ModelTheory/Fraisse [Kol24]
/-- Map between successive structures in `system`.-/
noncomputable def map_step (m : ℕ) : system K_fraisse m ↪[L] system K_fraisse
    (m+1) :=
  maps_system K_fraisse (Nat.le_add_right m 1)

theorem factorize_with_map_step {m n : ℕ} (h : m ≤ n) :
    maps_system K_fraisse (h.trans (Nat.le_add_right n 1)) =
      (map_step K_fraisse n).comp (maps_system K_fraisse h) := ...

theorem transitive_maps_system {m n k : ℕ} (h : m ≤ n) (h' : n ≤ k) :
    (maps_system K_fraisse h').comp (maps_system K_fraisse h) =
      maps_system K_fraisse (h.trans h') := ...
```

Property 2 follows easily from the fact that any structure in K is equivalent to some `ess_surj_sequence n`, and that we join this structure at step n:

Property 2

```
-- Code in ModelTheory/Fraisse [Kol24]
theorem contains_K : ∀ M ∈ K, ∃ n, Nonempty (M ↪[L] system K_fraisse n) := by
  intro A h
  -- We get `n` and an equivalence `g : M ≃[L] ess_surj_sequence n`.
  let ⟨n, ⟨g⟩⟩ := ess_surj_sequence_spec K_fraisse ⟨A, h⟩
  use n
  constructor
  -- We need to find an embedding `M ↪[L] system K_fraisse n`
  -- Since we can compose something with `g`, we can reduce to having to find
    an embedding `ess_surj_sequence n ↪[L] system K_fraisse n`.
  apply Nonempty.map (Embedding.comp · g.toEmbedding)
```

```
  -- We treat two cases: ‘n = 0‘ and ‘n = m + 1‘.
  cases n
  -- For ‘n = 0‘ we can take the identity since the definition of ‘system 0‘
    is ‘ess_surj_sequence 0‘.
  · exact ⟨Embedding.refl ..⟩
  -- For ‘n = m + 1‘, it follows from the property of the join function that
    we have an embedding.
  · simp only [system, init_system]
    exact extend_and_join_spec_2 ..
```

Now we will prove Property 3. The key fact is for that any `FGEquiv` f defined on `system m`, we know that `f = sequence_FGEquiv (system m) n` for some n. In the definition of `init_system`, we see that at step `(Nat.pair m n) + 1`, we extend the mapping of `sequence_FGEquiv ((init_system K_fraisse (Nat.pair m n)).2 m).1 n`, and since `sequence_FGEquiv (system m) n = ((init_system K_fraisse (Nat.pair m n)).2 m).1`, it should also extend f, and the bulk of the proof will be to convince Lean of that.

Property 3

```
-- Code in ModelTheory/Fraisse [Kol24]
/-- The ‘FGEquiv‘ which is extended at step ‘n+1‘ in ‘system‘.-/
noncomputable def FGEquiv_extended (n : ℕ) :
    FGEquiv L ((init_system K_fraisse n).2 (Nat.unpair n).1).1
      ((init_system K_fraisse n).2 (Nat.unpair n).1).1 :=
  sequence_FGEquiv K_fraisse ((init_system K_fraisse n).2 (Nat.unpair n).1).1
    (Nat.unpair n).2

theorem map_step_is_extend_and_join (r : ℕ) :
    map_step K_fraisse r = ((extend_and_join K_fraisse (ess_surj_sequence
    K_fraisse (r+1))
      (f := ((FGEquiv_extended K_fraisse r).1.map
        ((init_system K_fraisse r).2 (Nat.unpair r).1).2))
      (PartialEquiv.map_dom (((init_system K_fraisse r).2
        (Nat.unpair r).1).2) _ ▷ FG.map _ (FGEquiv_extended K_fraisse
    r).2)).2) := ...

theorem all_fgequiv_extend {m : ℕ} (f : L.FGEquiv (system _ m) (system _ m)) :
    ∃ n, ∃ h : m ≤ n, f.val.is_fully_extendable_through (maps_system K_fraisse
    h) := by

  -- First, we reduce to the case ‘f = sequence_FGEquiv (system (Nat.unpair
    r).1) (Nat.unpair r).2‘
  let ⟨n, hn⟩ := sequence_FGEquiv_spec K_fraisse f
  let r := Nat.pair m n
  have h_unpair_m : m = (Nat.unpair r).1 := by simp only [Nat.unpair_pair, r]
  have h_unpair_n : n = (Nat.unpair r).2 := by simp only [Nat.unpair_pair, r]
  -- We would like to replace ‘m‘ by ‘(Nat.unpair r).1‘, and similarly for
    ‘n‘, but Lean does not accept it, because ‘r‘ is defined in function of
    ‘m‘ and ‘n‘, and we would get a circular definition. So we first need Lean
    to forget the value of ‘r‘, so that it becomes a generic natural.
```

```
  clear_value r
  cases h_unpair_m
  cases h_unpair_n
  use r+1
  use (Nat.unpair_left_le r).trans (Nat.le_add_right r 1)

  -- The goal is now to prove that ‘f‘ is extended by the embedding ‘system
    (Nat.unpair r).1 ↪[L] system (r+1)‘.
  -- We want to apply ‘map_step_is_extend_and_join‘ and the property that
    ‘extend_and_join‘ extends a partial equivalence, so we need to replace ‘f‘
    by ‘FGEquiv_extended r‘ in the goal.
  let f' := f.1.map (Embedding.ofEq (system_eq_as_structures K_fraisse
    (Nat.unpair_left_le r)))
  have h_f'_map : f'.map ((init_system _ r).2 (Nat.unpair r).1).2 =
      f.1.map (maps_system _ (Nat.unpair_left_le r)) := by
    apply PartialEquiv.map_map
  cases hn
  -- mapping ‘f‘ through the embedding we get from the equality ‘system_eq_as
    structures‘ gets us ‘FGEquiv_extended r‘.
  have h_f' : f' = FGEquiv_extended K_fraisse r := by
    have H {A B : K} (h : A = B) :
        (sequence_FGEquiv K_fraisse A (Nat.unpair r).2).val.map
          (Embedding.ofEq (congr_arg Subtype.val h)) =
        (sequence_FGEquiv K_fraisse B (Nat.unpair r).2).val := by
      cases h
      simp only [Embedding.ofEq_refl, PartialEquiv.map_refl]
    exact H (system_eq K_fraisse (Nat.unpair_left_le r))

  -- We replace ‘system (Nat.unpair r).1 ↪[L] system (r+1)‘ by the
    composition ‘system (Nat.unpair r).1 ↪[L] system r ↪[L] system (r+1)‘
  rw [← transitive_maps_system K_fraisse (Nat.unpair_left_le r)
    (Nat.le_add_right r 1)]
  -- We apply ‘comp_is_fully_extendable_through‘, which says that if the
    mapping of ‘f‘ through ‘g‘ is extended by ‘h‘, then ‘f‘ is extended by
    ‘h.comp g‘.
  apply PartialEquiv.comp_is_fully_extendable_through
  -- We now have to show that the mapping of ‘f‘ to ‘system r‘ is extended by
    the embedding ‘system r ↪[L] system (r+1)‘. Using ‘h_f'‘ and ‘h_f'_map‘,
    we can reduce the goal to show that the mapping of ‘FGEquiv_extended r‘ is
    extended, which we get from ‘map_step_is_extend_and_join‘.
  rw [← h_f'_map, h_f', ← map_step, map_step_is_extend_and_join]
  apply extend_and_join_spec_1
```

We finally have everything we need to prove that the direct limit of this system is a Fraïssé limit.

Existence of Fraïssé limits

```
-- Code in ModelTheory/Fraisse [Kol24]
/-- A Fraisse class in a language with countably many functions has a Fraisse
```

```
    limit.-/
theorem exists_fraisse_limit (K_fraisse : IsFraisse K) : ∃ M : Bundled.{w}
    L.Structure,
    ∃ _ : Countable M, IsFraisseLimit K M := by
  -- Auxiliary properties to be able to take the direct limit
  let _ (i : ℕ) : L.Structure ((Bundled.α ∘ Subtype.val ∘ system K_fraisse) i)
    :=
    Bundled.str _
  have _ : DirectedSystem (Bundled.α ∘ Subtype.val ∘ system K_fraisse)
      fun _ _ h ↦ ⇑(maps_system K_fraisse h) := by
    constructor
    intro _ _
    simp only [Function.comp_apply, maps_system_self, Embedding.refl_apply]
    intro _ _ _ _ _ _
    simp only [Function.comp_apply, ← Embedding.comp_apply,
    transitive_maps_system]

  -- ‘M‘ is the direct limit of ‘system‘.
  let M := DirectLimit (L := L) (Bundled.α ∘ Subtype.val ∘ system K_fraisse)
    (@maps_system _ _ K_fraisse _)
  use ⟨M, DirectLimit.instStructureDirectLimit ..⟩
  -- ‘M‘ is countable because it is the direct limit of countably many
    countable structures.
  have M_c : Countable M := by
    rw [← Structure.cg_iff_countable (L := L)]
    apply DirectLimit.cg
    simp only [Function.comp_apply, Structure.cg_of_countable, implies_true]
  use M_c
  -- ‘of n‘ is the embedding of ‘system n‘ into ‘M‘.
  let of (n : ℕ) : (system K_fraisse n ↪[L] M) :=
    DirectLimit.of L ℕ (Bundled.α ∘ Subtype.val ∘ system K_fraisse) _ n

  refine ⟨?_, ?_⟩

  -- We show that ‘M‘ is ultrahomogeneous‘, by showing that it forms an
    extension pair with itself and the fact that it is equivalent to
    ultrahomogeneous.
  · rw [isUltrahomogeneous_iff_IsExtensionPair Structure.cg_of_countable]
    intro ⟨f, f_fg⟩ m
    -- We have ‘f‘ a partial equivalence between finitely generated
    substructures of ‘M‘, and ‘m : M‘. The goal is to show that there is a
    partial equivalence extending ‘f‘ and whose domain contains ‘m‘.
    -- ‘A‘ is the substructure of ‘M‘ generated by the domain and codomain of
    ‘f‘ and ‘m‘.
    let A := f.dom ⊔ f.cod ⊔ (closure L {m})
    let A_fg : A.FG := FG.sup (FG.sup f_fg (f.dom_fg_iff_cod_fg.1 f_fg))
    (fg_closure_singleton m)
    -- ‘A’‘ is a substructure of ‘system n‘ whose mapping to ‘M‘ is equal to
    ‘A‘.
```

```
  let ⟨n, A', hA'⟩ := DirectLimit.exists_fg_substructure_in_Sigma A A_fg
  -- So `A` is contained in the image of `system n` in `M`.
  have in_range : f.dom ⊔ f.cod ⊔ (closure L {m}) ≤ (of n).toHom.range := by
    unfold A at hA'
    rw [← hA']
    exact Hom.map_le_range
  -- Therefore we can get `f'` a partial equivalence which is the preimage
  of `f` in `system n`.
  let ⟨f',f'_map⟩ := (PartialEquiv.exists_preimage_map_iff (of n) f).2
    (le_sup_left.trans in_range)
  have f'_fg : f'.dom.FG := by
    apply FG.of_map_embedding (of n)
    rwa [← PartialEquiv.map_dom, f'_map]
  -- We show that `f'` is extended by `of n`.
  have H : f'.is_fully_extendable_through (of n) := by
    -- By Property 3, there is some `m` such that the embedding `system n ↪
  [L] system m` extends `f'`.
    let ⟨m, hnm, f'_extended⟩ := all_fgequiv_extend K_fraisse ⟨f', f'_fg⟩
    unfold of
    -- We can replace `of n` by the composition `system n ↪[L] system m ↪
  [L] M`
    rw [← DirectLimit.of_comp_f (hij := hnm)]
    -- We use a lemma saying that a partial equivalence `f` is extended by
  `g.comp h` if `f` is extended by `h`.
    exact PartialEquiv.is_fully_extendable_through_comp _ _ _ f'_extended
  -- From the fact that `f'` is extended by `of n`, we can get a partial
  equivalence `g'` on `M` extending the mapping of `f'` and whose domain
  contains the image of `system n`.
  let ⟨g', map_f'_le, range_le_g'⟩ := H
  -- And we define `g` by restricting the domain of `g'` to `A`.
  let g := g'.domRestrict (in_range.trans range_le_g')
  -- We show that `g` extends `f` by their definitions and a simple lemma on
  restrictions of partial equivalences.
  have f_le_g : (⟨f, f_fg⟩ : FGEquiv L M M) ≤ ⟨g, A_fg⟩ := by
    rw [Subtype.mk_le_mk]
    apply PartialEquiv.le_domRestrict
    exact le_sup_left.trans le_sup_left
    rw [f'_map] at map_f'_le
    exact map_f'_le
  -- `m` is in the domain of `g`.
  have m_in_dom : m ∈ g.dom := by
    unfold g
    unfold PartialEquiv.domRestrict
    simp only
    rw [← closure_eq f.dom, ← closure_eq f.cod, ← closure_union, ←
  closure_union]
    apply subset_closure
    exact mem_union_right (f.dom ∪ (f.cod : Set M)) rfl
  use ⟨g, A_fg⟩
```

```
-- We show that the age of ‘M‘ is exactly ‘K‘
· -- We use the lemma saying that the age of a direct limit is the union of
  the ages of the components.
  rw [age_directLimit]
  apply Set.ext
  intro S
  -- We have a structure ‘S‘ and we want to show that it is contained in the
  union of ages if and only if it is contained in ‘K‘.
  -- ‘mem_iUnion‘ says that an element is contained in an union if and only
  if it is contained in some component of the union.
  rw [mem_iUnion]
  refine ⟨?_, ?_⟩
  · -- First suppose it is contained in some component. We use the hereditary
  property of ‘K‘.
    rintro ⟨i, S_in_age⟩
    exact K_fraisse.hereditary ((Subtype.val ∘ system K_fraisse) i)
      (by simp only [Function.comp_apply, Subtype.coe_prop]) S_in_age
  · -- Then suppose it is contained in ‘K‘. We use Property 2.
    intro S_in_K
    let ⟨n, ⟨inc_S⟩⟩ := contains_K K_fraisse S S_in_K
    use n
    simp only [age, Function.comp_apply, mem_setOf_eq]
    exact ⟨IsFraisse.FG S S_in_K, ⟨inc_S⟩⟩
```

3.6 Fraïssé limits are unique

The proof that Fraïssé Limits are unique up to equivalence is easy now that we have the back-and-forth method.

Lemma 3.17. Let M be an ultrahomogeneous structure, S a finitely generated structure, T a structure, and three embeddings $f : S \hookrightarrow M$, $g : S \hookrightarrow T$ and $h : T \hookrightarrow M$. Then, there exists an embedding $h' : T \hookrightarrow M$ such that $h' \circ g = f$.

Proof. Let $A := f(S)$ and $A' = h(g(S))$. We have the equivalence $f \circ g^{-1} \circ h^{-1} : A' \simeq A$, and using ultrahomogeneity we can extend it to an automorphism $F : M \simeq M$. We define $h' := F \circ h$, and so

$$h' \circ g = F \circ h \circ g = f \circ g^{-1} \circ h^{-1} \circ h \circ g = f$$

□

In Lean, we do the same, but with a lot of work devoted to prove the sequence of equations that appears at the end of the proof:

Proof of lemma 3.17

```
-- Code in ModelTheory/Fraisse [AKWb]
-- Any embedding from a finitely generated ‘S‘ to an ultrahomogeneous
    structure ‘M‘ can be extended to an embedding from any structure with an
    embedding to ‘M‘. -/
theorem IsUltrahomogeneous.extend_embedding (M_homog : L.IsUltrahomogeneous M)
    {S : Type*}
    [L.Structure S] (S_FG : FG L S) {T : Type*} [L.Structure T] [h : Nonempty
    (T ↪[L] M)]
    (f : S ↪[L] M) (g : S ↪[L] T) :
    ∃ f' : T ↪[L] M, f = f'.comp g := by
  let ⟨r⟩ := h
  let s := r.comp g
  let ⟨t, eq⟩ := M_homog s.toHom.range (S_FG.range s.toHom) (f.comp
    s.equivRange.symm.toEmbedding)
  use t.toEmbedding.comp r
  -- We have defined the embedding, and the rest is just a bunch of
    simplifications of the equation.
  change _ = t.toEmbedding.comp s
  ext x
  have eq' := congr_fun (congr_arg DFunLike.coe eq) ⟨s x, Hom.mem_range.2 ⟨x,
    rfl⟩⟩
  simp only [Embedding.comp_apply, Hom.comp_apply,
    Equiv.coe_toHom, Embedding.coe_toHom, coeSubtype] at eq'
  simp only [Embedding.comp_apply, ← eq', Equiv.coe_toEmbedding,
    EmbeddingLike.apply_eq_iff_eq]
  apply (Embedding.equivRange (Embedding.comp r g)).injective
  ext
  simp only [Equiv.apply_symm_apply, Embedding.equivRange_apply, s]
```

Theorem 3.18 (Uniqueness of Fraïssé Limits)**.** Fraïssé limits are unique up to equivalence.

Proof. Let K be a class of finitely generated structures, and M and N be Fraïssé limits of K. We first show that M and N form an extension pair. Suppose that $f : A \simeq B$ is an equivalence between finitely generated substructures of M and N, and $m \in M$. Let S be the substructure generated by A and m. We have $\operatorname{Age} M = \operatorname{Age} N$, so there exists an embedding $g : S \hookrightarrow N$. Using lemma 3.17 with f, the inclusion of A in S, and g, we get an embedding $h : S \hookrightarrow N$ that extends f.

By symmetry, N and M also form an extension pair. Finally, to apply the theorem 2.1, we just need to show that there exists at least one equivalence between finitely generated substructures of M and N. Let $A \subseteq M$ be the substructure generated by the empty set, since $\operatorname{Age} M = \operatorname{Age} N$ there is a substructure $A' \subseteq N$ and an equivalence $A \simeq A'$. □

In Lean, we first show that two Fraïssé limits form an extension pair. It closely follows the proof we just did:

Fraïssé limits form an extension pair

```
-- Code in ModelTheory/Fraisse [AKWb]
theorem isExtensionPair (hM : IsFraisseLimit K M) (hN : IsFraisseLimit K N) :
    L.IsExtensionPair M N := by
  intro ⟨f, f_FG⟩ m
  -- We have an equivalence ‘f‘ between finitely generated substructures of
    ‘M‘ and ‘N‘, and an element ‘m : M‘.
  -- ‘S‘ is the substructure generated by the domain of ‘f‘ and ‘m‘.
  let S := f.dom ⊔ closure L {m}
  have S_FG : S.FG := f_FG.sup (Substructure.fg_closure_singleton _)
  -- ‘S‘ is in the age of ‘N‘, since it is in the age of ‘M‘.
  have S_in_age_N : ⟨S, inferInstance⟩ ∈ L.age N := by
    rw [hN.age, ← hM.age]
    exact ⟨(fg_iff_structure_fg S).1 S_FG, ⟨subtype _⟩⟩
  -- So ‘S‘ has an embedding in ‘N‘.
  have nonempty_S_N : Nonempty (S ↪[L] N) := S_in_age_N.2
  -- We use ‘extend_embedding‘ to get an embedding ‘g : S ↪[L] N‘ such that
    its composition with the inclusion ‘f.dom ↪[L] S‘ is equal to the
    inclusion ‘f.cod ↪[L] N‘ composed with ‘f‘.
  let ⟨g, g_eq⟩ := hN.ultrahomogeneous.extend_embedding
    (f.dom.fg_iff_structure_fg.1 f_FG)
    ((subtype f.cod).comp f.toEquiv.toEmbedding) (inclusion (le_sup_left : _ ≤
    S))
  refine ⟨⟨⟨S, g.toHom.range, g.equivRange⟩, S_FG⟩,
    subset_closure.trans (le_sup_right : _ ≤ S) (mem_singleton m), ⟨
    le_sup_left, ?_⟩⟩
  -- The rest is showing that ‘g‘ as an equivalence between ‘S‘ and a
    substructure of ‘N‘ is indeed an extension of ‘f‘.
  ext
  simp [Subtype.mk_le_mk, PartialEquiv.le_def, g_eq]
```

And the proof that Fraïssé limits are unique up to equivalence:

Uniqueness of Fraïssé limits

```
-- Code in ModelTheory/Fraisse [AKWb]
/-- The Fraisse limit of a class is unique, in that any two Fraisse limits are
    isomorphic. -/
theorem nonempty_equiv : Nonempty (M ≃[L] N) := by
  -- ‘S‘ is the substructure generated by the empty set in ‘M‘.
  let S : L.Substructure M := ⊥
  have S_fg : FG L S := (fg_iff_structure_fg _).1 Substructure.fg_bot
  -- It has an embedding in ‘N‘, since it is a member of ‘L.age N‘.
  obtain ⟨_, ⟨emb_S : S ↪[L] N⟩⟩ : ⟨S, inferInstance⟩ ∈ L.age N := by
    rw [hN.age, ← hM.age]
    exact ⟨S_fg, ⟨subtype _⟩⟩
  -- Therefore there exists a partial equivalence ‘v‘ between ‘M‘ and ‘N‘.
  let v : M ≃ₚ[L] N := {
    dom := S
    cod := emb_S.toHom.range
```

```
    toEquiv := emb_S.equivRange
  }
  -- And we apply the back-and-forth method.
  exact ⟨Exists.choose (equiv_between_cg cg_of_countable cg_of_countable
    ⟨v, ((Substructure.fg_iff_structure_fg _).2 S_fg)⟩ (hM.isExtensionPair hN)
      (hN.isExtensionPair hM))⟩
```

3.7 Fraïssé limit of finite graphs

In Mathlib, the Fraïssé limits of finite sets and finite total orders are already present, and they are respectively the countable sets and the countable total dense orders without minimal or maximal element. In this subsection, we will prove that any countable simple graph with the extension property is the Fraïssé limit of the finite simple graphs.

In this subsection, all the graphs are simple graphs. The language of graphs is simply a binary relation symbol of adjacency $\sim$, and the theory is that $\sim$ is irreflexive and symmetric. This is how it is done in Mathlib:

```
-- Code in ModelTheory/Graph, written by Anderson [And]
-- Definition of the relation symbols. We only have one for arity 2, ‘adj‘.
inductive graphRel : ℕ → Type
  | adj : graphRel 2
  deriving DecidableEq

/-- The language consisting of a single relation representing adjacency. -/
def graph : Language := ⟨fun _ => Empty, graphRel⟩
  -- This is so that Lean derives automatically that this language is
    relational, meaning that there are no function symbols.
  deriving IsRelational

/-- The symbol representing the adjacency relation. -/
abbrev adj : Language.graph.Relations 2 := .adj
```

The last line allows us to write `RelMap adj ![x, y]` for $x \sim y$, and this is how it will be written in the code. The part `![x, y]` is a practical way to define a function from `Fin 2` sending 0 to x and 1 to y. The theory is written as:

```
-- Code in ModelTheory/Graph, written by Anderson [And]
/-- The theory of simple graphs. -/
def Theory.simpleGraph : Language.graph.Theory :=
  {adj.irreflexive, adj.symmetric}
```

The Rado graph, also called the random graph, is an isomorphism class of countable graphs characterized by the following property:

Definition 3.19 (Extension property)**.** A graph G has the extension property if for any two disjoint finite sets of vertices $A, B \subseteq V(G)$, there exists some vertex $v \in V(G) \setminus A \cup B$ such that v is adjacent to all the vertices in A and not adjacent

to all the vertices in B.

The translation in Lean is straightforward:

```
-- Code in ModelTheory/Graph [Kol25]
/-- A graph has the extension property if for for any two disjoint finite sets
    of vertices, there exists a vertex which is adjacent to all vertices in
    one and to no vertices in the other. It characterizes the Rado graph. -/
def ExtensionProperty : Prop :=
    ∀ {A B : Finset V}, ∀ (_ : Disjoint A B), ∃ v ∉ A ∪ B,
      (∀ a ∈ A, RelMap adj ![v, a]) ∧ ∀ b ∈ B, ¬ RelMap adj ![v, b]
```

We will show that a countable graph with this property is the Fraïssé limit of finite graphs, and that any countable graph can embed in it. First, we show this important result:

Theorem 3.20. Let G, H be two graphs, and G satisfies the extension property. Then (H, G) is an extension pair.

Proof. Let $f : H' \simeq G'$ be an equivalence between two finitely generated substructures of H and G. Since there are no function symbols in the language of graphs, substructures are finitely generated if and only if they are finite. Also, substructures simply correspond to subsets of vertices. Let $\tilde{v}$ be a vertex not in G'. We define

$$A := \{f(v) \mid v \in G', v \sim \tilde{v}\}$$

$$B := \{f(v) \mid v \in G', v \nsim \tilde{v}\}$$

Since f is injective, A and B are disjoint, so we can use the extension property to get a vertex w in G such that $v \sim w$ for all $v \in A$, and $v \nsim w$ for all $v \in B$. Then we can extend f by sending $\tilde{v}$ to w. □

We will now prove that in Lean. First we prove that two vertices in a substructure are adjacent if and only if they are adjacent in whole structure.

```
-- Code in ModelTheory/Graph [Kol25]
-- This is saying that in a substructure, the adjacency relation is the same
    as the one in the whole structure.
theorem substructure_adj_iff {S : Language.graph.Substructure V} (x y : S) :
    RelMap adj ![x, y] ↔ RelMap adj ![(x : V), y] := by
  simp only [RelMap, Substructure.inducedStructure]
  constructor <;> intro h <;> convert h using 1 <;> rw [←List.ofFn_inj] <;>
    rfl
```

Next, we prove that a substructure is also a model of the theory of graphs. This is easy using the previous lemma:

```
-- Code in ModelTheory/Graph [Kol25]
theorem substructure_models_simpleGraph (S : Language.graph.Substructure V) :
    S ⊨ Theory.simpleGraph := by
  -- Replace the theory by its definition, so adjacency being irreflexive and
    symmetric.
  rw [Theory.simpleGraph_model_iff]
  -- We use the previous lemma to replace the adjacency in the substructure
    for the adjacency in the whole structure.
  simp_rw [substructure_adj_iff]
  exact ⟨fun x y ↦ adj_irrefl (V := V) x y, fun _ _ h ↦ adj_symm h⟩
```

We have all we need to do the proof:

```
-- Code in ModelTheory/Graph [Kol25]
theorem ExtensionProperty.extensionPair_Countable (ext_prop :
    ExtensionProperty V) :
    IsExtensionPair Language.graph W V := by
  -- This line is so that Lean knows we use classical logic and not
    constructive logic, which is necessary for some definitions we use during
    the proof.
  classical
  -- We didn't give a name to the instances that `V` and `W` model the theory
    of simple graphs, we can give them a name now with the tactic rename_i.
  rename_i V_simple_graph _ W_simple_graph
  -- Being an extension pair was defined with partial equivalences, but it is
    simpler to use the equivalent definition of being able to extend
    embeddings, so we replace it with that.
  rw [isExtensionPair_iff_exists_embedding_closure_singleton_sup]
  intro S S_fg f m
  -- So at this point, we have a substructure `S`, an embedding `f` of `S`
    into `V`, and an element `m`. The goal is to find an embedding `g` of the
    substructure generated by `m` and `S` into `V` which extends `f`.
  -- `A` is the image by `f` of the set of vertices in `S` which are adjacent
    to `m`.
  let A := f '' {v | RelMap adj ![m, v]}
  -- `S` is finite since it is finitely generated and the language is
    relational.
  have : Finite S := S_fg.finite
  -- There is a special type for finite sets, `Finset`, which we need to use
    the extend property. `A` is a finite set, since it is the image of `S`
    which is finite, so we can construct a `Finset` element from `A`.
  let A_Finset := Set.Finite.toFinset (Finite.Set.finite_image .. : Finite A)
  -- `B` is the image by `f` of vertices in `S` not adjacent to `m`.
  let B := f '' {v | ¬ RelMap adj ![m, v]}
  let B_Finset := Set.Finite.toFinset (Finite.Set.finite_image .. : Finite B)

  -- We prove that `A` and `B` are disjoint as `Finset` elements.
  have A_B_disjoint : Disjoint A_Finset B_Finset := by
    -- First we use the fact that they are disjoint as `Finset` if they are
```

```
  disjoint as sets.
  apply Set.Finite.disjoint_toFinset.2
  -- Since ‘f‘ is injective, and both ‘A‘ and ‘B‘ were the images of sets by
  ‘f‘, we can replace them by their preimage.
  refine (Set.disjoint_image_iff f.injective).2 ?_
  -- Then we use the fact that two sets are disjoint if and only if their
  intersection is a subset of the empty set, so any element in their
  intersection must be an element of the empty set.
  rw [Set.disjoint_iff]
  -- We take ‘x‘ a vertex, and ‘hx‘ the property of being contained in the
  intersection.
  intro x hx
  -- The property of being contained in an intersection is equivalent to
  being contained in one set and contained in the other set, and being
  contained in a set is equivalent to verifying the property that defines
  the set, so after that ‘hx‘ is a ‘Prop‘ that tells us that ‘x‘ is adjacent
  to ‘m‘ and ‘x‘ is not adjacent to ‘m‘.
  simp only [Set.mem_inter_iff, Set.mem_setOf_eq] at hx
  -- We separate ‘hx‘ in two different variables.
  let ⟨_, _⟩ := hx
  -- And we have a contradiction.
  contradiction

-- Of course, the image of any vertex in ‘S‘ by ‘f‘ must be contained in the
  union of ‘A‘ and ‘B‘.
have A_B_cover_image : ∀ x, f x ∈ A ∪ B := by
  intro x
  -- We simplify everything to: ‘x‘ is adjacent to ‘m‘ or ‘x‘ is not
  adjacent to ‘m‘.
  simp only [Set.mem_union, Set.mem_image, Set.mem_setOf_eq,
  EmbeddingLike.apply_eq_iff_eq, exists_eq_right, A, B]
  -- And we conclude by the law of excluded middle.
  exact Classical.em (RelMap adj ![m, ↑x])

-- We use the extension property on ‘A‘ and ‘B‘, so we get a vertex ‘v‘ in
  ‘V‘ which is not ‘A‘ and not in ‘B‘, which is adjacent to all vertices in
  ‘A‘ and no vertices in ‘B‘.
let ⟨v, v_not_in_AB, v_adj_A, v_not_adj_B⟩ := ext_prop A_B_disjoint
-- But all these properties were stated with ‘A‘ and ‘B‘ with ‘Finset‘ type,
  but we don't need the fact that they are finite anymore, and it is easier
  to work with sets, so the next line replaces everything with sets.
simp only [A_Finset, B_Finset, Finset.mem_union, Set.Finite.mem_toFinset]
  at v_not_in_AB v_adj_A v_not_adj_B
-- ‘v‘ is not in the image of ‘S‘ by ‘f‘, since ‘v‘ is not in the union of
  ‘A‘ and ‘B‘, and they cover the image.
have v_not_in_image : ∀ x, f x ≠ v := fun x h ↦ v_not_in_AB (h ▸
  A_B_cover_image x)
-- For any vertex ‘x‘ in ‘S‘, its image is adjacent to ‘v‘ if and only if
  its image is in ‘A‘.
```

```
have v_adj_iff_A : ∀ x : S, RelMap adj ![v, f x] ↔ f x ∈ A := by
  intro x
  -- the 'if' part follows from the fact that `v` is adjacent to all
  elements in `A`.
  refine ⟨?_, fun H ↦ v_adj_A (f x) H⟩
  -- Now for the 'only' part. Suppose that `x` is adjacent to `v`.
  intro v_adj_fx
  -- We know that `f x` is contained in the union of `A` and `B`, and this
  equivalent to `x` being in `A` or in `B`. We will prove for both cases.
  cases (Set.mem_union ..).1 (A_B_cover_image x)
  -- If `x` is in `A`, then `x` is in `A` obviously.
  · assumption
  -- If `x` is in `B`, we will construct a contradiction.
  · by_contra
    -- `v` is not adjacent to any elements in `B`, but it is adjacent to `f
  x`, and `f x` is in `B`.
    exact v_not_adj_B (f x) (by assumption) (by assumption)

-- We define `S'` the substructure generated by `m` and `S`.
let S' := Substructure.closure Language.graph {m} ⊔ S

-- We will separate in two cases: `m` is in `S`, and `m` is not in `S`.
cases Classical.em (m ∈ S)

-- If `m` is in `S`, it is quite simple.
case inl m_in_S =>
  -- `S'` is equal to `S` (propositionally, not definitionally). In
  particular, it is contained in `S`, so there is an inclusion of this
  substructure into `S`. We
  have : S' = S := by
    apply sup_eq_right.2
    exact Substructure.closure_le.2 (Set.singleton_subset_iff.2 m_in_S)
  -- So in particular, `S'` is contained in `S`, so there is an inclusion of
  this substructure into `S`. We compose it with `f`, and Lean can see by
  definition that it is an extension of `f`.
  use f.comp (Substructure.inclusion (le_of_eq this))
  rfl

-- Now the case when `m` is not in `S`.
case inr m_not_in_S =>
  -- We first prove that a vertex is in `S'` if and only if it is equal to
  `m` or it is in `S`.
  have mem_S'_iff : ∀ x, x ∈ S' ↔ x = m ∨ x ∈ S := by
    intro x
    unfold S'
    -- We just simplify everything, using in particular the fact that with a
  relational language, the substructure generated by a set doesn't have
  additional elements.
    rw [← Substructure.mem_coe, ← Substructure.closure_eq S, ←
```

```
Substructure.closure_union]
  simp only [Set.singleton_union, Substructure.closure_eq_of_isRelational,
Set.mem_insert_iff, SetLike.mem_coe, Substructure.closure_eq]

-- We define ‘g‘, a function from ‘S’‘ to ‘V‘ which sends any element in
‘S‘ to its image by ‘f‘, and otherwise to ‘v‘. We need to prove that this
is an embedding.
let g (s : S') : V := if h : ↑s ∈ S then f ⟨s, h⟩ else v

-- First, we prove that it commutes with adjacency, so two vertices are
adjacent if and only if their images by ‘g‘ are adjacent.
have g_morphism : ∀ x y, RelMap adj ![x, y] ↔ RelMap adj ![g x, g y] := by
  intro ⟨x, hx⟩ ⟨y, hy⟩
  unfold g
  -- Since we know that an element is in ‘S’‘ if and only if they are
equal to ‘m‘ or they are in ‘S‘, we can separate in four cases for ‘x‘ and
‘y‘. Since for each case we know if ‘x‘ and ‘y‘ are in ‘S‘ or not (since
‘m‘ is not in ‘S‘), we simplify all the "if then", for all the cases in
one line. We also apply the result that two vertices are adjacent if and
only if they are adjacent in the whole structure.
  cases (mem_S'_iff x).1 hx <;> cases (mem_S'_iff y).1 hy
    <;> rename_i h' h'' <;> simp only [h', h'', m_not_in_S, ↓reduceDIte,
        substructure_adj_iff]
  -- First case, both ‘x‘ and ‘y‘ are equal to ‘m‘, so it has already been
simplified to ‘m‘ being adjacent to ‘m‘ if and only if ‘v‘ is adjacent to
‘v‘. We use the fact that adjacency is irreflexive to show that in both
directions, we get a contradiction.
  · constructor <;> intro H <;> by_contra <;> exact adj_irrefl _ H
  -- This case is simplified to ‘m‘ adjacent to ‘y‘ if and only if ‘v‘ is
adjacent to ‘f y‘. We use that ‘v‘ is adjacent to an element if and only
if it is contained in ‘A‘, and after that it is just simplifications in
autopilot.
  · simp only [v_adj_iff_A, Set.mem_image, Set.mem_setOf_eq,
EmbeddingLike.apply_eq_iff_eq, exists_eq_right, A]
  -- This case is simplified to ‘x‘ adjacent to ‘m‘ if and only if ‘f x‘
is adjacent to ‘v‘. We use the fact that adjacency is symmetric, and then
we do exactly the same simplifications as for the previous case.
  · nth_rw 2 [adj_symm']
    rw [adj_symm']
    simp only [v_adj_iff_A, Set.mem_image, Set.mem_setOf_eq,
EmbeddingLike.apply_eq_iff_eq, exists_eq_right, A]
  -- This case is simplified to ‘x‘ adjacent to ‘y‘ if and only if ‘f x‘
is adjacent to ‘f y‘. We can use the fact that ‘f‘ is an embedding, so it
commutes with ‘adjacency‘, but some work is needed since it is not exactly
in the right form with the ‘![·,·]‘ notation.
  · have H := f.map_rel adj ![⟨x, h'⟩, ⟨y, h''⟩]
    rw [substructure_adj_iff] at H
    convert H.symm
    rw [←List.ofFn_inj]
```

```
    rfl

-- We now prove that `g` is injective.
have g_inj : Function.Injective g := by
  -- Suppose that the images of `x` and `y` are equal. We need to prove
that `x` and `y` are equal as elements of `S'`.
  intro ⟨x, hx⟩ ⟨y, hy⟩ h
  -- This is equivalent to being equal as elements of `W`.
  simp only [Subtype.mk.injEq]
  unfold g at h
  -- We again separate in all different cases, with `x` and `y` being
equal to `m` or contained in `S`. We also do a bunch of simplifications,
and two cases are already done.
  cases (mem_S'_iff x).1 hx <;> cases (mem_S'_iff y).1 hy <;> rename_i h'
h'' <;>
    simp only [h', h'', m_not_in_S, ↓reduceDIte, v_not_in_image] at h ⊢
  -- First case with `x = m` and `y` contained in `S`, we have a
hypothesis that `v` is equal to `f y`, but we already know that this is
impossible, so we get a contradiction.
  · by_contra
    exact v_not_in_image _ h.symm
  -- In this case, we have `x` and `y` both in `S`, so we can conclude by
the fact that `f` is injective. The second line is because technically `f`
being injective implies that `x` and `y` are equal as elements of `S`, not
as elements of `V`.
  · convert f.injective h
    exact Subtype.mk_eq_mk.symm

-- With all that, we can finally define the embedding with underlying
function `g`, and we use it.
use {
  inj' := g_inj
  -- This says that `g` commutes with adjacency, but again because of
technicalities with the `![·,·]` function, it is not exactly stated like we
stated it before, and we need to do some transformations.
  map_rel' := by
    intro n r
    cases r
    intro x
    have h := (g_morphism (x 0) (x 1)).symm
    convert h <;> simp only [←List.ofFn_inj] <;> rfl
}
-- The goal is now to prove that `g` extends `f`. But it follows by
simplifying everything.
ext x
let ⟨x, x_in_S⟩ := x
simp only [Embedding.comp_apply, Substructure.coe_inclusion,
Set.inclusion_mk,
  Embedding.comp_apply]
```

```
    simp only [DFunLike.coe, x_in_S, ↓reduceDIte]
```

Corollary 3.21. Let G and H be two countable graphs satisfying the extension property. Then G and H are isomorphic.

Proof. The empty function is an equivalence between the empty substructures of G and H, and we extend it by Theorem 2.1 to a full equivalence. □

We didn't prove this in Lean since it follows from the fact that they are Fraïssé limits, which we will prove later.

But we also get that any countable graph embeds in a graph with the extension property:

Corollary 3.22. Let G and H be two countable graphs, and G satisfies the extension property. Then H embeds in G.

Proof. The empty function is an equivalence between the empty substructures of H and G so we get an embedding by Theorem 2.2. □

In Lean:

```
-- Code in ModelTheory/Graph [Kol25]
theorem ExtensionProperty.embedding_from_countable (ext_prop :
    ExtensionProperty V) [Countable W] :
    Nonempty (W ↪[Language.graph] V) :=
  -- We apply the theorem ‘embedding_from_cg‘. Since this theorem tells us
    that there exists an embedding, we use ‘Exists.choose‘ to take one.
  ⟨Exists.choose <|
    embedding_from_cg (L := Language.graph) (M := W) (N := V)
    Structure.cg_of_countable
      inhabited_FGEquiv_of_IsEmpty_Constants_and_Relations.default
        (ext_prop.extensionPair_Countable _)⟩
```

Corollary 3.23. Any countable graph with the extension property is ultrahomogeneous.

Proof. Follows directly from Lemma 3.13. □

With all that, we can finally prove that a countable graph with the extension property is the Fraïssé limit of finite graphs:

Theorem 3.24. A countable graph G with the extension property is the Fraïssé limit of finite graphs.

Proof. By the previous corollary, it is ultrahomogeneous. Any finitely generated substructure is a finite graph, and by Corollary 3.22, any finite graph embeds in G, therefore $\mathrm{Age}\, G$ is the class of finite graphs. □

In Lean:

```
-- Code in ModelTheory/Graph [Kol25]
/-- Any graph satisfying the extension property is the Fraisse limit of the
    class of finite
  graphs. -/
theorem ExtensionProperty.isFraisseLimit_finite_graphs (ext_prop :
    ExtensionProperty V)
    [Countable V] : IsFraisseLimit
      { G : CategoryTheory.Bundled Language.graph.Structure | G ⊨
    Theory.simpleGraph ∧ Finite G }
        V := by
  constructor
  -- First we prove that it is ultrahomogeneous.
  · rw [isUltrahomogeneous_iff_IsExtensionPair Structure.cg_of_countable]
    exact ext_prop.extensionPair_Countable V
  -- Then we prove that its age is equal to the class of finitely generated
    graphs. This is equivalent to say that for any structure ‘G‘ in the
    language of graphs, ‘G‘ is in the age of ‘V‘ if and only if it is in the
    class of finitely generated graphs.
  · ext G
    -- We simplify to ‘G‘ being finite and having an embedding into ‘V‘ if and
    only if it is a model of the theory of graphs and it is finite.
    simp only [age, Set.mem_setOf_eq, Structure.fg_iff_finite]
    constructor
    -- First suppose that ‘G‘ is finite and has an embedding ‘f‘ into ‘V‘.
    Then we use the fact that any substructure of ‘V‘ must also verify the
    theory of graphs.
    · intro ⟨G_finite, ⟨f⟩⟩
      refine ⟨?_, G_finite⟩
      apply StrongHomClass.theory_model f.equivRange.symm
    -- Suppose that ‘G‘ models the theory of graphs and is finite, then we use
    the previous result that any countable graph embeds in ‘V‘.
    · intro ⟨G_graph, G_finite⟩
      refine ⟨G_finite, ?_⟩
      apply ext_prop.embedding_from_countable (W := G)
```

Of course, we haven't proved that there exists a countable graph satisfying the extension property. The first construction (or at least a directed version of it) was discovered by Ackermann [Ack37], and then other constructions were discovered and studied by Erdős and Rényi [ER63] and Rado [Rad64].

Definition 3.25 (Rado graph)**.** We define a series of sets recursively: $A_0 = \emptyset$, and $A_{n+1} = \{a \mid a \subseteq A_n\}$. We then define $A := \bigcup A_n$, and we put a structure of graph as follows: for any two $a, b \in A$, we have $a \sim b$ if and only if $a \in b$ or $b \in a$.

This construction was discovered by Ackermann. It is only one of many different ways of constructing a graph satisfying the extension property. All these constructions are

equivalent, in the sense that they construct graphs which are all isomorphic.

Theorem 3.26. The Rado graph satisfies the extension property and is countable.

Proof. First, it is a graph, since the definition of the adjacency relation is symmetric, and it is also irreflexive because no subset contains itself. It is countable, since each A_n is finite. Suppose that we have $B, C \subseteq A$ two disjoint finite subsets of vertices. Then there exists some n such that $B, C \subseteq A_n$. We have $A_n \in A_{n+1}$, and we take

$$D := B \cup \{A_n\} \in A_{n+2}$$

D is adjacent to all the vertices in B, since it contains them. It contains no element in C, since C cannot contain A_n, since we don't have $A_n \in A_n$, and B and C are disjoint. Also, it cannot be contained in any element of C, since it would mean that A_n is contained in a set contained in a set contained in A_n and therefore $A_n \in A_n$. □

We didn't construct it in Lean, since it was already in project by another contributor to construct the Rado graph via another method.

4 Future works

There are many promising directions for extending this work. Only a small portion of model theory has been formalized using interactive theorem provers. Future efforts could focus on formalizing key definitions and results in areas such as stability, categoricity, saturation, quantifier elimination, minimality, and o-minimality.

The Hrushovski construction is a simple generalization of Fraïssé limits, and its existence and uniqueness can be proved in essentially the same way. It could be an interesting extension of this work to try to formalize this construction in Lean.

References

[Ack37] Wilhelm Ackermann. Die widerspruchsfreiheit der allgemeinen mengenlehre. *Mathematische Annalen*, 114(1):305–315, 1937.

[AHvD] Aaron Anderson, Jesse Michael Han, and Floris van Doorn. Modeltheory/basic file in the mathlib repository. GitHub repository. https://github.com/leanprover-community/mathlib4/blob/master/Mathlib/ModelTheory/Basic.lean, Accessed: January 31, 2025.

[AKa] Aaron Anderson and Gabin Kolly. Modeltheory/directlimit file in the mathlib repository. GitHub repository. https://github.com/leanprover-community/mathlib4/blob/master/Mathlib/ModelTheory/DirectLimit.lean, Accessed: January 31, 2025.

[AKb] Aaron Anderson and Gabin Kolly. Modeltheory/finitelygenerated file in the mathlib repository. GitHub repository. https://github.com/leanprover-community/mathlib4/blob/master/Mathlib/ModelTheory/FinitelyGenerated.lean, Accessed: January 31, 2025.

[AKc] Aaron Anderson and Gabin Kolly. Modeltheory/substructures file in the mathlib repository. GitHub repository. https://github.com/leanprover-community/mathlib4/blob/master/Mathlib/ModelTheory/Substructures.lean, Accessed: January 31, 2025.

[AKWa] Aaron Anderson, Gabin Kolly, and David Wärn. Modeltheory/directlimit file in the mathlib repository. GitHub repository. https://github.com/leanprover-community/mathlib4/blob/master/Mathlib/ModelTheory/PartialEquiv.lean, Accessed: January 31, 2025.

[AKWb] Aaron Anderson, Gabin Kolly, and David Wärn. Modeltheory/fraisse file in the mathlib repository. GitHub repository. https://github.com/leanprover-community/mathlib4/blob/master/Mathlib/ModelTheory/Fraisse.lean, Accessed: January 31, 2025.

[Alp24] AlphaProof and AlphaGeometry teams. AI achieves silver-medal standard solving International Mathematical Olympiad problems. https://deepmind.google/discover/blog/ai-solves-imo-problems-at-silver-medal-level/, July 2024. Accessed: 2024-12-18.

[And] Aaron Anderson. Modeltheory/graph file in the mathlib repository. GitHub repository. https://github.com/leanprover-community/mathlib4/blob/master/Mathlib/ModelTheory/Graph.lean, Accessed: January 31, 2025.

[Ber07] Stefan Berghofer. First-order logic according to fitting. *Archive of Formal Proofs*, August 2007. https://isa-afp.org/entries/FOL-Fitting.html, Formal proof development.

[Com20] Mathlib Community. The lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, page 367–381, New York, NY, USA, 2020. Association for Computing Machinery.

[Com22] Mathlib Community. Completion of the liquid tensor experiment, blog post. https://leanprover-community.github.io/blog/posts/lte-final/, 2022. Accessed: 04.12.2024.

[dMC] Leonardo de Moura and Mario Carneiro. Data/nat/pairing file in the mathlib repository. GitHub repository. https://github.com/leanprover-community/mathlib4/blob/master/Mathlib/Data/Nat/Pairing.lean, Accessed: January 31, 2025.

[ER63] P. Erdős and A. Rényi. Asymmetric graphs. *Acta Mathematica Academiae Scientiarum Hungarica*, 14(3):295–315, 1963.

[Fro21] Asta Halkjær From. Soundness and completeness of an axiomatic system for first-order logic. *Archive of Formal Proofs*, September 2021. https://isa-afp.org/entries/FOL_Axiomatic.html, Formal proof development.

[GPTS22] Emmanuel Gunther, Miguel Pagano, Pedro Sánchez Terraf, and Matías Steinberg. The independence of the continuum hypothesis in isabelle/zf. *Archive of Formal Proofs*, March 2022. https://isa-afp.org/entries/Independence_CH.html, Formal proof development.

[Hod97] W. Hodges. *A Shorter Model Theory*. Cambridge University Press, 1997.

[HvD19] Jesse Michael Han and Floris van Doorn. A Formalization of Forcing and the Unprovability of the Continuum Hypothesis. In John Harrison, John O'Leary, and Andrew Tolmach, editors, *10th International Conference on Interactive Theorem Proving (ITP 2019)*, volume 141 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 19:1–19:19, Dagstuhl, Germany, 2019. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[HvD20] Jesse Michael Han and Floris van Doorn. A formal proof of the independence of the continuum hypothesis. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, page 353–366, New York, NY, USA, 2020. Association for Computing Machinery.

[JJS12] Peter Koepke Julian J. Schlöder. The gödel completeness theorem for uncountable languages. *Formalized Mathematics*, 20(3):199–203, 2012.

[KA24] Gabin Kolly and Aaron Anderson. Pull request #9967: Proof that fraïssé limits are unique, 2024. https://github.com/leanprover-community/mathlib4/pull/9967, Accepted into Mathlib4.

[Kol24] Gabin Kolly. Pull request #18876: Proof of the existence of fraïssé limits, 2024. https://github.com/leanprover-community/mathlib4/pull/18876.

[Kol25] Gabin Kolly. Pull request #20649: Characterization of the fraïssé limit of finite graphs, 2025. https://github.com/leanprover-community/mathlib4/pull/20649.

[Mas21] Patrick Massot. Why formalize mathematics? https://www.imo.universite-paris-saclay.fr/~patrick.massot/files/exposition/why_formalize.pdf, December 2021. Accessed: 2024-12-18.

[Mat21] Mathlib Community. Mathlib4 repository. https://github.com/leanprover-community/mathlib4, May 2021. GitHub repository created in May 2021, maintained by the Mathlib Community.

[Mat25] Mathlib Community. Mathlib community statistics. https://leanprover-community.github.io/mathlib_stats.html, 2025. Accessed: January 3, 2025.

[MU21] Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction – CADE 28*, pages 625–635, Cham, 2021. Springer International Publishing.

[O'C05] Russell O'Connor. *Essential Incompleteness of Arithmetic Verified by Coq*, page 245–260. Springer Berlin Heidelberg, 2005.

[Pau13] Lawrence C. Paulson. Gödel's incompleteness theorems. *Archive of Formal Proofs*, November 2013. https://isa-afp.org/entries/Incompleteness.html, Formal proof development.

[Rad64] Richard Rado. Universal graphs and universal functions. *Acta Arithmetica*, 9(4):331–340, 1964. Available as a PDF.

[SdLAR24] Fabián Fernando Serran Suárez, Thaynara Arielly de Lima, and Mauricio Ayala-Rincón. Compactness theorem for propositional logic and combinatorial applications. *Archive of Formal Proofs*, August 2024. https://isa-afp.org/entries/Prop_Compactness.html, Formal proof development.

[Tea23] Formalized Formal Logic Team. Formalized formal logic. https://github.com/FormalizedFormalLogic, 2023.

[Wä] David Wärn. Order/ideal file in the mathlib repository. GitHub repository. https://github.com/leanprover-community/mathlib4/blob/master/Mathlib/Order/Ideal.lean, Accessed: January 31, 2025.