University of Marburg

Faculty of Mathematics and Computer Science

Bachelor Thesis

Modelling of Coalgebra in Lean

by
Qais Hamarneh
September 2019

Supervisor:
Prof. Dr. Heinz Peter Gumm

Work-Group Formal Methods

Erklärung

Ich, Qais Hamarneh (Informatikstudent an der Philipps-Universität Marburg, Matrikelnummer 2773350), versichere an Eides statt, dass ich die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Die hier vorliegende Bachelorarbeit wurde weder in ihrer jetzigen noch in einer ähnlichen Form einer Prüfungskommission vorgelegt.

I, Qais Hamarneh (computer science student at Philipps-Universität Marburg, Matrikelnummer 2773350), affirm in lieu of oath that I have written this bachelor thesis independently and that I have not used any other sources and aids than those indicated. The bachelor thesis presented here was neither submitted in its present form nor in a similar form to an examination board.

Marburg, September 23, 2019

Qais Hamarneh

Abstract

In this thesis, we explore the Lean theorem prover. We present the predefined formalization of category theory in Lean and extend it to include the definitions of certain limits and colimits and their instances in the category of sets. We build on this to formalize the basic definitions and theorems of universal coalgebra.

Contents

1. Introduction

1.1. Motivation

Checking the correctness of mathematical proofs is a rigorous, exhausting and often thankless job. Many proofs take years to examine and even then they might not be fully verified, as was the case with Thomas Hales' proof of the Kepler conjecture [1]. This makes mathematical verification a perfect job for a computer, which raises the demand for automated proof assistants and theorem provers.

The principles for theorem provers date back to the early foundations of logic in classic Greece. However, the logic used in most proof assistants today is built on Bertrand Russell's type theory, the work of David Hilbert on formal logic, Alonzo Church's Lambda calculus and the connection between them through the Curry-Howard-Correspondence (or propositions as types) [2].

In this work, we explore how category theory is formalized in the Lean theorem prover and add to it, in order to build a base for formalizing the basic definitions and theorem of universal coalgebra.

Lean is a relatively modern theorem prover [3] and is still an ongoing project. Nevertheless, it is already being used to formalize many fields of mathematics.

Universal coalgebra is the category theoretical dual structure of universal algebra. It serves as the theory of state based systems. This makes it of great importance for computer science.

1.2. Work structure

In chapter 2, we introduce the Lean theorem prover and its logical foundation with a short introduction to its syntax and programming paradigms.

Afterwards in chapter 3, we take a look at the basics of category theory and how they are modeled in Lean. Each concept is introduced first mathematically[1], based heavily on the work of Professor H. Peter Gumm in [4] and then its formalization in Lean. The Lean code in this chapter can be devided into two parts. The first comes from the category theory library in Lean-mathlib [2]. The second part, we developed to allow us to facilitate the modeling of coalgebra.

In chapter 4, we build on concepts introduced in chapter 3, to explore the basics of coalgebra. We follow the same structure as in 3, first a mathematical definition, then its modeling.

[1] Snippets of the proofs will only be presented in text. The full code can be found in the appendix.

[2] Lean-mathlib is developed by the lean community and it covers many branches of mathematics. It also introduces many new tactics, of which I made use in this work.
github.com/leanprover-community/mathlib

The mathematical definitions in this chapter are based on [4]. The code, however, was developed from scratch.

1.3. Related Work

As mentioned above, the mathematical part of this work is based on the work Professor H. Peter Gumm in [4] and the formalization is built on Lean-mathlib developed by Lean's open source community.

Other notable work is the ongoing project to formalize universal algebra in Lean lead by Prof. William DeMeo [5]. The idea of this project was the inspiration to explore Lean as theorem prover and start this work.

Many other fields of mathematics and logic are being formalized in Lean today including among others homo algebra, number theory, analysis, topology and set theory.

2. Background

In this chapter we will introduce the basics of the logical foundations and syntax of Lean. It is mainly based on Lean's online tutorial [3], unless specified otherwise.

2.1. Lean Theorem Prover

Lean is an open source theorem prover. It was launched by Leonardo de Moura at Microsoft Research Center in 2013 and is being developed at Microsoft Research and Carnegie Mellon University. It has a small trusted kernel based on dependent type theory or more precisely [3]:

> "A version of dependent type theory called *Calculus of Constructions* with a countable hierarchy of non-cumulative universes and inductive types."

We are going to explore those ideas and how they appear in Lean:

2.2. Simple Type Theory

In simple type theory, every expression has a type. We start with one or more "small types" also called the type of individuals [6]. This type corresponds to `Type 0` in Lean; we will explain the number 0 later on. We can then build new types from existing ones. If α and β are types, then the type of all functions type from α to β is $\alpha \to \beta$ (Lean syntax allows the use of unicode symbols and numeric subscript, for example λ φ ψ $\circ$ $\mathbb{A}_1$ $\mathbb{A}_2$) and their product type is $\alpha \times \beta$.

Lean uses Church's simply typed lambda calculus with all three constructions:

```
variables (n : ℕ) (f : ℕ → ℕ → ℕ)        -- typed variable

def somefunction : ℕ → ℕ := λ m: ℕ  ,      -- function abstraction
                            f n m       -- function application
                            -- application associate left
```

2.3. Dependent Type Theory

Dependent type theory extends type theory by treating types as first-class citizens. This means that each type is an object and thus in turn has a type. Types like natural numbers

$\mathbb{N}$, integers $\mathbb{Z}$ and `bool` are objects of the type of small types `Type 0` (or in short `Type`). The type `Type 0` is an element of the type `Type 1`, which is an element of type `Type 2` and so on in an infinite hierarchy.

Types that contain other types are called *universes* and the natural number defining their position in the hierarchical order is called *universe level*.

Each universe is closed under function and product types. Another word to define a universe is `Sort`. However, `Sort u` is an element of the universe `Type u`, which means `Sort u+1` is the same universe as `Type u`. We will see the importance of the two keywords, when we discuss propositions.

We can declare types of any universe:

```
universe u
variables (A B : Type u)
variable  (S : Sort)
#check A                    -- returns Type (u+1)
#check S                    -- returns Type or Type 0
     -- #check is used to get the type of an expression.
```

In dependent type theory, as the name suggests, types can be *depend* on parameters. This feature can be illustrated in two main examples: Pi-types and Sigma-types:

Pi-types are generalizations of the function type in simple type theory. Given `(A: Type)` and a function `(B: A → Type)`, we can think of `B` as a product of an A-indexed family of types, we can construct the Pi-type `(Π a : A, B a)` or mathematically:

$$\Pi_{a:A}\ B_a \ = \{f : A \to \bigcup_{a:A} B_a \mid f(a) \in B_a\} \ \subseteq \left(\bigcup_{a:A} B_a\right)^A$$

This obviously includes the type of functions `A → B` if we consider `B` a constant function (independent from `A`) or `(Π a : α, B)`:

$$\Pi_{a:A}\ B \ = \{f : A \to B \mid f(a) \in B\} = A \to B$$

Sigma-types are generalizations of the product type in simple type theory. Given `(A: Type )` and a function `(B: A → Type)`, we can think of `B` as a sum of an A-indexed family of types, we can construct the Sigma-type `(Σ a : A, B a)` or mathematically:

$$\Sigma_{a:A}\ B_a \ = \{(a\ ,\ b) \mid a \in A\ \ b \in B_a\} \ \subseteq A \times \bigcup_{a:A} B_a$$

This in turn contains the product type $A \times B$ if we consider `B` a constant function (independent from A) or `(Σ a : A, B)`:

$$\Sigma_{a:A}\ B \ = \{(a\ ,\ b) \mid a \in A\ \ b \in B\} = A \times B$$

2.4. Propositions as Types

Lean's type checker depends on a central idea in simple type theory. A proof in type theory is nothing else than a program of lambda calculus. This is an embodiment of the Curry-Howard-isomorphism, which connects the following concepts [2]:

propositions	⇔	types
proofs	⇔	programs
normalization of proofs	⇔	evaluations or programs

Accordingly, we read `(a : α)` for any type `α` as "a is a proof of `α`". To make use of this, Lean considers any logical proposition as a type. This type is either empty or has one element, which means all proofs of a given theorem are "definitionally equivalent". The universe of all propositions is called `Prop`, which is a syntactic sugar for `Sort 0`, the bottom of the universes hierarchy. Here we can see the reason for using two keywords to declare a universe. Take the following example:

```
class has_add (α : Type u) := (add : α → α → α)

instance nat_has_add : has_add ℕ :=
{                   -- This is obviously wrong, but it works.
    add := λ n m , n
}

instance p_has_add (p : Prop): has_add p :=
{                   -- This does not work.
    add := λ pr₁ pr₂ , pr₁
}
```

Disregarding the word `class` and `instance` for now, this says that we can define an `add` function on any type `α`. However, it does not work for propositions, because a proposition `p` is an element of `Prop` or `Sort 0`, which does not match any `Type u` for any natural number `u`.

The universe `Prop` is, just like all universes, closed under the function type and the product type.

The function type `(f : p → q)` corresponds to logical implication. Given a proof (an element) of a proposition (a type) `(h : p)`, the application of `f` to `h` yields `f h : q`, a proof of `q`.

The product type, on the other hand, corresponds to logical conjunction.
Given `h₁ : p` and `h₂ : q`, then `⟨h₁ , h₂⟩ : p ∧ q`. The logical disjunction is defined as an inductive type:

```
inductive or (a b : Prop) : Prop
    | inl (h : a) : or
    | inr (h : b) : or
```

The negation is simply `¬ p := p → false`, where `false` is defined as an empty inductive type and `true` as an inductive type with one element.

2.5. Constructing a Proof in Lean

Lean allows two ways to construct a proof. The first way is functional, where the user provides an element of the required type "directly" using predefined functions and assumptions:

```
def n : ℕ := nat.add 5 3 -- using predefined function nat.add
lemma and_intro {p q: Prop} (h₁: p) (h₂: q) : p ∧ q :=
   and.intro h₁ h₂       -- using the given assumptions h₁ and h₂.
```

The only difference between a `definition` and a `lemma` (or a `theorem`) is that the type checker (in Lean it is called elaborator) would not unfold a lemma (or a theorem), because proofs are definitionally equivalent.

The second way is imperative using the so-called "tactic mode", where we can use different tactics to change the "goal", in order to make it easier to construct or to make Lean prove on its own.

```
theorem and_to_p {p q : Prop} (h : p ∧ q) : p :=
    begin
        cases h with h₁ h₂,     -- cases is a form of
                                -- pattern matching
        exact h₁
    end
```

The curly brackets {} indicate implicit parameters, that can be inferred from other parameters or the context. We can provide implicit parameter by writing @ before the name of the function.

Tactic mode can be accessed using `begin ... end` or the keyword `by` for one tactic and `by {...}` for a series of tactics. We can use tactics in place of any expression. We will introduce some other tactics as they come along.

3. Introduction to Category Theory and its Lean-Formalization

The mathematical definitions and theorems in this chapter are based on the chapter "Basics of Category Theory" in [4].

3.1. Categories

A category $\mathbb{C}$ consists of a class $\mathbb{O}$ of objects and a class $\mathbb{M}$ of morphisms (sometimes called arrows or maps) between those objects. For each pair of objects (A, B) there is a class of morphisms $Hom(A, B)$. For each morphism $f \in Hom(A, B)$ we call A the domain and B the codomain of f, and we write $f : A \to B$. The morphisms of a category must satisfy the following conditions:

1. For each object A there exists a morphism id_A, which starts and ends in A, and
2. morphisms $f : A \to B$ and $g : B \to C$ can be composed to a new morphism $g \circ f : A \to C$ respecting the following conditions:
   a) Identity: $f \circ id_A = f = id_B \circ f$
   b) Associativity: If $f : A \to B$, $g : B \to C$ and $h : C \to D$, then $(h \circ g) \circ f = h \circ (g \circ f)$.

Categories are modeled in Lean's mathlib, using *type classes*. A *type class* represents a family of types, that share a common feature. Elements of a type class, which are declared as an `instance` (or within square bracket) are stored in something called *instances table* and are used by the *elaborator* implicitly, when fields of the *type class* are called. This allows us, for example, to define a morphism `f: X ⟶ Y` and the *elaborator* implicitly infers the category based on `X` and `Y`.

The class `has_hom` comprises objects, that can be combined with an operator `hom`. Objects and morphisms do not have to reside in the same universe level, thus `has_hom` must reside in a universe, that contains both types. The type of objects is an element of `Type u` (to avoid having a proposition as the class of objects see 2.4) and the type of morphisms is an element `Sort v`, which is in turn an element of `Type v`, then the class `has_hom` must reside in `max u v`:

```
universes v u
class has_hom (obj : Type u) : Type (max u v) :=
  (hom : obj → obj → Sort v)
```

The class `category_struct`, which extends `has_hom`, has conditions 1 and 2:

```
class category_struct (obj : Type u)
        extends has_hom.{v} obj : Type (max u v) :=
          -- .{v} tells Lean, the universe level of hom.
   (id      : Π X : obj, hom X X)
   (comp    : Π {X Y Z : obj}, (X ⟶ Y) → (Y ⟶ Z) → (X ⟶ Z))
```

The long arrows ⟶ are infix notation for morphisms. And lastly, the class `category`, which extends `category_struct`, includes the conditions 2a and 2b:

```
class category (obj : Type u)
extends category_struct.{v} obj : Type (max u v) :=
   (id_comp' : ∀ {X Y : obj} (f : hom X Y), 𝟙 X ≫ f = f . obviously)
   (comp_id' : ∀ {X Y : obj} (f : hom X Y), f ≫ 𝟙 Y = f . obviously)
   (assoc'   : ∀ {W X Y Z : obj} (
      f : hom W X) (g : hom X Y) (h : hom Y Z),
         (f ≫ g) ≫ h = f ≫ (g ≫ h) . obviously)
```

The composition notation `f ≫ g` used here is read g after `f`. However, to avoid confusion with mathematical notation, we replace it with a local notation ⊚, with which the previous example is written `g ⊚ f`.
𝟙 is the *id*-morphism notation, with a small syntactical difference to the `id` function. In `id` the type is an implicit parameter, where in 𝟙 it is an explicit one. Thus, in category set (see 3.2):

```
   id = 𝟙 A = @id A
```

Remark. *An important concept presented in the definition of category is the use of the tactic `obviously`. This allows the user to define a category by just providing the fields from the upper classes `has_hom` and `category_struct` and Lean would attempt to automatically prove the conditions, marked with `. obviously`.*

3.1.1. Special Morphisms

While there are many kinds of special morphisms, we focus on two, that will be used often in this work:

Monos and Epis

A morphism $f : A \to B$ is called

- left-cancellable or mono, if:

$$\forall g\ h : C \to A.\ f \circ g = f \circ h \Rightarrow g = h$$

- right-cancellable or epi, if:

$$\forall g\ h : B \to C.\ g \circ f = h \circ f \Rightarrow g = h$$

Remark. *Monomorphisms are drawn with tailed arrows* $\rightarrowtail$*, while epimorphisms with two headed arrows* $\twoheadrightarrow$*.*

Monos and epis are also modeled in Lean as *type classes*. However, since the composition notation is reversed in the modeling category theory, with respect to $\gg$ monos become right-cancellable and epis left-cancellable.

```
class epi  (f : X ⟶ Y) : Prop :=
(left_cancellation : Π {Z : C} (g h : Y ⟶ Z)
                       (w : f ≫ g = f ≫ h), g = h)
class mono (f : X ⟶ Y) : Prop :=
(right_cancellation : Π {Z : C} (g h : Z ⟶ X)
                       (w : g ≫ f = h ≫ f), g = h)
```

To minimize the confusion around this point we use an `abbreviation`[1]:

```
abbreviation left_cancel {C : Type v} [category C] {A B C : C}
   (f : A ⟶ B) [mo : mono f] {g h : C ⟶ A} :
   (f ⊚ g = f ⊚ h) → g = h := assume m, (cancel_mono f).1 m

abbreviation right_cancel {C : Type v} [category C] {A B C : C}
   (f : A ⟶ B) [ep: epi f] {g h : B ⟶ C} :
   (g ⊚ f = h ⊚ f) → g = h := assume e, (cancel_epi f).1 e
```

Here we see an example of a *type class*. Defining the *type class instance* `[category C]` allows us to use the morphism notation between elements of `C` and it tells Lean that these morphisms satisfy the conditions of a category. Later when we use this abbreviation, the *elaborator* looks for a stored instance of this category and use it automatically.

[1] A syntactic sugar equivalent to lemma and theorem, mostly used to refer to another lemma in a different way.

3.1.2. Functors

Functors relate different categories in a similar way, as maps relates different sets. However, a functor must satisfy these conditions:

Let F be a functor between **C** and **D**, then:

1. for each object A in **C**, there is a unique object $F(A)$ in **D**
2. to each **C**-morphism $f : A \to B$, there is a unique **D**-morphism $Ff : F(A) \to F(B)$, such that:
   a) $Fid_A = id_{F(A)}$
   b) $F(f \circ g) = Ff \circ Fg$ for all $g : C \to A$

Functors, along with these conditions are modeled as structures in a very straightforward way:

```
structure functor (C : Type u₁) [category.{v₁} C]
                  (D : Type u₂) [category.{v₂} D]
                    : Type (max v₁ v₂ u₁ u₂) :=
(obj        : C → D)
(map        : Π {X Y : C}, (X ⟶ Y) → ((obj X) ⟶ (obj Y)))
(map_id'    : ∀ (X : C), map (𝟙 X) = 𝟙 (obj X) . obviously)
(map_comp'  : ∀ {X Y Z : C} (f : X ⟶ Y) (g : Y ⟶ Z),
               map (f ≫ g) = (map f) ≫ (map g)      . obviously)
```

The notation for functors is:

```
  infixr ` ⥤ `:26 := functor
```

The letter 'r' at the end of `infixr` stands for right associative and `:26` indicates the precedence.

3.2. The Category of Sets

The category *Set* has as objects all sets and as morphisms all maps between sets. However, in Lean, sets correspond to types, and category *Set* becomes the category `types`, which is modeled like this:

```
instance types : large_category (Sort u) :=
{  hom     := λ a b, (a → b),
   id      := λ a, id,
   comp    := λ _ _ _ f g, g ∘ f }        -- An underscore is to tell the
                                    -- system to fill in automatically.
```

A large category, as explained in *mathlib* documentation, is a category, whose objects reside in one universe level higher than the universe level of the morphisms, which allows for categories like the category *Set* and the category of groups, etc.

Remark. *Notice that we do not need to provide the fields of the class* `category`*, because they were marked with* `"obviously"` *(see 3.1) and in this case, they are simple enough for the tactic* `obviously` *to prove.*

Lemma 3.2.1. *In the category Set, a morphism is mono iff (if and only if) it is injective and it is epi iff it is surjective. This lemma is proven in Lean's mathlib:*

```
lemma mono_iff_injective {X Y : Type u} (f : X ⟶ Y) :
        mono f ↔ function.injective f := ... -- omitted
lemma epi_iff_surjective {X Y : Type u} (f : X ⟶ Y) :
        epi f ↔ function.surjective f := ...
```

For the next two chapters, let us define the following variables:

```
variables {F : Type u ⥤ Type u}
      -- Endofunctor on the category of types (sets)
        {A B : Type u}
```

An interesting property of set-endofunctors is presented in the following lemma:

Lemma 3.2.2. *Let F be a set-endofunctor, $X \neq \emptyset$ and $f : X \to Y$ an injective map, then so is Ff.*

Proof. This is the first proof, we look at in Lean and it introduces some interesting tactics to facilitate the proof:

- We can define an object of a structure using (among other ways) curly brackets with named parameters.
- `let x := t` is used to make `x` a syntactical abbreviation of the `t` to `x`, i.e. every occurrence of `x` in `v` would be replaced by `t`. On the other hand, `have x : T := t` would hide its `t` and only holds its type `T`.
- `calc` is a tactic that allows the user to write the proof as a series of mathematical equations with the proof of each step written after a colon.
- `rfl` is short for the equation reflexive property `eq.refl _`.

```
lemma mono_preserving_functor [inhabited A]
        (f : A ⟶ B) (inj : injective f) : mono (F.map f) :=
{ right_cancellation :=
    begin
      assume (Z:Type u) (Fg Fh : Z ⟶ F.obj A)
            (w : (F.map f) ∘ Fg = (F.map f) ∘ Fh),
      let inv : B ⟶ A := inv_fun f,
      have id_inv : inv ∘ f = id := funext (left_inverse_inv_fun inj),
```

```
    calc
    Fg  = (@id (F.obj A)) ∘ Fg             : rfl
    ... = (𝟙 (F.obj A)) ∘ Fg               : rfl
    ... = (F.map (𝟙 A)) ∘ Fg               : by rw functor.map_id'
    ... = (F.map (@id A)) ∘ Fg             : rfl
    ... = (F.map (inv ∘ f)) ∘ Fg           : by rw id_inv
    ... = (F.map (inv ⊚ f)) ∘ Fg           : rfl
    ... = ((F.map inv) ⊚ (F.map f)) ∘ Fg   : by rw functor.map_comp
    ... = (F.map inv) ∘ ((F.map f) ∘ Fg)   : rfl
    ... = (F.map inv) ∘ ((F.map f) ⊚ Fg)   : rfl
    ... = (F.map inv) ∘ ((F.map f) ⊚ Fh)   : by rw w
    ... = (F.map inv) ∘ ((F.map f) ∘ Fh)   : rfl
    ... = ((F.map inv) ∘ (F.map f)) ∘ Fh   : rfl
    ... = ((F.map inv) ⊚ (F.map f)) ∘ Fh   : rfl
    ... = (F.map (inv ⊚ f)) ∘ Fh           : by rw functor.map_comp
    ... = (F.map (inv ∘ f)) ∘ Fh           : rfl
    ... = (F.map (@id A)) ∘ Fh             : by rw id_inv
    ... = (F.map (𝟙 A)) ∘ Fh               : rfl
    ... = (𝟙 (F.obj A)) ∘ Fh               : by rw functor.map_id'
    ... = (@id (F.obj A)) ∘ Fh             : rfl
    ... = Fh                               : rfl
  end
}
```

□

3.3. Diagram Lemmas

Before discussing the diagram lemmas, we need a few definitions. For $f : A \to B$ we define:

- the image of $S \subseteq A$ under f as

$$f[S] := \{f(a) | a \in S\}$$

  which is predefined in Lean as:

```
def image {α : Type u} {β : Type v} (f : α → β) (s : set α)
      : set β := {b | ∃ a, a ∈ s ∧ f a = b}
```

- the *kernel* of f:

$$ker\ f := \{(a_1, a_2) \in A \times A | f(a_1) = f(a_2)\}$$

  The curried form of this relation:

```
def kern (f : A ⟶ B) : A → A → Prop :=
      λ a₁ a₂ , f a₁ = f a₂
```

- A map can be defined by its graph:

$$G(f) := \{(a, f(a)) | a \in A\} \quad = \quad \{(a,b) | a \in A \wedge b \in B \wedge \exists! a.\ f(a) = b\}$$

```
noncomputable def graph_to_map
   (G :  A → B → Prop)
   (h : ∀ a : A , ∃! b : B, G a b) : A → B :=
      λ a , some (h a)
```

This method is marked `noncomputable`, because it depends on the axiom of choice `some`, which is not always computable.

Lemma 3.3.1 (Diagram-Lemma)**.** *1. Let $f : A \twoheadrightarrow B$ be a surjective map and $g : A \to C$ arbitrary. There is a unique map $h : B \to C$ with $h \circ f = g$, iff $Kern f \subseteq Kern g$.*

$$\begin{tikzcd} A \arrow[r, two heads, "f"] \arrow[dr, "g"'] & B \arrow[d, dotted, "\exists! h"] \\ & C \end{tikzcd}$$

2. Let $f : A \rightarrowtail B$ be an injective map and $g : C \to A$ an arbitrary map. Then there is a unique map $h : C \to B$ with $f \circ h = g$, iff $image\, g \subseteq image\, f$.

$$\begin{tikzcd} A \arrow[r, tail, "f"] & B \\ & C \arrow[ul, "g"] \arrow[u, dotted, "\exists! h"'] \end{tikzcd}$$

Proof. The proof is done in two parts:

1. When f is surjective: The proof is fairly long. We will present a snippet of the code here and the rest can be found in Lean-Code "Diagram Lemma" in the appendix.
   The new concepts that appear in the proof are:
   - `cases ex`, we have already seen an example of the `cases` tactic (see 2.5). However, here it uses the axiom of choice, as explained in the comment.
   - The infix notation $\rhd$ stands for `eq.subst`.

```
lemma diagram_surjective (f : A ⟶ B) (g : A ⟶ C)
                   (sur: surjective f)
      : (∃! h : B ⟶ C , h ∘ f = g) ↔ (sub_kern f g)
   := iff.intro -- splits the proof into two directions.
   begin
      assume ex,

      cases ex with h spec,
      -- h : B ⟶ C
      -- spec : h ∘ f = g ∧ ∀ h₁ , h₁∘ f = g → h₁ = h,
      exact spec.1 ▹ kern_comp f h
   end
   begin
      let h : B → C := λ b : B, g (surj_inv sur b),
      ...
      have s4 : ∀ a , h (f a) = g a := ...,
      have s6 : ∀ h₂ :B → C , h₂ ∘ f = g → h₂ = h :=  ...,

      exact exists_unique.intro h (funext s4) s6,
   end
```

2. When f is injective:

```
lemma diagram_injective (f: B ⟶ A) (g: C ⟶ A)
                   (inj : injective f)
      : (∃! h : C ⟶ B , f ∘ h = g) ↔ (range g ⊆ range f)
:= iff.intro
begin
   tidy -- The → direction is relatively easy
       -- Lean can prove it alone using the tactic "tidy".
end
begin
   assume im,
   show ∃ h : C → B , f ∘ h = g ∧ ∀ h₁ , f ∘ h₁ = g → h₁ = h,
   -- defining the graph
   let G : C → B → Prop := λ c b , g c = f b,
   ...
   have G2 :  ∀ c : C , ∃! b : B, G c b := ...
   let h : C ⟶ B := graph_to_map G G2,
   ...
end
```

□

3.4. Orthogonality

A very important structure in category theory is given by so-called E-M-squares. These are commutative squares (see diagram 3.4), where $f \circ e = m \circ g$, typically with a diagonal d, as shown in this diagram:

$$\begin{tikzcd} X \arrow[r, two heads, "e"] \arrow[d, "g"'] & Y \arrow[d, "f"] \arrow[dl, dotted, "\exists!\, d"'] \\ Z \arrow[r, tail, "m"'] & U \end{tikzcd}$$

Remark. *A diagram is called commutative if all possible compositions, with the same domain and codomain, are equal.*

Based on this structure, we can build a few observations, presented in the following lemmas:

Lemma 3.4.1. *Given a category* **C** *and the E-M-Square, if:*

1. *e is epi and the upper triangle commutes or*
2. *m is mono and the lower triangle commutes,*

then d makes both triangles commute and d is unique.

Proof. We start with the upper triangle, that is to show:

$$epi\ e \wedge d \circ e = g \Rightarrow m \circ d = f \wedge \forall d_1 : Y \to Z.\ d_1 \circ e = g \Rightarrow d_1 = d$$

```
lemma commutative_triangles_epi {C : Type v} [category C]
                       {X Y Z U : C}
            (e : X ⟶ Y) (f : Y ⟶ U)
            (g : X ⟶ Z) (m : Z ⟶ U)
            (h : f ⊚ e = m ⊚ g) (d: Y ⟶ Z) [epi e] (g_ed: g = d ⊚ e)
      : f = m ⊚ d ∧ ∀ d₁ : Y ⟶ Z , g = d₁ ⊚ e → d₁ = d  := ...
```

The other direction is quite similar:

$$mono\ m \wedge m \circ d = f \Rightarrow d \circ e = g \wedge \forall d_1 : Y \to Z.\ m \circ d_1 = f \Rightarrow d_1 = d$$

```
lemma commutative_triangles_mono {C : Type v} [category C]
                          {X Y Z U : C}
            (e : X ⟶ Y) (f : Y ⟶ U)
            (g : X ⟶ Z) (m : Z ⟶ U)
            (h : f ⊚ e = m ⊚ g) (d: Y ⟶ Z) [mono m] (f_md: f = m ⊚ d)
      : g = d ⊚ e ∧ ∀ d₁ : Y ⟶ Z , f = m ⊚ d₁ → d₁ = d := ...
```

□

The following lemma, with some restrictions, can be applied to all categories. We prove it, however, only for the category *Set*, using the fact, that monomorphisms are the same as injective maps and epimorphisms are the same as surjective maps in the category *Set*.

Lemma 3.4.2. *In the category Set, given an E-M-square, where e is epi and m is mono, there exists a unique diagonal d, which makes the triangles commute.*

Proof. The proof is mathematically simple using the diagram lemmas. However it was relatively long in Lean (see A.1.2).

```
lemma E_M_square {X Y Z U : Type u}
      (e : X ⟶ Y) (ep : epi e)
      (f : Y ⟶ U) (g : X ⟶ Z)
      (m : Z ⟶ U) (mo : mono m)
      (h : f ⊚ e = m ⊚ g) :
      ∃! d : Y ⟶ Z , (g = d ⊚ e ∧ f = m ⊚ d) :=
begin
   ...
   have diagram_sur : _ :=
      diagram_surjective e g ((epi_iff_surjective e).1 ep),
   have diagram_inj : _ :=
      diagram_injective m f ((mono_iff_injective m).1 mo),
   ...
   have range_f_m : range f ⊆ range m :=
      calc range f = range (f ⊚ e)    : eq_range_if_surjective e f sur
            ...    = range (m ⊚ g)    : by rw h
            ...    ⊆ range m         : range_comp_subset_range g m,
            -- See  A.1.2 for definition of eq_range_if_surjective.
   ...
   have kern_e_g : sub_kern e g :=
      sub_kern_if_injective e f g m h inj,
            -- See  A.1.2 for definition of sub_kern_if_injective.
   ...
end
```

□

3.5. Limits and Colimits

Limits and Colimits are key concepts of category theory. The two concepts can be summarized as dual to each other[7].

Remark. *A structure in category theory is called dual of another structure if they can be obtained from each other by reversing the arrows and the order of compositions.*

In a category $\mathbb{C}$, a **limit** over a family of objects $(A_i)_{i \in I}$, with the morphisms between them $d_{ij} : A_i \to A_j$, is an object V in $\mathbb{C}$ with morphisms $(f_i : V \to A_i)_{i \in I}$, such that $d_{ij} \circ f_i = f_j$ for all $i, j \in I$ and for any object W (competitor) in $\mathbb{C}$ with morphisms $(g_i : W \to A_i)_{i \in I}$, where $d_{ij} \circ g_i = g_j$ for all $i, j \in I$, there exists a unique morphism $h : W \to V$, where:

$$\forall i \in I. \quad g_i \circ h = f_i$$

$$\begin{tikzcd}
 & V \arrow[r, dotted, "\exists! h"] \arrow[dl, "f_j"'] \arrow[d, "f_i"] & W \arrow[dll, "g_j"'] \arrow[dl, "g_i"] \\
A_j & A_i &
\end{tikzcd}$$

A **colimit** over a family of objects $(A_i)_{i \in I}$ and the morphisms between them is an object U in $\mathbb{C}$ with morphisms $(f_i : A_i \to U)_{i \in I}$, such that $f_j \circ d_{ij} = f_i$ for all $i, j \in I$ and for each object Q (competitor) in $\mathbb{C}$ with morphisms $(g_i : A_i \to Q)_{i \in I}$, where $g_j \circ d_{ij} = g_i$ for all $i, j \in I$, there exists a unique morphism $h : U \to Q$, where:

$$\forall i \in I. \quad h \circ f_i = g_i$$

$$\begin{tikzcd}
 & U \arrow[r, dotted, "\exists! h"] & Q \\
A_j \arrow[ur, "f_j"] \arrow[urr, "g_j"'] & A_i \arrow[u, "f_i"] \arrow[ur, "g_i"'] &
\end{tikzcd}$$

Special Limits

3.5.1. Equalizer

Definition 1 (Equalizer)**.** *Let $(f\, g : A \to B)$ be two parallel[2] morphisms. A morphism $e : E \to A$ is an equalizer of f and g, if:*

- *$f \circ e = g \circ e$ and*
- *For each object Q in $\mathbb{C}$ with morphism $q : Q \to A$, such that $f \circ q = g \circ q$, there exists a unique morphism $h : Q \to E$ with $q = e \circ h$.*

$$\begin{tikzcd} E \arrow[r, "e"] & A \arrow[r, shift left, "f"] \arrow[r, shift right, "g"'] & B \\ Q \arrow[u, dotted, "\exists! h"] \arrow[ur, dashed, "q"'] & & \end{tikzcd}$$

The modelling of equalizers over two parallel morphisms is straight forward.

```
def is_equalizer {ℂ : Type u} [category ℂ]
    {A B E : ℂ} (f g : A ⟶ B)
    (e : E ⟶ A) : Prop :=
    f ⊚ e = g ⊚ e ∧
    Π {Q : ℂ} (q : Q ⟶ A),
        f ⊚ q = g ⊚ q → ∃! h : Q ⟶ E, q = e ⊚ h
```

Lemma 3.5.1. *The equalizer in the category Set is a subset E of A, which equalizes the two maps, i.e.*

$$E := \{a \in A \mid f(a) = g(a)\}$$

along with its inclusion in A.

Proof. To begin, we need to define the equalizer set and the inclusion:

```
def eqaulizer_set : set A := λ a, f a = g a
def inclusion {A : Type u} (S : set A)
    : S → A := λ s , s
notation S ‘ ↪ ‘ A := @inclusion A S
```

The proof from this point is very simple:
Notice that we can use the angle brackets ⟨⟩ instead of `and.intro`. We can do that, because and is defined as a structure and we can define object of a structure using angle brackets with ordered parameters.

[2]Morphisms with the same domain and codomain

```
lemma eqaulizer_set_is_equalizer :
    is_equalizer f g (eqaulizer_set f g ↪ A) :=
    let E := eqaulizer_set f g in
    let e := E ↪ A in
    ⟨    -- These brackets replace ‘and.intro‘
        have elements : ∀ a, (f ∘ e) a = (g ∘ e) a :=
                λ a, a.property,
        funext elements
        ,
        begin
            intros Q q fq_gq,
            ...
            let h : Q → E := λ b, ⟨q b, s1 b⟩,
            ...
            -- to show ∃! h : Q ⟶  E, q = e ⊚ h
            use h,       -- use h as example of existence
            split,       -- split the existence condition and
                         -- the uniqueness condition
            exact ...,
            intros h₁ spec_h₁,
            tidy,     -- simplifies the goal to h₁ x = h x for x ∈ Q
            ...
        end
    ⟩
```

□

3.5.2. Product

Definition 2. *Let A and B be objects in $\mathbb{C}$. An object P together with morphisms $\pi_A : P \to A$ and $\pi_B : P \to B$ is called product of A and B, if for each other object Q with morphisms $q_A : Q \to A$ and $q_B : Q \to B$ there exists a unique morphism $p : Q \to P$, such that $q_A = \pi_A \circ p$ and $q_B = \pi_B \circ p$*

$$\begin{tikzcd}
Q \arrow[rr, dotted, "\exists! p"] \arrow[dr, dashed, "q_A"'] \arrow[drrr, dashed, "q_B"] & & P \arrow[dl, "\pi_A"] \arrow[dr, "\pi_B"] & \\
& A & & B
\end{tikzcd}$$

The modeling of the product is also almost word for word:

```
def is_product {X : Type v} [category X]
    (A B : X)
    {P : X} (π₁ : P ⟶ A) (π₂ : P ⟶ B): Prop :=
    Π {Q : X} (q₁ : Q ⟶ A) (q₂ : Q ⟶ B),
      ∃! p : Q ⟶ P, q₁ = π₁ ⊚ p ∧ q₂ = π₂ ⊚ p
```

Lemma 3.5.2. *The product in the category Set is the cartesian product:*

$$A \times B = \{(a, b) \mid a \in A \text{ and } b \in B\}$$

along with the projection maps π_A and π_B

Proof. Lean can do most of the proof by itself except for constructing the unique map $p : Q \to P$. However, once provided with the correct morphism the equality and the uniqueness can be proven automatically by the tactic `tidy`.

```
lemma cartesian_product_is_product :
    is_product A B prod.fst prod.snd :=
    begin
        intros Q q₁ q₂,
        let p : Q → (A × B) := λ k, ⟨q₁ k,q₂ k⟩,
        use p,
        tidy,
    end
```

□

In this context, we can introduce the concept of *jointly mono*.

Definition 3 (jointly mono)**.** *A family of morphisms $(m_i : P \to A_i)_{i \in I}$ is called jointly mono, if:*

$$\forall Q.\ \forall f, g : Q \to P.\ \ (\forall i \in I. m_i \circ f = m_i \circ g) \Rightarrow f = g$$

The following lemma can be proven in any category and indeed we did that. The proof can be found in the A.1.4. However, proving it just for category Set, presents an interesting challenge in Lean, which is **structural equality**.

Lemma 3.5.3. *The projections from the Cartesian product to the components are jointly mono.*

Proof. Mathematically, the proof is very obvious. However, proving the equality of two objects of a given structure in Lean required some search and the use of tactics. The tactic `ext1` (among other use cases) splits the goal of equality between two structures into separate goals for each of the structures's components:

```
lemma jointly_mono_set {Q : Type u} {f g: Q → (A × B)}
        (h1 : prod.fst ∘ f = prod.fst ∘ g)
        (h2 : prod.snd ∘ f = prod.snd ∘ g):
        f = g :=
    have elements : ∀ q : Q , f q = g q :=
        assume q,
        have s1 : prod.fst (f q) = prod.fst (g q) :=
           have s11 : (prod.fst ∘ f) q = (prod.fst ∘ g) q := by rw h1,
           s11,
        have s2 : prod.snd (f q) = prod.snd (g q) :=
           have s21 : (prod.snd ∘ f) q = (prod.snd ∘ g) q := by rw h2,
           s21,
        by {
            ext1,
            exact s1, exact s2
        },
    funext elements
```

□

Remark. *Another point worth mentioning here is the need for the extra step (as in* s11 *and* s21*), which we will discuss later in chapter 5.*

3.5.3. Pullback

Definition 4. *Let $f_1 : A_1 \to B$ and $f_2 : A_2 \to B$ be a sink*[3]*. An object P together with the morphisms $p_1 : P \to A_1$ and $p_2 : P \to A_2$ is called pullback of the f_1 and f_2, if*

- *$f_1 \circ p_1 = f_2 \circ p_2$ and*
- *for each object Q with source*[4] *$q_1 : Q \to A_1$ and $q_2 : Q \to A_2$, such that:*

$$f_1 \circ q_1 = f_2 \circ q_2$$

there exists a unique morphism $h : Q \to P$ with $p_1 \circ h = q_1$ and $p_2 \circ h = q_2$.

$$\begin{tikzcd}
 & & A_1 \arrow[dr, "f_1"] & \\
Q \arrow[urr, dashed, bend left, "q_1"] \arrow[r, dotted, "\exists! h"] \arrow[drr, dashed, bend right, "q_2"'] & P \arrow[ur, "p_1"] \arrow[dr, "p_2"] & & B \\
 & & A_2 \arrow[ur, "f_2"'] &
\end{tikzcd}$$

[3]A sink is a family of morphisms with the same codomain.
[4]A source is a family of morphisms with the same domain.

```
def is_pullback {X : Type v} [category X]
    {A₁ A₂ B : X}
    (f : A₁ ⟶ B) (g : A₂ ⟶ B)
    {P : X} (p₁ : P ⟶ A₁) (p₂ : P ⟶ A₂): Prop :=
    f ⊚ p₁ = g ⊚ p₂ ∧
    Π {Q : X} (q₁ : Q ⟶ A₁) (q₂ : Q ⟶ A₂),
        f ⊚ q₁ = g ⊚ q₂ →
            ∃! h : Q ⟶ P, q₁ = p₁ ⊚ h ∧ q₂ = p₂ ⊚ h
```

The pullback can be constructed in two steps. First, taking the product $(A_1 \times A_2)$ and then taking the equalizer of the morphisms $(f_i \circ \pi_i : (A_1 \times A_2) \to B)$. We can prove this last statement in the following lemma.

Lemma 3.5.4. *Let A_1, A_2 and B be objects in $\mathbb{C}$ along with morphisms $f_1 : A_1 \to B$ and $f_2 : A_2 \to B$, if both the product $A_1 \times A_2$ and the equalizer of $(f_1 \circ \pi_1)$ and $(f_2 \circ \pi_2)$ exist, then so does the pullback of f_1 and f_2 and it is the same as the aforementioned equalizer.*

Proof.

```
lemma equalizer_product_is_pullback
    {X : Type u} [category X]
    {A₁ A₂ B P E : X} (f : A₁ ⟶ B) (g : A₂ ⟶ B)
    (π₁ : P ⟶ A₁) (π₂ : P ⟶ A₂) (pr : is_product A₁ A₂ π₁ π₂)
    (e : E ⟶ P) (eqauliz : is_equalizer (f ⊚ π₁) (g ⊚ π₂) e) :
      is_pullback f g (π₁ ⊚ e) (π₂ ⊚ e)
    :=
    ⟨ begin
        tidy
     end
    ,
    begin
        intros Q q₁ q₂ fq₁_gq₂,
        let p : Q ⟶ P := some (pr  Q q₁ q₂),
        ...

        have eq_comp : f ⊚ π₁ ⊚ p  = g ⊚ π₂ ⊚ p :=
            calc f ⊚ π₁ ⊚ p   = f ⊚ (π₁ ⊚ p)   : by tidy
               -- No associativity is defined for ⊚
                     ...
                     ...         = g ⊚ (π₂ ⊚ p)  : by rw spec_p.2
                     ...         = g ⊚ π₂ ⊚ p    : by tidy,
        ...
    end ⟩
```

□

Special Colimits

3.5.4. Coequalizer

Definition 5 (Coequalizer)**.** *Let $f, g : A \to B$ be parallel morphisms. A morphism $c : B \to C$ is called coequalizer of the f and g, if*

- *$c \circ f = c \circ g$ and*
- *for each object Q with morphism $q : B \to Q$, where $q \circ f = q \circ g$, there exists a unique morphism $h : C \to Q$, such that $q = h \circ c$.*

$$\begin{tikzcd} A \arrow[r, shift left, "f"] \arrow[r, shift right, "g"'] & B \arrow[r, "c"] \arrow[dr, dashed, "q"'] & C \arrow[d, dotted, "\exists! h"] \\ & & Q \end{tikzcd}$$

```
def is_coequalizer {X : Type u} [category X]
    {A B C : X} (f g : A ⟶ B) (c : B ⟶ C) : Prop :=
    c ⊚ f = c ⊚ g ∧
    Π (Q : X) (q : B ⟶ Q),
      q ⊚ f = q ⊚ g →
      ∃! h : C ⟶ Q , h ⊚ c = q
```

To construct the coequalizer of $f_i : A \to B$ in category Set, we define the relation Θ as the smallest equivalence relation on B, generated from the relation:

$$R := \{(f(a), g(a)) | a \in A\} = \{(b_1, b_2) | \ b_1, b_2 \in B \ \ and \ \exists a \in A. \ f(a) = b_1 \wedge g(a) = b_2\}$$

The projection map $\pi_\Theta : B \to B/\Theta$ with $\pi_\Theta(b) = [b]_\Theta$ is the coequalizer of the f_i.

Constructing this, takes a few steps in Lean:

1. First, defining the relation R:

   ```
   def R : B → B → Prop := λ b₁ b₂ , ∃ a , f a = b₁ ∧ g a = b₂
   ```

2. Then, we can define the equivalence relation Θ generated from R:

   ```
   def Θ : B → B → Prop := eqv_gen (R f g)
   ```

   eqv_gen is a predefined relation, defined inductively to generate the smallest equivalence relation based on a given relation.

3. Next, we define a *setoid*, which is a set (type) equipped with an equivalence relation.

```
def Θ_setoid : setoid B := eqv_gen.setoid (R f g)
```

4. Having this *setoid* allows us to define the quotient set $B\backslash\Theta$ and the coequalizer map $\pi_\Theta : B \to B\backslash\Theta$

```
definition theta := quotient (Θ_setoid f g)

definition coequalizer : B → (theta f g) := λ b, ⟦b⟧
```

Proof.

- The first part of the proof is made simple by the predefined functions on quotients and on equivalence relations in Lean such as `quot.sound`, which asserts that:

$$\forall a, b.\ a\,R\,b \ \Rightarrow\ [\![a]\!] = [\![b]\!]$$

- The second part uses the diagram lemmas 3.3.1. However, proving that the kernel of the coequalizer π_Θ is a subset of the kernel of any competitor q requires tactics to work with *inductive types* in Lean. Therefore, we split the proof into two separate lemmas to concentrate on this very important structure in the second lemma.

```
lemma quot_is_coequalizer
    : is_coequalizer f g (coequalizer f g) :=
    ⟨
      have elements : ∀ a : A,
         ((coequalizer f g) ∘ f) a = ((coequalizer f g) ∘ g) a :=
        assume a,
        ...,
        have Θfg : Θ f g (f a) (g a) :=
                 eqv_gen.rel (f a) (g a) Rfg,
        quot.sound Θfg,
      funext elements
      ,
      begin
        intros Q q qfg,
        let co := (coequalizer f g),
        have sub_ker : sub_kern co q := coequalizer_kern f g Q q qfg,
        exact (diagram_surjective co q
              (quot_is_surjective f g)).elim_right sub_ker
      end
    ⟩
```

The function `rec_on` is defined automatically inside the namespace of every inductive type in Lean. We can use `apply inductive_type.rec_on` to split the goal into separate goals for each inductive case.

```
lemma coequalizer_kern :
    Π (Q : Type u) (q : B → Q),
    q ∘ f = q ∘ g → sub_kern (coequalizer f g) q :=
    begin
        intros Q q qfg,
        let co := (coequalizer f g),
        let ker := kern co,
        let ker_q := kern q,

        intros b₁ b₂ kb1b2,

        have quotb1b2 : ⟦b₁⟧ = ⟦b₂⟧  := kb1b2,
        let Θb1b2 : eqv_gen (R f g) b₁ b₂ :=
            @quotient.exact B (Θ_setoid f g)
            b₁ b₂
            quotb1b2,

        apply eqv_gen.rec_on Θb1b2,
        -- relation:
        exact ...,
        -- reflexive:
        exact (λ x, rfl),
        -- symmetric:
        exact (λ x y _ (h : q x = q y), eq.symm h),
        -- transitive:
        exact (λ x y z _ _ (h₁ : q x = q y) (h₂ : q y = q z),
                    eq.trans h₁ h₂),
    end
```

□

3.5.5. Sum

Definition 6 (Sum)**.** *Let A and B be objects in* **C**. *An object S together with morphisms $e_A : A \to S$ and $e_B : B \to S$ is called sum of the A and B (e_A and e_B are also called the canonical injections), if for each other object Q with morphisms $q_A : A \to Q$ and $q_B : B \to Q$ there exists a unique morphism $s : S \to Q$, so that $q_A = s \circ e_A$ and $q_B = s \circ e_B$.*

$$\begin{tikzcd}
& S \arrow[rr, dotted, "\exists! s"] & & Q \\
A \arrow[ur, "e_A"] \arrow[urrr, dashed, "q_A"'] & & B \arrow[ul, "e_B"'] \arrow[ur, dashed, "q_B"'] &
\end{tikzcd}$$

We start by showing, that the canonical injections are *jointly epi*, the dual concept of jointly mono, which is to say:

$$\forall Q.\ \forall f, g : S \to Q.\ \ (f \circ e_A = g \circ e_A \wedge f \circ e_B = g \circ e_B) \Rightarrow f = g$$

Lemma 3.5.5. *If an object S along with $e_A : A \to S$ and $e_B : B \to S$ is a sum of A and B, then e_A and e_B are jointly epi.*

Proof. The proof is rather obvious. However, we use it here, to show a very useful tactic `rw`. We use `rw` here to change the goal to simplify the proof:

```
lemma jointly_epi_cat {X : Type v} [category X]
    {A B S Q : X} (e₁ : A ⟶ S) (e₂ : B ⟶ S)
    (is_sm : is_sum S e₁ e₂) (f g: S ⟶ Q)
        (h1 : f ⊚ e₁ = g ⊚ e₁)
        (h2 : f ⊚ e₂ = g ⊚ e₂)
    : f = g :=
    begin
      exact ...
        have g_s : g = s:= ...,
        -- The goal here is f = g
        rw g_s,
        -- Here the goal becomes f = s
        -- where s is the unique morphism obtained
        -- from the sum to its competitor.
        exact ...
    end
```

□

Lemma 3.5.6. *The disjoint union of the sets A_1 and A_2, i.e.:*

$$A_1 \uplus A_2 = \{(1, a) |\ \ a \in A_1\} \cup \{(2, a) |\ \ a \in A_2\}$$

along with maps $e_1 : A_1 \to S$ and $e_2 : A_2 \to S$ defined as $e_i : a \mapsto (i, a)$ is the sum of A_1 and A_2 in the category Set.

Proof. The disjoint union is defined inductively in lean:

```
inductive sum (α : Type u) (β : Type v)
   | inl {} (val : α) : sum
   | inr {} (val : β) : sum
```

The {} notation in front of both `inl` and `inr` indicates that the type arguments are implicit and Lean would attempt to infer it from the context. If it is not able to, we would have to write `@inl α β`, for instance. This would allow us to introduce another way of dealing with *inductive types*, this time we show how to construct a map whose domain is an *inductive type*.
For that we use the tactic `induction`, which splits the goal into similar goals for each of the inductive case.

```
lemma disjoint_union_is_sum :
    is_sum  (A ⊕ B)
            inl
            inr :=
    begin
        assume Q q₁ q₂,
        show ∃! s : (A ⊕ B) ⟶ Q, (q₁ = s ∘ inl ∧ q₂ = s ∘ inr), from
        let s : (A ⊕ B) ⟶ Q :=
            begin
                intro ab,  -- ab ∈ (A ⊕ B)
                induction ab,

                case inl :
                    begin
                        exact q₁ ab
                    end,
                case inr :
                    begin
                        exact q₂ ab
                    end in
            end,
        ...
    end
```

□

3.5.6. Pushout

Definition 7 (Pushout)**.** *Let $f : A \to B_1$ and $g : A \to B_2$ be a source. An object P together with morphisms $p_1 : B_1 \to P$ and $p_2 : B_2 \to P$ is called pushout of f and g, provided that:*

- *$p_1 \circ f = p_2 \circ g$, and*
- *to each other object Q, with morphisms $q_1 : B_1 \to Q$ and $q_2 : B_2 \to Q$, satisfying $q_1 \circ f = q_2 \circ g$, there exists a unique morphism $h : P \to Q$ with $h \circ p_1 = q_1$ and $h \circ p_2 = q_2$.*

$$\begin{tikzcd}
 & B_1 \arrow[dr, "p_1"] \arrow[drr, dashed, bend left, "q_1"] & & \\
A \arrow[ur, "f"] \arrow[dr, "g"'] & & P \arrow[r, dotted, "\exists! h"] & Q \\
 & B_2 \arrow[ur, "p_2"'] \arrow[urr, dashed, bend right, "q_2"'] & &
\end{tikzcd}$$

```
def is_pushout  {X : Type v} [category X]
    {A B₁ B₂: X} (f : A ⟶ B₁) (g : A ⟶ B₂)
    (P : X) (p₁ : B₁ ⟶ P) (p₂ : B₂ ⟶ P)
    : Prop
    :=  p₁ ⊚ f = p₂ ⊚ g ∧
        Π (Q : X) (q₁ : B₁ ⟶ Q) (q₂ : B₂ ⟶ Q),
          q₁ ⊚ f = q₂ ⊚ g → (∃! h : P ⟶ Q,
                                    q₁ = h ⊚ p₁ ∧ q₂ = h ⊚ p₂)
```

Pushouts are the dual structures of pullbacks, and thus they can be constructed in an opposite way to pullbacks, by taking the sum of the objects $\Sigma_{i \in I} B_i$ and then the pushout object would be the coequalizer object of the morphisms $(e_i \circ f_i)_{i \in I}$ along with the morphims $(c \circ e_i)_{i \in I}$.

$$\begin{tikzcd}
 & B_1 \arrow[dr, "e_1"] \arrow[drr, bend left, "c \circ e_1"] & & \\
A \arrow[ur, "f"] \arrow[dr, "g"'] & & B_1 \uplus B_2 \arrow[r, "c"] & P \\
 & B_2 \arrow[ur, "e_2"'] \arrow[urr, bend right, "c \circ e_2"'] & &
\end{tikzcd}$$

We can prove the previous statement very easily in Lean by defining the following lemma:

Lemma 3.5.7. *Given a source $(f_i : A \to B_i)_{i \in I}$, if both the sum $\Sigma_{i \in I} B_i$ and the coequalizer of the morphisms $(e_i \circ f_i : \Sigma_{i \in I} B_i \to C)_{i \in I}$ exist, then the coequalizer object C along with the morphisms $(c \circ e_i)_{i \in I}$ is the pushout of f_i.*

Proof. The proof does not introduce any new strategies in Lean. It is entirely based on the proofs of sum and coequalizer:

```
lemma coequalizer_sum_is_pushout
    {X : Type v} [category X]
    {A B₁ B₂: X} (f : A ⟶ B₁) (g : A ⟶ B₂)
    (S P : X) (s₁ : B₁ ⟶ S) (s₂ : B₂ ⟶ S)
    (p : S ⟶ P)
    (is_sm : is_sum S s₁ s₂)
    (is_coeq: is_coequalizer (s₁ ⊚ f) (s₂ ⊚ g) p):
    is_pushout f g P (p ⊚ s₁) (p ⊚ s₂)
    :=
    begin
        ...
    end
```

□

With that we have defined the basics of category theory needed for the understanding of coalgebra. There are obviously many more theories and lemmas, that one can prove in this context, but they go beyond the scope of this work.

4. Introduction to Coalgebra and its Lean-Formalization

4.1. State Based Systems

State based systems are systems, whose behavior depends on their internal state, which typically can not be observed. Universal coalgebra is the theory of state based systems [4]. Therefore, before defining coalgebra, we will introduce two examples of state based systems, deterministic and one nondeterministic. We will not go into details of these systems. We mainly want to show how they can be represented as coalgebras.

4.1.1. Automata

Definition 8. *An automaton can be defined over a set of symbols* Σ *with a set of output (data)* Γ *by a 4-tuple* (S, δ, s_0, γ)*, where:*

S	*is a set of states,*
$\delta : S \times \Sigma \to S$	*is the transition function,*
$s_0 \in S$	*is the initial state and*
$\gamma : S \to \Gamma$	*output function*

This is easily be modeled in Lean as a structure, except that δ is in the curried form:

```
universes u₃ u₂ u₁

structure Automaton  (Sigma : Type u₂)  (Γ : Type u₃):
    Type (max (u₁+1) u₂ u₃) :=
        (State : Type u₁)
        (δ : State → Sigma → State)
        (s₀ : State)
        (γ : State → Γ)
```

Notice that the universe level of this structure must be at least as high as the universes of `Sigma` and `Γ` and one level higher than `State`, so it would contain `State` as a field, hence `Type (max (u₁+1) u₂ u₃)`.

We define a coercion from an Automaton to its state type, which allows for implicit conversions between an `Automaton Sigma Γ` and the type `State`:

```
instance Automaton_to_Type : has_coe_to_sort (Automaton Sigma Γ) :=
      ⟨Type u₁ , λ A , A.State⟩
```

Definition 9 (Homomorphism). *Given two automata $A = (S_A, \delta_A, a_0, \gamma_A)$ and $B = (S_B, \delta_B, b_0, \gamma_B)$, defined over Σ with output set Γ, a map $\tau : S_A \to S_B$ is called a homomorphism if:*

$$\forall a \in S_A \,.\, \gamma_A(a) = \gamma_B(\tau(a)) \;\wedge\; \forall e \in \Sigma \,.\, \tau(\delta_A(a,e)) \;=\; \delta_B(\tau(a), e)$$

```
def is_homomorphism  {A B : Automaton Sigma Γ} (τ : A → B) : Prop :=
        ∀ a : A, A.γ a = B.γ (τ a) ∧
             ∀ e : Sigma , τ (A.δ a e) = B.δ (τ a) e
```

Acceptors

Acceptors are automata, whose output set is the set $2 = \{true, false\}$. They play an important rule in computer science, where they are used to recognize words (tokens) of a programming language.

Given the alphabet Σ, words over Σ are elements of the set Σ^*, which is defined inductively:

$\epsilon \in \Sigma^*$ and
if $e \in \Sigma$ and $v \in \Sigma^*$ then $e.v \in \Sigma^*$

We can model that in Lean inductively:

```
inductive word (Sigma : Type u₂)
        | ε  {}          : word
        | nonempty       : Sigma → word → word

notation e ‘ · ‘ v := word.nonempty e v
```

We can use this definition to recognize words using an acceptor:

```
def δ_Star {A : Automaton Sigma Γ} (s : A):
    word Sigma      →  A
    | ε            :=  s
    | (e·v)        :=  A.δ (δ_Star v) e

def accepted (A : Automaton Sigma Prop) (w : word Sigma) : Prop :=
    A.γ (δ_Star A.s₀ w)
```

Notice in the definition of δ_Star how the explicit parameter `(s : A)` is fixed in the recursive call. This is a design choice in Lean. Only the parameters in the type of the function (after the colon) can change in recursive calls.

4.1.2. Kripke Structures

Definition 10. *A Kripke-Structure can be defined over a set of properties Φ as triple (S, T, v), where:*

S	*is a set of states,*
$T \subseteq S \times S$	*is the transition relation and*
	we normally write $s_1 \twoheadrightarrow s_2$ to express $(s_1, s_2) \in T$
$v : S \to \mathscr{P}(\Phi)$	*is a validity function*
	$v(s)$ is the set of the properties $p \in \Phi$ which hold in state s

To model this in Lean, we can think of the set T as a map between S and $\mathscr{P}(S)$. While the curried form of this relation is $T : S \to S \to 2$, we found the modelling with the power set easier to work with.

```
universes u v w

structure Kripke (Φ : Type u): Type (max u (v+1)) :=
                (State : Type v)
                (T : State → set State)
                (v : State → set Φ)
```

We also define a coercion from a Kripke-structure to state type, similar to the one in automata (see 4.1.1):

Definition 11 (Homomorphism)**.** *Given two Kripke-structures $A = (S_A, T_A, v_A)$ and $B = (S_B, T_B, v_B)$, defined over Φ, a map $\sigma : S_A \to S_B$ is called a homomorphism if:*

$$\forall a_1, a_2 \in S_A \,.\; a_1 \twoheadrightarrow a_2 \Rightarrow \sigma(a_1) \twoheadrightarrow \sigma(a_2) \wedge$$

$$\forall a \in S_A,\, b \in S_B \,.\; (\sigma(a)) \twoheadrightarrow b \Rightarrow \exists a' \in S_A \,.\; a \twoheadrightarrow a' \wedge \sigma(a') = b \wedge$$

$$\forall a \in S_A \,.\; v_A(a) = v_B(\sigma(a))$$

```
def is_homomorphism {A B : Kripke Properties} (σ : A → B) : Prop :=
          (∀ a₁ a₂ :A , a₂ ∈ A.T a₁ → (σ a₂) ∈ B.T (σ a₁)) ∧
          (∀ (a : A) (b : B) , b ∈ B.T (σ a) →
                   ∃ a': A ,  a' ∈ A.T a ∧ σ a' = b) ∧
          (∀ a : A , A.v a = B.v (σ a))
```

4.2. Coalgebra

4.2.1. Basic Definitions and the Category of Coalgebras

Coalgebras

Definition 12. *A Coalgebra $\mathbb{A}$ of signature (a Set-endofunctor) F or an F-Coalgebra (in lean `Coalgebra F`) is a pair (A, α) consisting of a set A, called the carrier of $\mathbb{A}$ and a map $\alpha : A \to FA$, called the structure of $\mathbb{A}$.*

$$\mathrm{A} \xrightarrow{\alpha} \mathrm{F(A)}$$

```
structure Coalgebra (F : Type u ⥤ Type u)  :=
        (carrier : Type u) (α : carrier ⟶ F.obj carrier)
```

For the rest of this chapter, we define the following variables:

```
  variables {F : Type u ⥤ Type u}
          {𝔸 𝔹 ℂ: Coalgebra F}
```

Homomorphism

A map φ between two F-coalgebras $\mathbb{A}$ and $\mathbb{B}$ is called homomorphism from $\mathbb{A}$ to $\mathbb{B}$ if:

$$\alpha_{\mathbb{B}} \circ \varphi = F\varphi \circ \alpha_{\mathbb{A}}$$

i.e. the following diagram commutes:

$$\begin{tikzcd}
\mathrm{A} \arrow[r, "\varphi"] \arrow[d, "\alpha_{\mathbb{A}}"'] & \mathrm{B} \arrow[d, "\alpha_{\mathbb{B}}"] \\
\mathrm{F(A)} \arrow[r, "\mathrm{F}\,\varphi"'] & \mathrm{F(B)}
\end{tikzcd}$$

```
def is_coalgebra_homomorphism (φ : 𝔸 →  𝔹) : Prop :=
      𝔹.α ∘ φ =  (F.map φ) ∘ 𝔸.α
```

Based on this definition, we define the set of all coalgebra-homomorphism with domain $\mathbb{A}$ and codomain $\mathbb{B}$, which would later help us define the category of coalgebras Set_F.

```
def homomorphism (𝔸 𝔹 : Coalgebra F) : set (𝔸 → 𝔹):=
      λ φ, is_coalgebra_homomorphism φ
```

Remark. *The coercion between* `set X` *for some type* `X` *and* `subtype p` *, where* `p` *is the characteristic function of the set, is predefined in Lean. This allows us to define an element* `x` *of* `(S: set X)` *as a pair* `⟨x : X, h : x ∈ S⟩`*. It also allows us to use sets as types.*

Similar to automata and Kripke-structures, we can define a coercion from an $F-$coalgebra to its carrier and a coercion from the set `homomorphism 𝔸 𝔹` to the set of all maps `𝔸 → 𝔹`:

```
instance Coalgebra_to_sort : has_coe_to_sort (Coalgebra F) :=
      ⟨Type u,  λ 𝔸, 𝔸.carrier⟩

instance homomorphism_to_map (𝔸 𝔹 : Coalgebra F) :
                     has_coe_to_fun (homomorphism 𝔸 𝔹) :=
      { F := λ _, 𝔸 → 𝔹, coe := λ m, m.val}
```

Category Set_F

Now that we have to type `(Coalgebra F)` and the type `(homomorphism 𝔸 𝔹)`, all we need to do is prove that the category Set identity morphism is a homomorphism and the composition of two homomorphisms is a homomorphism, in order to define the category of coalgebras (called Set_F): Both proofs are quite simple and can be done using only the tactic `calc`. However, in the proof of composition, we notice, how definitions in Lean are hidden, unless we explicitly ask Lean to unfold them. This can be done in many different ways and here we see one of them:

```
lemma id_is_hom (𝔸 : Coalgebra F) : is_coalgebra_homomorphism (@id 𝔸) :=
   ...
```

```
lemma comp_is_hom (φ : homomorphism 𝔸 𝔹) (ψ : homomorphism 𝔹 ℂ)
                  : is_coalgebra_homomorphism (ψ ∘ φ) :=
   have ab : 𝔹.α ∘ φ =  F.map φ ∘ 𝔸.α := φ.property,
   have bc : ℂ.α ∘ ψ =  F.map ψ ∘ 𝔹.α := ψ.property,
   calc
           (ℂ.α ∘ ψ) ∘ φ = (F.map ψ) ∘ 𝔹.α  ∘ φ     : by rw bc
           ... = (F.map ψ) ∘ (F.map φ) ∘ 𝔸.α       : by rw ab
           ...
```

Now we are ready to define the category of coalgebras:

```
instance coalgebra_category : category (Coalgebra F) :=
{
   hom  := λ 𝔸 𝔹, homomorphism 𝔸 𝔹,
   id   := λ 𝔸 , ⟨@id 𝔸, id_is_hom 𝔸⟩,
   comp := λ 𝔸 𝔹 ℂ φ ψ, ⟨ψ ∘ φ, comp_is_hom 𝔸 𝔹 ℂ φ ψ⟩
}
```

Just like in the category *Set*, Lean was able to prove the conditions `id_comp'`, `comp_id'` and `assoc'` on its own (see 3.1).

4.2.2. Automata as Coalgebras

In this section, we will explore how an automaton can be seen as a coalgebra and how the homomorphisms between these coalgebras match the definition of automata homomorphisms.

Given the automaton $A = (S, \delta, \gamma, s_0)$ over the alphabet Σ with output set Γ, we define the functions:

$\delta' : S \to S^\Sigma$	the curried form of δ
$\alpha : S \to \Gamma \times S^\Sigma$	$s \mapsto \big(\gamma(s), \delta'(s)\big)$
	which combines of δ' and γ

We can consider α a coalgebra structure by defining the signature (functor) F, such that:

$$F(S) = \Gamma \times S^\Sigma$$

$$\varphi : S_A \to S_B \mapsto F\varphi : \Gamma \times S_A^\Sigma \to \Gamma \times S_B^\Sigma$$

$$F\varphi((o_1, f)) = (o_1, \varphi \circ f)$$

```
universes u₁ u₂ u₃

variables {Sigma : Type u₂} {Γ : Type u₃}

def F : Type u₁ ⥤ Type (max u₁ u₂ u₃) :=
  {
      obj := λ S, Γ × (Sigma → S),
      map := λ {A B} φ, λ ⟨d₁ , δ⟩ , ⟨d₁ , λ e, φ (δ e)⟩,
  }
```

Notice that Lean can prove the functor conditions `map_id'` and `map_comp'` by itself in this case (see 3.1.2).

With that we can define the coalgebra (S, α):

```
def α (A : Automaton Sigma Γ): A → F.obj A :=
      λ s, ⟨A.γ s,  A.δ s⟩
```

```
def Automata_Coalgebra (A : Automaton Sigma Γ): Coalgebra F :=
    ⟨A.State , α A⟩
```

We can also show the equivalence between the notation of automata homomorphism and the corresponding coalgebra homomorphism:

```
lemma Automata_Coalgebra_hom {A B : Automaton Sigma Γ} (φ : A → B):
    is_homomorphism φ ↔
    @is_coalgebra_homomorphism
        F (Automata_Coalgebra A) (Automata_Coalgebra B) φ :=
  begin
     split,
     intro h,
     dsimp at *,
     ext1 s,
     dsimp at *,
     have hs := h s,
     ext1,
    ...
  end
```

- The first `dsimp at *` unfolds the goal into its definition, while the second one simplifies the goal from the form $(f \circ g)(x) = h(x)$ to $f(g(x)) = h(x)$
- We have already seen `ext1` in 3.5.2. However, here the first one simplifies a goal of functional equality $f = g$ into $f(s) = g(s)$ for any s from the common domain, while the second one, as we have seen before, breaks the goal of equality between two structures into the equality between each of the structure parameters.

4.2.3. Kripke Structures as Coalgebras

In this section, we define a coalgebra based on a Kripke structure the same way we did with automata:

Given the Kripke-structure $K = (S, T, v)$ over the set of properties Φ, we define the functions:

$t : S \to \mathscr{P}(S) \qquad s \mapsto U$, where $s' \in U \Leftrightarrow (s, s') \in T$

$\alpha : S \to \mathscr{P}(S) \times \mathscr{P}\Phi \qquad s \mapsto (t(s), v(s))$

We can consider α a coalgebra structure by defining the signature (functor) F, which maps a set S to $\mathscr{P}(S) \times \mathscr{P}(\Phi)$ and a map $\sigma : S_A \to S_B$, to a map that maps $(U, P) : \mathscr{P}(S) \times \mathscr{P}(\Phi)$ to a pair:

$$(\sigma[U], P)$$

```
variable {φ : Type v}
def F : Type u ⥤ Type (max v u) :=
{
```

```
    obj := λ S, (set S) × (set φ),
    map := λ {A B} φ, λ ⟨U , P⟩ , ⟨image φ U, P⟩,
    map_id' :=  ...,
    map_comp' :=
        begin
            ...
            calc
            F._match_1 (g ∘ f) (S, P)
                = ⟨image (g ∘ f) S, P⟩          : rfl
            ... = ⟨image g (image f S), P⟩      : by rw [img_comp f g S]
            ... = F._match_1 g (F._match_1 f (S, P)) : rfl
        end
}
```

In this case Lean was not able to prove the functor-conditions on its own. However, this proof even raises an issue with the tactic `tidy`, where it, instead of failing, generates a proof, that is rejected by Lean. We will discuss this issue in the next chapter.

Because of that, the proof was done without the use of `tidy`. However, the other strange issue, that appears in the proof is that `_match_1` is not part of the definition of functor. In fact it does not come up anywhere in the category theory library, yet it shows up as part of the goal, and we have to use it in the proof to denote `F.map`. So far we have not found an explanation for this.

Now we can define the coalgebra that represents Kripke structures (S, α):

```
def α (K : Kripke φ): K.State → F.obj K.State :=
        λ S, ⟨K.T S,  K.v S⟩

def Kripke_Coalgebra (K : Kripke φ): Coalgebra F :=
    ⟨K.State , α K⟩
```

Just like we did with automata, we show the the equivalence between the Kripke structure homomorphism and the corresponding coalgebra homomorphism. However, the proof in Lean this time is much longer than the one in automata.

```
lemma Kripke_Coalgebra_hom {K1 K2 : Kripke φ} (φ : K1 → K2):
    is_homomorphism φ φ ↔
    @is_coalgebra_homomorphism
         F (Kripke_Coalgebra K1) (Kripke_Coalgebra K2) φ :=
    ...
```

Due to the difficult way, homomorphisms in Kripke structures are defined, this proof is long and hard to read. It does not serve as a good example to explain new concepts in Lean. However, all the functions and tactics, used in this proof, appear in other proofs.

4.3. Diagram lemma

Theorem 4.3.1. *If a homomophism is bijective, then its inverse is also a homomorphism.*

Proof. The proof can be done in a simple calculation using the definition of *bijective* in Lean, which states that a bijective function has a left and right inverse:

```
theorem bij_inverse_of_hom_is_hom
            (φ : homomorphism 𝔸 𝔹)
            (bij : bijective φ) :
   -- Lean allows us to add definitions inside the type.
   let inv : 𝔹 → 𝔸 := some (bijective_iff_has_inverse.1 bij) in
      is_coalgebra_homomorphism inv := ...
```

□

Before proving the diagram lemma for coalgebras, we need to prove two helpful lemmas, defined using the following diagram:

$$
\begin{tikzcd}
A \arrow[r, "f"] \arrow[rr, bend left, "g \circ f"] \arrow[d, "\alpha_{\mathbb{A}}"'] & B \arrow[r, "g"] \arrow[d, "\alpha_{\mathbb{B}}"] & C \arrow[d, "\alpha_{\mathbb{C}}"] \\
F(A) \arrow[r, "F\,f"'] \arrow[rr, bend right, "F(g \circ f)"'] & F(B) \arrow[r, "F\,g"'] & F(C)
\end{tikzcd}
$$

Lemma 4.3.2. *If $g \circ f$ is a homomorphism and f is a surjective (or epi) homomorphism, then g is a homomorphism:*

Proof. The proof can be done in a simple calculation then right canceling the epi function f:

```
lemma surj_to_hom  (f: 𝔸.carrier ⟶ 𝔹.carrier) (g: 𝔹.carrier ⟶ ℂ.
    carrier)
            (hom_gf : is_coalgebra_homomorphism (g ∘ f))
            (hom_f : is_coalgebra_homomorphism f)
            (ep : epi f) : is_coalgebra_homomorphism g :=
        have  h1 : (ℂ.α ∘ g) ⊚ f = (F.map g ∘ 𝔹.α) ⊚ f := ...,
        right_cancel f h1
```

□

Lemma 4.3.3. *If $g \circ f$ is a homomorphism and g is a injective (or mono) homomorphism, then f is a homomorphism:*

Proof. To prove this lemma, we need to consider two cases. First case is if the carrier of $\mathbb{B}$ is not empty, here we can use the lemma in 3.2 to cancel $F(g)$. Second case is if the carrier of $\mathbb{B}$ is empty, then the carrier of $\mathbb{A}$ must be empty and f is obviously a homomorphism. We prove that in the lemma `empty_hom_codom` (see appendix for proof A.2.3).
We split the proof in Lean using the tactic `cases` together with the function `classical.em`, where `classical` stands for classical logic and `em` for the law of the excluded middle. This tactic takes a proposition and splits the proof into the proposition and its negation.

```
lemma inj_to_hom (f : 𝔸.carrier ⟶ 𝔹.carrier)
                 (g : 𝔹.carrier ⟶ ℂ.carrier)
                 (hom_gf : is_coalgebra_homomorphism (g ∘ f))
                 (hom_g : is_coalgebra_homomorphism g)
                 (inj : injective g) : is_coalgebra_homomorphism f :=
begin
     cases classical.em (nonempty 𝔹) with n_em_𝔹 emp_𝔹,
     have  h1 : (F.map g) ⊚ (F.map f) ⊚ 𝔸.α = F.map g ⊚ (𝔹.α ⊚ f) :=
    ...,

     haveI inh_𝔹 : inhabited 𝔹 := ⟨choice n_em_𝔹⟩,
      -- haveI is used to define an instance.
     haveI fg_mono : mono (F.map g) := mono_preserving_functor g inj,

     exact  left_cancel (F.map g) (eq.symm h1),

     exact empty_hom_codom f emp_𝔹
end
```

□

Lemma 4.3.4 (Coalgebraic Diagram Lemma). *Let $\varphi : \mathbb{A} \twoheadrightarrow \mathbb{B}$, $\psi : \mathbb{A} \to \mathbb{C}$ be homomorphisms, and φ is surjective. Then there exists a unique homomorphism $\chi : \mathbb{B} \to \mathbb{C}$ such that $\chi \circ \varphi = \psi$, iff $kern\, \varphi \subseteq kern\, \psi$.*

$$\begin{tikzcd} \mathbb{A} \arrow[r, two heads, "\varphi"] \arrow[dr, "\psi"'] & \mathbb{B} \arrow[d, dotted, "\exists! \chi"] \\ & \mathbb{C} \end{tikzcd}$$

Proof. The proof does not introduce any new ideas in Lean.

```
lemma coalgebra_diagram (φ : homomorphism 𝔸 𝔹) (ψ : homomorphism 𝔸 ℂ)
                              (sur : surjective φ)
   : (∃! χ : homomorphism 𝔹 ℂ , χ ∘ φ = ψ) ↔ (sub_kern φ ψ)
   :=  ...
```

□

4.4. Subcoalgebra

Definition 13 (Open subset)**.** *A subset $S \subseteq \mathbb{A}$ is called open, if there exists some (structure) map $\alpha : S \to F(S)$, so that the inclusion from S to $\mathbb{A}$, $S \hookrightarrow \mathbb{A}$ is a homomorphismus.*

The pair (S, α) is then called a subcoalgebra of $\mathbb{A}$. We can formalize open subsets and subcoalgebras as follows:

```
def openset {𝔸 : Coalgebra F} (S : set 𝔸) : Prop :=
     ∃ α : S → F.obj S ,  @is_coalgebra_homomorphism
                                 F ⟨S , α⟩ 𝔸 (S ↪ 𝔸)

structure SubCoalgebra (S : set 𝔸)  :=
   (α : S → F.obj S)
   (h: @is_coalgebra_homomorphism F ⟨S , α⟩ 𝔸 (S ↪ 𝔸))
```

Lemma 4.4.1. *Each open subset S of $\mathbb{A}$ carries a unique structure map $\alpha : S \to F(S)$, which makes (S, α) a subcoalgebra of $\mathbb{A}$.*

Proof. The proof will be divided into two parts based on whether S is empty or not:

```
lemma subcoalgebra_unique_structure (S : set 𝔸) (h : openset S)
         : let α := some h in
   ∀ σ : S → F.obj S,
      @is_coalgebra_homomorphism F ⟨S , σ⟩ 𝔸 (S ↪ 𝔸) → σ = α  :=
begin
   ...
   have h2 : ∀ (f₁ f₂ : S → F.obj S), f₁ = f₂ :=
           map_from_empty S (F.obj S) (nonempty_notexists emp),
   -- map_from_empty asserts that a map from an empty set is always
    unique.
   -- See  A.2.4 for definition
   ...
end
```

□

4.5. Homomorphic Image

Before introducing the concept of a homomorphic image, we need to prove the following lemma:

Lemma 4.5.1. *Given a surjective homomorphism $\varphi : \mathbb{A} \twoheadrightarrow \mathbb{B}$, the coalgebra structure $\alpha_{\mathbb{B}}$ can be determined using φ.*

Proof. The proof is relatively simple. However, it shows an important point, when dealing with tactics for the axiom of choice. Given a term `ex : ∃ x : X , p x`, we can use the tactic `cases ex with x p_x`, to apply the axiom of choice on `ex` to get `x : X` and a proof `p_x : p x`.
Another way of doing that is to use `some` and `some_spec`:

```
let x : X := some (ex) in
have p_x : p x := some_spec (ex),
```

These two forms are, however, not equivalent. In the latter form, `x` satisfies any property that `some ex` satisfies, while in the `cases` tactic, `x` would only satisfy `p x` and otherwise Lean "forgets" how we got `x`. This limitation was significant in this proof:

```
lemma surj_hom_to_coStructure (φ : homomorphism 𝔸 𝔹)
                        (sur : surjective φ):
        -- surjective is defined as ∀ b, ∃ a, φ a = b
    let χ : 𝔹 → F.obj 𝔹 := λ b,   -- χ is defined using some (sur b)
        ((F.map φ) ∘ 𝔸.α) (some (sur b)) in
            𝔹.α = χ :=
    begin
        ...
            let a := some (sur b),
            have a_b : φ a = b := some_spec (sur b),
            have χ_b : χ b = ((F.map φ) ∘ 𝔸.α) a := rfl,
            -- cases (sur b) with a a_b, would not work here.
            ...
        end,
        exact funext elements
    end
```

□

With this proof, we can make the following definition:

Definition 14 (Homomorphic image)**.** *If a surjective homomorphism $\varphi : \mathbb{A} \twoheadrightarrow \mathbb{B}$ exists, then $\mathbb{B}$ is called the homomorphic image of $\mathbb{A}$.*

```
def homomorphic_image (𝔸 𝔹: Coalgebra F) : Prop :=
    ∃ φ : homomorphism 𝔸 𝔹, surjective φ
```

To make use of this defintion, we need to prove one more theorem, **factorization**:

Theorem 4.5.2. *Given a homomorphism $\varphi : \mathbb{A} \twoheadrightarrow \mathbb{B}$ and two maps, a surjective map $f : A \twoheadrightarrow Q$ and an injective map $g : Q \rightarrow B$, where A and B are carriers of $\mathbb{A}$ and $\mathbb{B}$ respectively and Q is an arbitrary set, such that $\varphi = g \circ f$, there exists a unique coalgebra structure $\alpha_{\mathbb{Q}} : Q \rightarrow F(Q)$ that makes f a homomorphism. g is also a homomorphism in respect to the same structure.*

Proof. The proof can be easily constructed using E-M-Squares (see 3.4). However, it had to be done in two steps. First we prove the existence and uniqueness of the structure $\alpha_{\mathbb{Q}}$. Then, in a separate theorem, we prove that g is also a homomorphism in respect to that structure.

$$\begin{tikzcd}
\mathrm{A} \arrow[r, "f"] \arrow[rr, bend left, "\varphi = g \circ f"] \arrow[d, "\alpha_{\mathbb{A}}"'] & \mathrm{Q} \arrow[r, "g"] \arrow[d, dotted, "\exists! \alpha_{\mathbb{Q}}"] & \mathrm{B} \arrow[d, "\alpha_{\mathbb{B}}"] \\
\mathrm{F(A)} \arrow[r, "\mathrm{F} f"'] \arrow[rr, bend right, "F\varphi"'] & \mathrm{F(Q)} \arrow[r, "\mathrm{F}\, g"'] & \mathrm{F(B)}
\end{tikzcd}
\quad \text{or} \quad
\begin{tikzcd}
\mathrm{A} \arrow[r, "f"] \arrow[d, "Ff \circ \alpha_{\mathbb{A}}"'] & \mathrm{Q} \arrow[d, "\alpha_{\mathbb{B}} \circ g"] \arrow[dl, dotted, "\exists! \alpha_{\mathbb{Q}}"] \\
\mathrm{F(Q)} \arrow[r, "Fg"'] & \mathrm{F(B)}
\end{tikzcd}$$

The proofs are fairly long, yet with no new ideas.

```
theorem factorization {Q : Type u}
    (φ: homomorphism 𝔸 𝔹) (f: 𝔸.carrier ⟶ Q) (g: Q ⟶ 𝔹.carrier)
    (h: φ.val = g ∘ f) (sur: surjective f) (inj: injective g) :
        ∃! α_Q : Q ⟶ F.obj Q ,
            @is_coalgebra_homomorphism F 𝔸 ⟨Q , α_Q⟩ f := ...
```

```
theorem factorization_hom {Q : Type u}
    (φ: homomorphism 𝔸 𝔹) (f: 𝔸.carrier ⟶ Q) (g: Q ⟶ 𝔹.carrier)
        (h: φ.val = g ∘ f) (sur: surjective f) (inj: injective g) :
    let α_Q := some (factorization φ f g h sur inj) in
        @is_coalgebra_homomorphism F ⟨Q , α_Q⟩ 𝔹 g := ...
```

□

One way to factorize any function $f : A \rightarrow B$ is by using its range as the middle set. The range of a function is defined as (the second definition is the one used in Lean):

$$f[A] := \{f(a) |\ a \in A\} \quad = \quad \{b |\ \exists a \in A.\ f(a) = b\}$$

This way we can factorize f to f', the image restriction of f, and $\subseteq^B_{f[A]}$, the inclusion of $f[A]$ into B:

$$f' : A \rightarrow f[A], \quad \forall a \in A.\ \ f'(a) = f(a)$$
$$\subseteq^B_{f[A]} : f[A] \rightarrow B, \quad \forall b \in B.\ \ \subseteq^B_{f[A]} (b) = b$$

Obviously f' is surjective and $\subseteq^B_{f[A]}$ is injective.
Another notation for the inclusion $\subseteq^B_A$ is $A \hookrightarrow B$, which is the notation, we use in Lean.

Corollary 4.5.2.1. *If $\varphi : \mathbb{A} \to \mathbb{B}$ is a homomorphism, then the range of φ, is a homomorphic image of $\mathbb{A}$ and a subcoalgebra of $\mathbb{B}$.*

Proof. We divide the proof into three parts. Firstly, we need to prove the existence of a coalgebra structure, which makes the image restriction (in Lean it is called `range_factorization`) of f and the inclusion of the range of f homomorphisms:

```
lemma structure_existence (φ : homomorphism 𝔸 𝔹) [inhabited 𝔸]
    : ∃ α : (range φ) → F.obj (range φ),
        let ℝ : Coalgebra F :=⟨range φ , α⟩ in
        @is_coalgebra_homomorphism F 𝔸 ℝ (range_factorization φ) ∧
        @is_coalgebra_homomorphism F ℝ 𝔹 ((range φ) ↪ 𝔹) :=  ...
```

At this point, it is trivial to show, that $\varphi[A]$ is a homomorphic image of $\mathbb{A}$ and subcoalgebra of $\mathbb{B}$, (see the code in A.2.5). □

4.6. Colimits and Limits

We will start with colimits of the category Set_F, because they match their counterparts in the category Set, which is not the case, as we will discuss later, with limits.

In this section, we need to move back and forth between two categories, which means two instances of one type class, `category`. Lean is not able to interpret a morphism in category Set_F as a morphism in category Set. To allow this interpretation, we need to define this rather obvious lemma:

```
lemma eq_in_set {A : Type u} {S: set A} {a b: S}
: a.val = b.val ↔ a = b :=
    by {cases a, cases b, dsimp at *, simp at *}
```

The tactic `cases a` here deconstruct `a` into `⟨a_val, a_property⟩`. The tactic `dsimp at *` simplifies `⟨a_val, a_property⟩.val` to `a_val`.

Remark. *The tactic `dsimp` is similar to `simp`, except it only uses definitional equalities.*

4.6.1. Coequalizer

We want to show, that the coequalizer in the category Set is also the coequalizer in the category Set_F. We achieve that in two steps:

Lemma 4.6.1 (Proof of existence)**.** *Given two parallel homomorphisms $\varphi, \psi : \mathbb{A} \to \mathbb{B}$, there exists a unique coalgebra structure $\alpha : B \backslash \Theta \to F(\mathbb{B} \backslash \Theta)$, which makes the Set category coequalizer $\pi_\Theta : \mathbb{B} \to \mathbb{B} \backslash \Theta$ a homomorphism*

Proof. This proof is very simple, since everything was already shown in the category Set coequalizer. However, in order to use the `quot_is_coequalizer` lemma (see 3.5.4), we need to define the category Set morphisms φ_1 and ψ_1 explicitly, as shown:

```
theorem coequalizer_is_homomorphism :
    let 𝔹_Θ := theta φ ψ in
    let π_Θ : 𝔹.carrier ⟶ 𝔹_Θ := coequalizer φ ψ in
    ∃! α : 𝔹_Θ → (F.obj 𝔹_Θ),
    @is_coalgebra_homomorphism F 𝔹 ⟨𝔹_Θ , α⟩ π_Θ
    :=
   ...
    let φ₁  : 𝔸.carrier ⟶ 𝔹.carrier := φ.val,
    let ψ₁ : 𝔸.carrier ⟶ 𝔹.carrier := ψ.val,
   have h : is_coequalizer φ₁ ψ₁ π_Θ := quot_is_coequalizer φ₁ ψ₁,
   ...
```

□

Lemma 4.6.2 (Set_F coequalizer)**.** *Given two parallel homomorphisms $\varphi, \psi : \mathbb{A} \to \mathbb{B}$, the coalgebra $\mathbb{C} := (\mathbb{B}\backslash\Theta, \alpha_{\mathbb{C}})$, where $\alpha_{\mathbb{C}}$ is the structure shown in last theorem, along with the homomorphism $\pi_\Theta : \mathbb{B} \to \mathbb{C}$ is the coequalizer of φ and ψ in category Set_F.*

Proof. In this proof, we need to use the lemma `eq_in_set` every time we change from one category to the other.

```
theorem set_coequalizer_is_coalgebra_coequalizer :
    let 𝔹_Θ := theta φ ψ in
    let π_Θ : 𝔹.carrier ⟶ 𝔹_Θ := coequalizer φ ψ in
    let α : 𝔹_Θ → F.obj (𝔹_Θ):=
                  some (coequalizer_is_homomorphism φ ψ) in
    let h_π := (some_spec (coequalizer_is_homomorphism φ ψ)).1 in
    let co_𝔹_Θ : Coalgebra F := ⟨𝔹_Θ, α⟩ in
    let π₁ : 𝔹 ⟶ co_𝔹_Θ := ⟨π_Θ, h_π⟩ in

    is_coequalizer φ ψ π₁ :=
    ...
        let χ : co_𝔹_Θ ⟶ ℚ := ...
        have spec : χ ∘ π_Θ = q := ...,
        have h1 : (χ ⊚ π₁) = q := eq_in_set.1 spec,
        ...,
        have coeq3: χ₁.val = χ.val := ...,
        exact eq_in_set.1 coeq3,
    end
```

□

4.6.2. Sum

We can define a sum of two F-coalgebras $\mathbb{A} = (A, \alpha_{\mathbb{A}})$ and $\mathbb{B} = (B, \alpha_{\mathbb{B}})$ using the disjoint union as a carrier. The coalgebra structure is defined using the canonical injections $e_A : A \to A \uplus B$ and $e_B : B \to A \uplus B$:

$$\alpha_{\mathbb{A}\uplus\mathbb{B}} : A \uplus B \to F(A \uplus B)$$

$$\alpha_{\mathbb{A}\uplus\mathbb{B}}(1, a) := (F(e_A) \circ \alpha_{\mathbb{A}})(a)$$

$$\alpha_{\mathbb{A}\uplus\mathbb{B}}(2, b) := (F(e_B) \circ \alpha_{\mathbb{B}})(b)$$

In Lean syntax, this can be modeled using pattern matching:

```
def α_sum : (𝔸 ⊕ 𝔹) → F.obj (𝔸 ⊕ 𝔹)
          | (inl a) := ((F.map inl) ∘ 𝔸.α) a
          | (inr b) := ((F.map inr) ∘ 𝔹.α) b

def sum_of_coalgebras : Coalgebra F  := ⟨(𝔸 ⊕ 𝔹) ,  (α_sum 𝔸 𝔹)⟩
```

With respect to this coalgebra, from here on represented as $(\mathbb{A}\boxplus\mathbb{B})$, it is easy to show that the canonical injections are homomorphisms, in fact Lean can prove it by itself using the tactic `tidy` (see A.2.7).

Now we can show that this is in fact the sum in category Set_F:

Lemma 4.6.3. *Given two F-coalgebras $\mathbb{A}$ and $\mathbb{B}$, the coalgebra $(\mathbb{A}\boxplus\mathbb{B})$ is the sum of $\mathbb{A}$ and $\mathbb{B}$ in the category $Set - F$.*

Proof. The proof is about a hundred lines of code in Lean (see A.2.7). In part because we need to move back and forth between the categories and in part because the tactic `induction` forces the user to enter tactic mode in each case.

```
theorem set_sum_is_coalgebra_sum :
    let e₁ : 𝔸 → (𝔸 ⊕ 𝔹) := inl in
    let e₂ : 𝔹 → (𝔸 ⊕ 𝔹) := inr in
    let hom_e₁ : 𝔸 ⟶ (𝔸 ⊞ 𝔹) := ⟨e₁ ,  inl_is_homomorphism 𝔸 𝔹⟩ in
    let hom_e₂ : 𝔹 ⟶ (𝔸 ⊞ 𝔹) := ⟨e₂ ,  inr_is_homomorphism 𝔸 𝔹⟩ in
          is_sum (𝔸 ⊞ 𝔹) h_e₁ h_e₂ := ...
```

□

This sum of coalgebras allows us to prove the next lemma:

Lemma 4.6.4. *Given two subcoalgebras U and V of $\mathbb{A}$, both their union $U \cup V$ and their intersection $U \cap V$ are subcoalgebras of $\mathbb{A}$.*

Proof. We do the proof in two separate theorems. Both, however, are long proofs.

```
noncomputable theorem subcoalgebra_union_is_coalgebra
    {U₁ U₂ : set 𝔸}
    (S₁ : SubCoalgebra U₁)
```

```
    (S₂ : SubCoalgebra U₂)
    : SubCoalgebra (U₁ ∪ U₂) :=  ...
```

The first theorem considers $\mathbb{A}$ with the inclusions $i_1 : U_1 \hookrightarrow A$ and $i_2 : U_2 \hookrightarrow A$ a competitor to the sum $U_1 \boxplus U_2$ and therefore there exists a homomorphism $\varphi : (U_1 \boxplus U_2) \to \mathbb{A}$, that makes $e_1 = \varphi \circ i_1$ and $e_2 = \varphi \circ i_2$.

```
let S : Coalgebra F := ⟨U₁ , S₁.α⟩ ⊞ ⟨U₂ , S₂.α⟩,
have ex :=
         set_sum_is_coalgebra_sum ⟨U₁ , S₁.α⟩ ⟨U₂ , S₂.α⟩ 𝔸
             ⟨ (U₁↪ 𝔸), S₁.h⟩  ⟨(U₂ ↪ 𝔸) , S₂.h⟩,
let φ : S ⟶ 𝔸 := some ex,
```

The main part of the proof is to show that $\varphi[U_1 \boxplus U_2] = U_1 \cup U_2$. This part is relatively long, yet not so challenging and we tried to make it easy to read.
Then we can use the corollary 4.5.2.1 to show that $U_1 \cup U_2$ is a subcoalgebra of $\mathbb{A}$.

```
have all : ∀ a : 𝔸 , a ∈ (range φ) ↔ a ∈ (U₁ ∪ U₂) := ...
rw ←(eq_sets.1 all),
exact range_is_subCoalgebra φ,
```

The second part of the proof is to show that the intersection of two subcoalgebras $U \cap V$ is a subcoalgebra. If the intersection is empty, then the proof is obvious and we can show that in a lemma called `empty_openset` , which, like the name suggests, proves that any empty subset is an open subset (see A.2.7 for the proof).

If the intersection is not empty, we need to define two functions $p_w : U \to U \cap V$ and $q_w : \mathbb{A} \to V$, for some $w \in U \cap V$:

$$p_w(u) = \begin{cases} u & if\ u \in U \cap V \\ w & else \end{cases} \quad \text{and} \quad q_w(a) = \begin{cases} a & if\ a \in V \\ w & else \end{cases}$$

```
let p_w :U → U ∩ V := λ u , if uUV : u.val ∈ U ∩ V
                        then ⟨u.val , uUV⟩
                        else ⟨w.val , w.property⟩,
let q_w :𝔸 → V := λ a , if aV : a ∈ V
                     then ⟨a , aV⟩
                     else ⟨w.val , and.right w.property⟩,
```

However, Lean only accepts this definition, using `if p then t₁ else t₂`, if the proposition `p` is decidable. Therefore, we need to add the following instances of type class `decidable` to the theorem's parameters (assumptions).

```
[∀ x : 𝔸 , decidable (x ∈ U ∩ V)]
[∀ x : 𝔸 , decidable (x ∈ V)]
```

There are many ways to construct proofs over functions that use if-statements in Lean. In this proof, we present two tactical approaches. The first one is automated using the tactic `simp` equipped with a list of assumptions to prove each case, for example:

```
have pwu_u : p_w u = ⟨u.val , uI⟩:=
     by simp [p_w , rfl, uI],
```

This works only if all cases are simple to prove. The second allows for harder proofs:

```
have qwu_u: q_w u.val =  (⟨w.val , (w.property).2⟩ : V) :=
   begin
       simp[q_w],
       split_ifs,
       exact absurd (and.intro u.property h) uNI,
       exact rfl
   end,
```

The rest of the proof consists of previously explained tactics.

```
theorem subcoalgebra_intersection_is_coalgebra
    {U V : set 𝔸}
    (S₁ : SubCoalgebra U)
    (S₂ : SubCoalgebra V)
    [∀ x : 𝔸 , decidable (x ∈ U ∩ V)]
    [∀ x : 𝔸 , decidable (x ∈ V)]
    : openset (U ∩ V) :=
```

□

4.6.3. Pushout

Last chapter we showed that the pushout can be constructed from the sum and coequalizer, if they exists, in any category (see 3.5.6). All what is left for us to show here is the following lemma:

Lemma 4.6.5. *Given the homomorphisms $\varphi : \mathbb{A} \to \mathbb{B}_1$ and $\psi : \mathbb{A} \to \mathbb{B}_2$, the pushout set that results from taking the sum of $\mathbb{B}_1$ and $\mathbb{B}_2$ then the coequalizer of $(e_{\mathbb{B}_1} \circ \varphi)$ and $(e_{\mathbb{B}_2} \circ \psi)$, possesses a unique coalgebra structure that makes $(\pi_\Theta \circ e_{\mathbb{B}_1})$ and $(\pi_\Theta \circ e_{\mathbb{B}_2})$ homomorphisms.*

Proof. The proof depends mainly on the proofs of coequalizer and sum in category Set_F and does not introduce any new Lean concepts.

```
theorem pushout_is_coalgebra :
    let S := 𝔹₁ ⊞ 𝔹₂  in
    let 𝔹_Θ := @theta 𝔸 S (inl ∘ φ) (inr ∘ ψ) in
    let π_Θ := @coequalizer 𝔸 S (inl ∘ φ) (inr ∘ ψ) in
    ∃! α : 𝔹_Θ → F.obj 𝔹_Θ,
    let P : Coalgebra F := ⟨𝔹_Θ, α⟩ in
```

```
    @is_coalgebra_homomorphism F 𝔹₁ P (π_Θ ∘ inl) ∧
    @is_coalgebra_homomorphism F 𝔹₂ P (π_Θ ∘ inr)  := ...
```

□

4.6.4. Equalizer

Theorem 4.6.6. *Given homomorphisms $\varphi, \psi : \mathbb{A} \to \mathbb{B}$, their equalizer in the category Set_F exists and it is the largest coalgebra $[E]$ contained in the equalizer E from the category Set.*

Proof. To prove this, we first need to define the largest coalgebra inside a set:

```
def is_largest_coalgebra {S : set 𝔸} (P : set S): Prop :=
    (∃ α : P → F.obj P ,
        @is_coalgebra_homomorphism F ⟨P, α⟩ 𝔸
                        ((S ↪ 𝔸) ∘ (P ↪ S))) ∧
        ∀ P₁ : set S,
    (∃ α : P₁ → F.obj P₁ ,
    @is_coalgebra_homomorphism F ⟨P₁, α⟩ 𝔸
                    ((S ↪ 𝔸) ∘ (P₁ ↪ S))) → P₁ ⊆ P

noncomputable def largest_Coalgebra {S : set 𝔸} {P : set S}
                        (lar : is_largest_coalgebra P):
    Coalgebra F := ⟨P , some lar.1⟩
```

Notice that we define `P` as an element of `set S` and not `P ⊆ S`. There is an important difference between the two definitions. In the first, `P` is a an element of `set 𝔸` and it is a subset of `S`, while in the second `P` is an element of `set S` and Lean sees no connection between it and `𝔸`.

The reason for this choice comes in the following proof. Given a "competitor" coalgebra (Q, α_Q), we can obtain a unique map $f : Q \to E$, where E is the equalizer set from the category Set. We need to show that the image $f[Q]$ is a subset of the largest coalgebra $[E]$. However, in Lean `range f` is an element of `set E` and not connected to $\mathbb{A}$. This workaround makes the proof more complicated than it needs to be.

The inclusion map $\sigma_1 : [E] \hookrightarrow \mathbb{A}$ is defined as a composition of $[E]$ inclusion in E and then E inclusion of $\mathbb{A}$. This composition loses all the properties, like injectivity, already proven about inclusions. This required us to prove each property again, which makes the proof immensely longer.

```
theorem largest_subcoalgebra_equalizer
    {C : set (equalizer_set φ ψ)}
    (lar : is_largest_coalgebra C):
    let E := equalizer_set φ ψ in
    let e : E → 𝔸 := (E ↪ 𝔸) in
    let ℂ : Coalgebra F := ⟨C , some lar.1⟩  in
    let σ : ℂ ⟶ 𝔸 := ⟨(e ∘ (C ↪ E))  , some_spec lar.1⟩ in
```

```
    is_equalizer φ ψ σ := ...
```

□

4.7. Bisimulations

Definition 15 (Bisimulation)**.** *A binary relation $R \subseteq A \times B$, where A is the carrier of carrier of $\mathbb{A}$ and B is the carrier of $\mathbb{B}$, is called bisimulation if there exists a coalgebra structure $\alpha_{\mathbb{R}} : R \to F(R)$, which makes the projections $\pi_A : R \to A$ and $\pi_B : R \to B$ homomorphisms.*

$$\begin{tikzcd}
A \arrow[d, "\alpha_{\mathbb{A}}"'] & R \arrow[l, "\pi_A"'] \arrow[r, "\pi_B"] \arrow[d, "\alpha_{\mathbb{R}}"'] & B \arrow[d, "\alpha_{\mathbb{B}}"] \\
F(A) & F(R) \arrow[l, "F(\pi_A)"] \arrow[r, "F(\pi_B)"'] & F(B)
\end{tikzcd}$$

```
def is_bisimulation (R : set (𝔸 × 𝔹)) : Prop :=
    ∃ ρ : R → F.obj R,
    let ℝ : Coalgebra F := ⟨R , ρ⟩ in
    let π₁ : ℝ → 𝔸 := λ r, r.val.1 in
    let π₂ : ℝ → 𝔹 := λ r, r.val.2 in
    is_coalgebra_homomorphism π₁ ∧ is_coalgebra_homomorphism π₂
```

We could connect the concept of coalgebra homomorphisms with the notion of bisimulations through the following theorem:

Theorem 4.7.1. *A map $f : A \to B$ is a homomorphism between $\mathbb{A} = (A, \alpha_{\mathbb{A}})$ and $\mathbb{B} = (B, \alpha_{\mathbb{B}})$, iff its graph:*

$$\{(a, f(a)) \mid a \in A\}$$

is a bisimulations between $\mathbb{A}$ and $\mathbb{B}$.

Proof. The proofs at this point become harder to read, because we tend to use more and more tactics, which makes constructing the proofs quicker and simpler for the developer, yet harder for the reader.

```
theorem homomorphism_iff_bisimulation (f : 𝔸 → 𝔹):
    is_coalgebra_homomorphism f ↔ is_bisimulation (map_to_graph f)
    := ...
```

We first show the equality $f = \pi_2 \circ \pi_1^{-1}$ (it is easy to show, that the projections are bijective). This easily shows that f is a homomorphism.

```
have elements : ∀ a , (π₂ ∘ inv) a = f a :=  ...
```

We then use $\rho := F\pi_1^{-1} \circ \alpha_{\mathbb{A}} \circ \pi_1$ as a coalgebra structure.

```
let ρ := (F.map inv) ∘ 𝔸.α ∘ π₁,
use ρ,
```

Then with simple calculations, we can show that both projections are homomorphisms with respect to $(G(f), \rho)$.

□

Theorem 4.7.2. *Given the coalgebras $\mathbb{A}, \mathbb{B}$ and $\mathbb{P}$ with the homomorphisms $\varphi_{\mathbb{A}} : \mathbb{P} \to \mathbb{A}$ and $\varphi_{\mathbb{B}} : \mathbb{P} \to \mathbb{B}$, the relation:*

$$(\varphi_{\mathbb{A}}, \varphi_{\mathbb{B}})[P] = \{(\varphi_{\mathbb{A}}(p), \varphi_{\mathbb{B}}(p)) \mid \ p \in P\} =$$

$$\{(a, b) \mid \ a \in A \ \wedge \ b \in B \wedge \exists p \in P. \ \varphi_{\mathbb{A}}(p) = a \ \wedge \ \varphi_{\mathbb{B}}(p) = b\}$$

is a bisimulation between $\mathbb{A}$ and $\mathbb{B}$ and every bisimulation between $\mathbb{A}$ and $\mathbb{B}$ is of the same shape.

Proof. The second statement matches the definition of bisimulations and is not worth writing as part of the proof.

```
theorem shape_of_bisimulation :
    (Π (P : Coalgebra F) (φ₁ : P ⟶ 𝔸) (φ₂ : P ⟶ 𝔹),
        let R : set (𝔸 × 𝔹) :=
            λ ab : 𝔸 × 𝔹 , ∃ p ,φ₁ p = ab.1 ∧ φ₂ p = ab.2 in
      is_bisimulation R)
    := ...
```

The first statement is also relatively simple to prove with few calculations, if we define the coalgebra structure for the bisimulation as such:

$$\rho : (\varphi_{\mathbb{A}}, \varphi_{\mathbb{B}})[P] \to F((\varphi_{\mathbb{A}}, \varphi_{\mathbb{B}})[P]) := F\varphi \circ \alpha_{\mathbb{P}} \circ \varphi^{-1}$$

Where $\varphi : P \to (\varphi_{\mathbb{A}}, \varphi_{\mathbb{B}})[P]$ is the surjective map $\varphi(p) = (\varphi_{\mathbb{A}}(p), \varphi_{\mathbb{B}}(p))$.

```
let φ : P.carrier → R := λ p,
    let φ_p : 𝔸 × 𝔹 := ⟨φ₁ p,  φ₂ p⟩ in
    have φ_p_R : φ_p ∈ R := exists.intro p
        ⟨(by simp : φ₁ p = φ_p.1), (by simp : φ₂ p = φ_p.2)⟩,
    ⟨φ_p,  φ_p_R⟩,
have sur : surjective φ := λ r, by tidy,
let μ : R → P := surj_inv sur,
let ρ : R → F.obj R := (F.map φ) ∘ P.α ∘ μ,
```

□

5. Review of Lean

In this chapter, we discuss different aspects of the Lean theorem prover. This is a personal review and it focuses on the learning process and usability, assuming no previous experience with any automated proof assistants.

Syntax

Lean's syntax is very elegant. The combination of unicode symbols with the ability to define flexible notation allows for precise and readable definitions, like the definition of coequalizer:

```
def coequalizer : B → B_θ f g := λ b, ⟦b⟧
```

The syntax also allows for some flexibility. We can define an object of a structure in any of the follow ways:

```
def p1 (x : X) (y : Y) : X × Y := ⟨x , y⟩
def p2 (x : X) (y : Y) : X × Y := {fst := x, snd := y}
def p3 (x : X) (y : Y) : X × Y := prod.mk x y
```

To introduce variables we have `λ` and `assume` and in tactic mode `intro` and `intros`. This flexibility, however, can also cause some confusion. For instance, inside tactic mode we can not deconstruct variables in the declaration:

```
def add1 : (ℕ × ℕ) → ℕ := assume ⟨n , m⟩, n + m
def add_tactic : (ℕ × ℕ) → ℕ :=
    begin
        intro nm, -- we can not use ⟨n , m⟩ even with assume
        exact nm.1 + nm.2
    end

example (h0 : ∃ x, p x) (h1 :∀ x, p x → q x):
    ∃ x, q x :=
    let ⟨x, px⟩ := h0 in     -- does not work in tactic mode
        ⟨x, h1 x px⟩
```

We can also point to the use of the plural form `variable` and `variables`, `intro` and `intros`. These are simple examples of many inconsistencies, which makes writing code in Lean frustrating at times, especially for beginners.

Automation

We will discuss three tactics for automation, of which we make heavy use in the previous chapters.

We start with the rewrite tactic `rw`. While it is not exactly an automation tactic, it is supposed to make writing a proof more convenient and efficient according to [8]. It is certainly a useful tactic. However, it does not apply coersions and it does not unfold definitions, that Lean would otherwise automatically do, as the following example shows:

```
example (p : g ⊚ f = h ⊚ f) : g ∘ f = h ∘ f :=
    calc g ∘ f = g ⊚ f : rfl
           ... = h ⊚ f : by rw p
```

We can not use `rw p` directly, because `rw` does not apply reflexivity nor coercion, despite the fact that Lean accepts the proof as it is here and does the last step `h ⊚ f = h ∘ f` by itself. Instances, similar to this example, can be found at many places in our code.

The tactic `simp` uses the tactic `rw` and many more identities defined in Lean. It is supposed to offer a powerful form of automation. However, here is a list of simple proofs, that the simplifier would fail to do:

```
example (comp : g ∘ f = h) (x : X) : g (f x) = h x :=
   -- simp [comp] does not work
    have s : (g ∘ f) x = h x := by rw comp,    -- this works
    s     -- Lean accepts s as a proof.

example (h1 : f = g) (x₁ x₂: X) (h2 : f x₁ = f x₂)
      : g x₁ = g x₂ := h1 ▹ h2      -- this works
      -- simp [h1 , h2]    does not work

example  : p → ¬ (p ∧ q) → ¬ q :=
    λ (hp : p) (hnpq: ¬ (p ∧ q)) (hq : q),
    absurd (and.intro hp hq) hnpq
    -- simp [hp, hnpq, hq]     does not work

example (S: set X) (x₁ x₂ : S) : x₁.val = x₂.val → x₁ = x₂ :=
    λ h : x₁.val = x₂.val, by simp [h] -- does not work
```

Finally we look at the tactic `tidy`. This is probably the most powerful automation tactic. It is defined in the *mathlib* as opposed to `rw` and `simp`, which are defined in standard library. `tidy` is indeed very powerful and is able to prove all the previous examples. It applies a list of tactics repeatedly to the goal and recursively on new goals until none of them makes any progress.

However, in a few instances, `tidy` generates a proof, that is rejected by Lean. As we mentioned before, when defining the signature for the coalgebra of Kripke-structures (see 4.2.3). Using `tidy` in that proof gives the following error message:

```
type mismatch at application
  has_mem.mem x
term
  x
has type
  X_1
but is expected to have type
  X
types contain aliased name(s): X
remark: the tactic ‘dedup‘ can be used to rename aliases
```

Where all `x`, `X_1` and `X` are generated by `tidy` itself.

Using the automation tools is often trial and error. One has to develop a feeling for when it works and when it does not. However, it is clear that there is still a huge room for improvement.

Usability

In this section we will talk about smaller issues one faces, when using Lean:

- Lean provides a messages windows, that appears next to the editor (when using VS code). This window is very useful to keep track of the current variables and assumptions and the current goal or goals. However, this window only appears in tactic mode. This makes writing complicated proofs very hard without entering tactic mode.
- Error messages in Lean are not beginner friendly and often they point to the wrong place of the code. However, that is not always the case. More experienced users can depend on the error messages to keep track of the stage of the proof.
- Lean's documentation is good for getting started and writing very simple proofs. After that, there is little support for advanced users. However, the community on leanprover.zulipchat.com/ and stackoverflow.com is active and very helpful, albeit rather small.
- In our experience, we found Lean's performance to be excellent. However, our files never exceeded 500 lines of code. The only performance issue we found is that Lean reevaluates the entire file every time one edits something. It would save significant amount of time if the evaluation was more scope-limited.

6. Conclusion and Future Work

In this thesis, we explored the Lean theorem prover and looked at the formalization of category theory in it. We then added to the existing library proofs of the diagram lemmas and a formalization of the E-M-squares as well as simple definitions of certain limits and colimits with proofs of their existence in the category Set.

Building on that, we formalized the definition of coalgebras, coalgebra homomorphisms and the category Set_F. Using these definitions, we defined the coalgebras representing automata and showed the equivalence between the definitions of homomorphisms between coalgebras and homomorphisms between automata; and then we did the same with Kripke-structures. We then used the definitions of limits and colimits we wrote in chapter 3 to define certain limits and colimits in the category Set_F. At the end of this chapter, we formalized the definition of bisimulations and proved basic theories about them.

Lean's clean and elegant syntax along with the small size of its trusted kernel of axioms makes it very appealing as a tool to verify mathematical arguments. However, the steep learning curve and the weaknesses in the automation tools, as we discussed in chapter 5, keeps it from becoming more mainstream among mathematicians.

The field of coalgebra contains many other areas, that were not covered here and could still be formalized in future work. Among these areas are terminal coalgebras and coalgebraic modal logic. Additionally, the proofs included in this work could be broken down to smaller and more usable lemmas and theories, in order to integrate this project into Lean's mathematical library. Furthermore, when *Lean 4* is released, there might be a need to update this and many other projects accordingly.

Bibliography

[1] Thomas C Hales. A Proof of the Kepler Conjecture. *Annals of mathematics*, pages 1065–1185, 2005.

[2] Philip Wadler. Propositions as Types. *Commun. ACM*, 58(12):75–84, 2015.

[3] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. The Lean Theorem Prover (system description). In Amy P. Felty and Middeldorp Aart, editors, *International Conference on Automated Deduction*, pages 378–388. Springer, 2015.

[4] H. Peter Gumm. Universelle Coalgebra. *Th. Ihringer, Allgemeine Algebra, Heldermann Verlag*, 2003.

[5] William DeMeo. LEAN-UALIB. https://williamdemeo.gitlab.io/lean-ualib/. Accessed: September 23, 2019.

[6] William M. Farmer. The Seven Virtues of Simple Type Theory. *Journal of Applied Logic*, 6(3):267–286, 2008.

[7] Steven Roman. *An Introduction to the Language of Category Theory*. Compact Textbooks in Mathematics. Springer, 2017.

[8] Jeremy Avigad, Leonardo de Moura, and Soonho Kong. Theorem Proving in Lean. Release 3.4.0. https://leanprover.github.io/theorem_proving_in_lean/theorem_proving_in_lean.pdf, 2019. Accessed: September 23, 2019.

A. Lean Code

A.1. Category Theory

A.1.1. Diagram lemmas

```
lemma diagram_surjective (f : A ⟶ B) (g : A ⟶ C)
(sur: surjective f)
: (∃! h : B ⟶ C , h ∘ f = g) ↔ (sub_kern f g)
:= iff.intro
begin
   assume ex : ∃ h , (h ∘ f = g ∧ ∀ h₁, h₁ ∘ f = g → h₁ = h),
   show ∀ a₁ a₂ , kern f a₁  a₂ → kern g a₁ a₂,
   let h := some ex,
   have h0 := some_spec ex,
   have h1 : h ∘ f = g := and.left h0,
   have s1 : sub_kern f (h ∘ f) := kern_comp f h,
   exact h1 ▹ s1       -- short for eq.subst h1 s1
end
begin
   assume k : ∀ a b, f a = f b → g a = g b,
   show ∃ h , (h ∘ f = g ∧ ∀ h₁, h₁ ∘ f = g → h₁ = h),
   let h : B → C := λ b : B, g (surj_inv sur b),
   have s1 : ∀ a₁ a₂ , f a₁ = f a₂ → g a₁ = g a₂ := k,
   have s2 : ∀ a , f (surj_inv sur (f a)) = f a :=
         assume a , surj_inv_eq sur (f a),
   have s3 : ∀ a , g (surj_inv sur (f a)) = g a :=
         assume a , s1 (surj_inv sur (f a)) a (s2 a),
   have s4 : ∀ a , h (f a) = g a :=
         assume a,
         show g (surj_inv sur (f a)) = g a,
         from s3 a,
   have s5:  h ∘ f = g := funext s4,
   have s6 : ∀ h₂ :B → C , h₂ ∘ f = g → h₂ = h :=
      begin
         assume h₂ h2,
         have s61 : h₂ ∘ f = h ∘ f := by simp [s5 , h2],
         haveI s62 : epi f := (epi_iff_surjective f).2 sur,
         have left_cancel := s62.left_cancellation,
```

```
        have s63 : h₂ = h := left_cancel h₂ h s61,
        tidy
      end,
   have s7 : ∃ (h₁ : B → C), h₁ ∘ f = g ∧
            ∀ h₂, h₂ ∘ f = g → h₂ = h₁ :=
            exists.intro h (and.intro (funext s4) s6),
   tidy
end
```

```
lemma diagram_injective (f: B ⟶ A) (g: C ⟶ A)
               (inj : injective f)
   :(∃! h : C ⟶ B , f ∘ h = g) ↔ (range g ⊆ range f)
   := iff.intro
begin
   tidy
end
begin
   assume im,
   let G : C → B → Prop := λ c b , g c = f b,
   have G1 :  ∀ c : C , ∃ b : B, G c b
   :=
   have G10 : ∀ a : A , a ∈ range g →
               a ∈ range f := im,
      have G11 : ∀ c : C , g c ∈ range f :=
         λ c ,
         have G110 : g c ∈ range g := by tidy,
         G10 (g c) G110,
      have G12 : ∀ c : C , ∃ b : B , g c = f b :=
         λ c : C,
         have G110 : g c ∈ range f := G11 c,
         by tidy,
      G12,
   have G2 :  ∀ c : C , ∃! b : B, G c b :=
      λ c,
      have G20 : G c (some (G1 c)) := some_spec (G1 c),
      have G21 : f (some (G1 c)) = g c :=
         show f (some (G1 c)) = g c,
         from
         have G210 : _ := G c (some (G1 c)),
         by tidy,
      have G22 : ∀ b₁ : B, G c b₁ →
         b₁ = (some (G1 c)) :=
         λ b₁ : B, assume cbG : G c b₁,
         show b₁ = (some (G1 c)), from
```

```
        have G220 : f b₁ = g c := by tidy,
        have G221 : f (some (G1 c)) = f b₁ := by rw [G21 , G220],
        eq.symm (inj G221),  by tidy,
  let h : C ⟶ B := graph_to_map G G2,
  have G3 : ∀ c , (f ∘ h) c = g c:=
     assume c,
     have G31 : h c = some (G2 c) := by tidy,
     have G32 : _ := some_spec (G2 c),
     have G33 : G c (some (G2 c)) := and.left G32,
     by tidy,
  have G4 : f ∘ h = g := funext G3,
  have G5 : ∀ h₁ : C ⟶ B , f ∘ h₁ = g → h₁ = h :=
     assume h₁ fh,
     have G51 : f ∘ h₁ = f ∘ h := by rw [fh , G4],
     have G511 : f ⊚ h₁ = f ⊚ h := by tidy,
     have G52 : mono f := iff.elim_right (mono_iff_injective f) inj,
     have G53 : _ := G52.right_cancellation,
     G53 h₁ h G511,
  exact exists_unique.intro h G4 G5
end
```

A.1.2. Orthogonality

```
lemma commutative_triangles_epi
    {C : Type v} [category C] {X Y Z U : C}
          (e : X ⟶ Y) (f : Y ⟶ U)
          (g : X ⟶ Z) (m : Z ⟶ U)
          (h : f ⊚ e = m ⊚ g) (d : Y ⟶ Z)
          [epi e] :
    (g = d ⊚ e) → (∀ d₁ : Y ⟶ Z , g = d₁ ⊚ e → d₁ = d)
                   ∧ f = m ⊚ d
        :=
    begin
        intros g_ed,
        split,

        intros d₁ g_ed₁,
        exact eq.symm (right_cancel e (g_ed ▹ g_ed₁)),

        have ef_edm : f ⊚ e = m ⊚ d ⊚ e := by simp [g_ed, h],
        exact right_cancel e ef_edm
    end
```

```
lemma commutative_triangles_mono
        {C : Type v} [category C] {X Y Z U : C}
          (e : X ⟶ Y) (f : Y ⟶ U)
          (g : X ⟶ Z) (m : Z ⟶ U)
          (h : f ⊚ e = m ⊚ g) (d : Y ⟶ Z)
          [mono m] :
    f = m ⊚ d → (∀ d₁ : Y ⟶ Z , f = m ⊚ d₁ → d₁ = d)
                 ∧ g = d ⊚ e
        :=
    begin
        assume f_dm,

        split,

        assume d₁ f_dm₁,
        exact eq.symm (left_cancel m (f_dm ▹ f_dm₁)),

        have edm_gm : m ⊚ d ⊚ e = m ⊚ g := f_dm ▹ h,
        have edm_gm1: m ⊚ (d ⊚ e) = m ⊚ g := by tidy,
        exact eq.symm (left_cancel m edm_gm1)
    end
```

```
lemma E_M_square {X Y Z U : Type u}
          (e : X ⟶ Y) (ep : epi e)
          (f : Y ⟶ U) (g : X ⟶ Z)
          (m : Z ⟶ U) (mo : mono m)
          (h : f ⊚ e = m ⊚ g) :
          ∃! d : Y ⟶ Z , (g = d ⊚ e ∧
                          f = m ⊚ d) :=
begin
    have sur : surjective e := (epi_iff_surjective e).1 ep,
    have inj : injective m  := (mono_iff_injective m).1 mo,

    have range_f_m : range f ⊆ range m :=
        calc range f = range (f ⊚ e)    : eq_range_if_surjective e f sur
                ...  = range (m ⊚ g)    : by rw h
                ...  ⊆ range m         : range_comp_subset_range g m,

    have kern_e_g : sub_kern e g :=
        sub_kern_if_injective e f g m h inj,

    cases ((diagram_injective m f inj).2 range_f_m) with d₁ ex_uni1,
    cases ((diagram_surjective e g sur).2 kern_e_g)  with d₂ ex_uni2,
```

```
    have ex1 : m ∘ d₁ = f := ex_uni1.1,
    have ex2 : d₂ ∘ e = g := ex_uni2.1,

    have uni1 : ∀ d : Y ⟶ Z,  m ∘ d = f → d = d₁ := ex_uni1.2,

    have em_mono : _ := (commutative_triangles e f g m h d₁),

    have h1 : (∀ (d₁₁ : Y ⟶ Z), f = m ⊚ d₁₁ → d₁₁ = d₁)
                    ∧ g = d₁ ⊚ e :=

        and.right em_mono ⟨mo , eq.symm ex1⟩,
    have h2 : ∀ (d₁₁  : Y ⟶ Z), f = m ⊚ d₁₁ → d₁₁ = d₁ :=
        assume d₁₁, and.left h1 d₁₁,

    have em_epi :=
        (commutative_triangles e f g m h d₂).1 ⟨ep ,eq.symm ex2⟩,

    have h4 : ∀ (d₁₁  : Y ⟶ Z), g =  d₁₁ ⊚ e → d₁₁ = d₂ :=
        assume d₁₁, and.left em_epi d₁₁,

    have d11 : g = d₁ ⊚ e := and.right h1,
    have d22 : f = m ⊚ d₂ := and.right em_epi,

    have d1_d2 : d₂ = d₁ := and.left h1 d₂ d22,
    have d12 : f = m ⊚ d₁ := eq.subst d1_d2 d22,

    have h5 : ∀ dₓ : Y ⟶ Z ,
            (g = dₓ ⊚ e ∧ f = m ⊚ dₓ) → dₓ = d₁ :=
            assume dₓ ged_fdm,
            uni1 dₓ (eq.symm ged_fdm.2),

    exact exists_unique.intro d₁ ⟨d11 , d12⟩ h5
end
```

```
lemma eq_range_if_surjective  (f: A ⟶ B) (g: B ⟶ C)
    (sur: surjective f) : range g = range (g ∘ f):=
    calc range g = image g (univ)          : by simp
          ...    = image g (range f)       : by rw
                               range_iff_surjective.2 sur
          ...    = range (g ∘ f)           : by tidy
```

```
lemma sub_kern_if_injective {X Y Z U : Type u}
    (e : X ⟶ Y) (f : Y ⟶ U)
```

```
    (g : X ⟶ Z) (m : Z ⟶ U)
    (h : e ≫ f = g ≫ m) (inj: injective m) : sub_kern e g :=
        begin
            assume x₁ x₂ xxe,
            have h01 : e x₁ = e x₂ := xxe,
            have h02 : m (g x₁) = m (g x₂) :=
            calc m (g x₁) = (g ≫ m) x₁     : rfl
                     ...     = (e ≫ f) x₁     : by rw h
                     ...     = f (e x₁)        : rfl
                     ...     = f (e x₂)        : by rw h01
                     ...     = (e ≫ f) x₂     : rfl
                     ...     = (g ≫ m) x₂     : by rw h,
            exact inj h02
        end
```

A.1.3. Equalizer

```
lemma eqaulizer_set_is_equalizer :
    is_equalizer f g (eqaulizer_set f g ↪ A) :=
    let E := eqaulizer_set f g in
    let e := E ↪ A in
    ⟨
        have elements : ∀ a, (f ∘ e) a = (g ∘ e) a :=
                λ a, a.property,
        funext elements
        ,
        begin
            intros Q q fq_gq,
            have s0 : ∀ b : Q , (f ⊚ q) b = (g ⊚ q) b :=
                        assume b, by rw fq_gq,
            have s1 : ∀ b : Q , q b ∈ E :=
                    assume b, s0 b,
            let h : Q → E := λ b, ⟨q b, s1 b⟩,
            have q_eh : q = e ∘ h := by tidy,
            use h,
            split,
            exact q_eh,
            intros h₁ spec_h₁,
            tidy,
            have inj : injective e := inj_inclusion A E,
            have ey_eh: e ∘ h₁ = e ∘ h := by rw [← spec_h₁ , q_eh],
            have elements : ∀ b, (e ∘ h₁) b = (e ∘ h) b :=
                    assume a, by rw ey_eh,
```

```
            dsimp at *,
            solve_by_elim
        end
    ⟩
```

A.1.4. Product

```
lemma cartesian_product_is_product :
    is_product A B prod.fst prod.snd :=
        begin
            intros Q q₁ q₂,
            let p : Q → (A × B) := λ k, ⟨q₁ k,q₂ k⟩,
            use p,
            tidy
        end
```

```
lemma jointly_mono
    {X : Type v} [category X]
    (A B P : X) {Q: X} (π₁ : P ⟶ A) (π₂ : P ⟶ B)
        (prod: is_product A B π₁ π₂)
        {s s₁: Q ⟶ P}
        (h1 : π₁ ⊚ s₁ = π₁ ⊚ s)
        (h2 : π₂ ⊚ s₁ = π₂ ⊚ s):
        s₁ = s :=
    begin
        have prod_Q := prod Q (π₁ ⊚ s₁) (π₂ ⊚ s₁),
        cases prod_Q with p spec_p,
        have spec_s1 : π₁ ⊚ s = π₁ ⊚ p :=
            h1 ▹ spec_p.1.1,
        have spec_s2 : π₂ ⊚ s = π₂ ⊚ p :=
            h2 ▹ spec_p.1.2,
        rw spec_p.2 s ⟨h1, h2⟩,
        exact spec_p.2 s₁ ⟨rfl, rfl⟩,
    end
```

A.1.5. Pullback

```
lemma equalizer_product_is_pullback_cat
    {X : Type u} [category X]
    {A₁ A₂ B P E : X}
    (f : A₁ ⟶ B) (g : A₂ ⟶ B)
    (π₁ : P ⟶ A₁) (π₂ : P ⟶ A₂)
```

```
(pr : is_product A₁ A₂ π₁ π₂)
(e : E ⟶ P)
(eqauliz : is_equalizer (f ⊚ π₁) (g ⊚ π₂) e) :
is_pullback f g
    (π₁ ⊚ e)
    (π₂ ⊚ e)
:=
⟨ begin
    tidy
 end
,
begin
    intros Q q₁ q₂ fq₁_gq₂,
    let p : Q ⟶ P := some (pr  Q q₁ q₂),
    have spec_p : q₁ = π₁ ⊚ p ∧ q₂ = π₂ ⊚ p :=
        (some_spec (pr Q q₁ q₂)).1,

    have eq_comp : f ⊚ π₁ ⊚ p  = g ⊚ π₂ ⊚ p :=
        calc f ⊚ π₁ ⊚ p   = f ⊚ (π₁ ⊚ p)    : by tidy
                ...         = f ⊚ q₁           : by rw ← spec_p.1
                ...         = g ⊚ q₂           : fq₁_gq₂
                ...         = g ⊚ (π₂ ⊚ p)    : by rw spec_p.2
                ...         = g ⊚ π₂ ⊚ p      : by tidy,

    let h : Q ⟶ E := some (eqauliz.2 p eq_comp),
    have spec_h : p = e ⊚ h :=
        (some_spec (eqauliz.2 p eq_comp)).1,
    use h,
    have h0 : q₁ = π₁ ⊚ e ⊚ h ∧ q₂ = π₂ ⊚ e ⊚ h :=
        ⟨
            by simp [spec_h , spec_p.1]
            ,
            by simp [spec_h , spec_p.2]
        ⟩ ,
    split,
    exact h0,
    assume (y : Q ⟶ E)
        (spec_y : q₁ = π₁ ⊚ e ⊚ y ∧ q₂ = π₂ ⊚ e ⊚ y),
    have s0 : π₁ ⊚ (e ⊚ h) = π₁ ⊚ (e ⊚ y) :=
        calc π₁ ⊚ (e ⊚ h) = π₁ ⊚ e ⊚ h       : by tidy
            ...             = q₁                 : eq.symm h0.1
            ...             = π₁ ⊚ e ⊚ y       : spec_y.1
            ...             = π₁ ⊚ (e ⊚ y)     : by tidy,
    have s1 : π₂ ⊚ (e ⊚ h) = π₂ ⊚ (e ⊚ y) :=
```

```
        calc π₂ ⊚ (e ⊚ h) = π₂ ⊚ e ⊚ h       : by tidy
        ...                = q₂                : eq.symm h0.2
        ...                = π₂ ⊚ e ⊚ y       : spec_y.2
        ...                = π₂ ⊚ (e ⊚ y)     : by tidy,
    have eh_ey : e ⊚ h = e ⊚ y :=
        jointly_mono A₁ A₂ P π₁ π₂ pr s0 s1,

    haveI m_e : mono e :=
        equalizer_is_mono (f ⊚ π₁) (g ⊚ π₂) e eqauliz,
    exact left_cancel e (eq.symm eh_ey)
end ⟩
```

A.1.6. Coequalizer

```
lemma coequalizer_kern :
    Π (Q : Type u) (q : B → Q),
    q ∘ f = q ∘ g → sub_kern (coequalizer f g) q :=
    begin
        intros Q q qfg,
        let co := (coequalizer f g),
        let ker := kern co,
        let ker_q := kern q,
        have k1 : ∀ b₁ b₂ , ker b₁ b₂ → ker_q b₁ b₂ :=
            begin
                assume b₁ b₂ kb1b2,
                have quotb1b2 : ⟦b₁⟧ = ⟦b₂⟧  := kb1b2,
                let Θb1b2 : eqv_gen (R f g) b₁ b₂ :=
                    @quotient.exact B (theta_setoid f g)
                    b₁ b₂
                    quotb1b2,
                apply eqv_gen.rec_on Θb1b2,
                exact (λ b₁ b₂ (h: ∃ a : A , f a = b₁ ∧ g a = b₂),
                    let a : A := some h in
                    calc q b₁ = (q ∘ f) a    : by simp [some_spec h]
                          ... = (q ∘ g) a    : by simp [qfg]
                          ... = q b₂         : by simp [some_spec h]),
                exact (λ x, rfl),
                exact (λ x y _ (h : q x = q y), eq.symm h),
                exact (λ x y z _ _ (h₁ : q x = q y) (h₂ : q y = q z),
                     eq.trans h₁ h₂),
            end,
        exact k1
    end
```

```
lemma quot_is_coequalizer
    : is_coequalizer f g (coequalizer f g) :=
    ⟨   funext (quot_equalizes_all f g),

        begin
            intros Q q qfg,
            let co := (coequalizer f g),
            have sub_ker : sub_kern co q := coequalizer_kern f g Q q qfg,
            exact (diagram_surjective co q
                     (quot_is_surjective f g)).elim_right
                  sub_ker
        end
    ⟩
```

A.1.7. Sum

```
lemma jointly_epi_cat {X : Type v} [category X]
    {A B S Q : X} (e₁ : A ⟶ S) (e₂ : B ⟶ S)
    (is_sm : is_sum S e₁ e₂)
        (f g: S ⟶ Q)
        (h1 : f ⊚ e₁ = g ⊚ e₁)
        (h2 : f ⊚ e₂ = g ⊚ e₂)
    : f = g :=
    begin
        have ex : _ := is_sm Q (f ⊚ e₁) (f ⊚ e₂),
        cases ex with s spec_s,
        have g_s : g = s := spec_s.2 g ⟨h1, h2⟩,
        rw g_s,
        exact spec_s.2 f ⟨rfl, rfl⟩
    end
```

```
lemma jointly_epi  {A B Q : Type u}
                    (s s₁: (A ⊕ B) → Q)
                    (h1 : s₁ ∘ inl = s ∘ inl)
                    (h2 : s₁ ∘ inr = s ∘ inr)
    : s₁ = s :=
    begin
        have all_ab : ∀ ab : (A ⊕ B) , s₁ ab = s ab :=
            begin
                intro ab,
                induction ab,
                case inl :
```

```
                {
                have s0 : (s₁ ∘ inl) ab = (s ∘ inl) ab
                    := by rw h1,
                exact s0
                },
            case inr :
                begin
                    have s0 : (s₁ ∘ inr) ab = (s ∘ inr) ab
                        := by rw h2,
                    exact s0
                end,
        end,
        exact funext all_ab
    end
```

```
lemma disjoint_union_is_sum :
    is_sum  (A ⊕ B) inl inr :=
    begin
        intros Q q₁ q₂,          -- competitor with 2 morphisms
        show ∃! s : (A ⊕ B) ⟶ Q, (q₁ = s ∘ inl ∧ q₂ = s ∘ inr)
                , from
        let s : (A ⊕ B) ⟶ Q :=
         --defining the unique morphism s : sum → Q inductively
            begin
                intro ab,
                induction ab,
                case inl :
                    begin
                        exact q₁ ab
                    end,
                case inr :
                    begin
                        exact q₂ ab
                    end,
            end in

        have commut_A : q₁ = s ∘ inl := rfl,
        have commut_B : q₂ = s ∘ inr := rfl,
        have unique : ∀ s₁ ,
                        (q₁ = s₁ ∘ inl ∧ q₂ = s₁ ∘ inr) →
                           s₁ = s :=
            assume s₁ h₁,
                jointly_epi s s₁
                    (by rw [← h₁.1 ,commut_A ])
```

```
                        (by rw [← h₁.2 ,commut_B ]),

        exists_unique.intro s
            ⟨commut_A , commut_B⟩ unique
    end
```

A.1.8. Pushout

```
lemma coequalizer_sum_is_pushout
    {X : Type v} [category X]
    {A B₁ B₂: X} (f : A ⟶ B₁) (g : A ⟶ B₂)
    (S P : X) (s₁ : B₁ ⟶ S) (s₂ : B₂ ⟶ S)
    (p : S ⟶ P)
    (is_sm : is_sum S s₁ s₂)
    (is_coeq: is_coequalizer (s₁ ⊚ f) (s₂ ⊚ g) p):
    is_pushout f g P (p ⊚ s₁) (p ⊚ s₂)
    :=
    begin
        have comm : (p ⊚ s₁) ⊚ f = (p ⊚ s₂) ⊚ g :=
            by tidy,
        split,
        exact comm,
        intros Q q₁ q₂ qfg,
        have sm : _ := is_sm Q q₁ q₂,
        let s := some sm,
        have spec : q₁ = s ⊚ s₁ ∧ q₂ = s ⊚ s₂
            := (some_spec sm).1,
        have sf_sg : s ⊚ s₁ ⊚ f = s ⊚ s₂ ⊚ g :=
            calc (s ⊚ s₁) ⊚ f
                    = q₁ ⊚ f       : by rw ← spec.1
                ... = q₂ ⊚ g       : qfg
                ... = s ⊚ s₂ ⊚ g : by rw spec.2,
        have coeq : ∃! (h : P ⟶ Q), h ⊚ p = s
         := is_coeq.2 Q s (by tidy),

        let h := some coeq,
        have h_spec : s = h ⊚ p := eq.symm (some_spec coeq).1,

        use h,
        split,
        split,
        by simp [h_spec , spec.1],
```

```
        by simp [h_spec , spec.2],
        intros q spec_q,
        have h01 : q₁ = q ⊚ p ⊚ s₁ := by tidy,
        have h02 : q₂ = q ⊚ p ⊚ s₂ := by tidy,

        have q_s : q ⊚ p = s :=
                jointly_epi_cat s₁ s₂ is_sm (q ⊚ p) s
                    (spec.1 ▹ (eq.symm h01))
                    (spec.2 ▹ (eq.symm h02)),

        exact (some_spec coeq).2 q q_s
    end
```

A.2. Coalgebra

```
lemma id_is_hom (𝔸 : Coalgebra F) : is_coalgebra_homomorphism (@id 𝔸) :=
   show 𝔸.α ∘ id =  (F.map id) ∘ 𝔸.α, from
   calc 𝔸.α ∘ id = (𝟙 (F.obj 𝔸)) ∘ 𝔸.α      : rfl
        ...      = (F.map (𝟙 𝔸)) ∘ 𝔸.α      : by rw
                            ← functor.map_id' F 𝔸
```

```
lemma comp_is_hom (φ : homomorphism 𝔸 𝔹) (ψ : homomorphism 𝔹 ℂ)
                  : is_coalgebra_homomorphism (ψ ∘ φ) :=
   have ab : 𝔹.α ∘ φ =  F.map φ ∘ 𝔸.α := φ.property,
   have bc : ℂ.α ∘ ψ =  F.map ψ ∘ 𝔹.α := ψ.property,
   calc
          (ℂ.α ∘ ψ) ∘ φ = (F.map ψ) ∘ 𝔹.α  ∘ φ     : by rw bc
          ... = (F.map ψ) ∘ (F.map φ) ∘ 𝔸.α       : by rw ab
          ... = ((F.map ψ) ⊚ (F.map φ)) ∘ 𝔸.α    : rfl
          ... = (F.map (ψ ⊚ φ)) ∘ 𝔸.α            : by rw
                            ← functor.map_comp
```

A.2.1. Automata as Coalgebras

```
lemma Automata_Coalgebra_hom {A B : Automaton Sigma Γ} (φ : A → B):
    is_homomorphism φ ↔
    @is_coalgebra_homomorphism
         F (Automata_Coalgebra A) (Automata_Coalgebra B) φ :=
        let 𝔸 := Automata_Coalgebra A in
        let 𝔹 := Automata_Coalgebra B in
```

```
    begin
        split,
        intro h,
        dsimp at *,
        ext1 s,
        dsimp at *,
        have hs := h s,
        ext1,
        have B_fst : (𝔹.α (φ s)).fst = B.γ (φ s) :=
            by tidy,
        have A_fst :
         (F.map φ ((Automata_Coalgebra A).α s)).fst = B.γ (φ s) :=
            by tidy,
        simp [B_fst , A_fst],
        have A_snd : (𝔹.α (φ s)).snd = ((F.map φ) (𝔸.α s)).snd :=
            by tidy,
        exact A_snd,

        intro co_h,
        dsimp at *,
        intro s,
        have h_s : (𝔹.α ∘ φ) s =
                (F.map φ ∘ 𝔸.α) s := by rw co_h,
        split,
        tidy
    end
```

A.2.2. Kripke Structures as Coalgebras

```
variable {φ : Type v}
def F : Type u ⥤ Type (max v u) :=
{
    obj := λ S, (set S) × (set φ),
    map := λ {A B} φ, λ ⟨U , P⟩ , ⟨image φ U, P⟩,
    map_id' :=
        begin
            intros X,
            dsimp at *,
            ext1,
            cases x with U P,
            dsimp at *,
            ext1,
            have im : image id U = U :=  by simp,
```

```
            exact im,
            refl
        end ,
    map_comp' :=
        begin
            intros X Y Z f g,
            dsimp at *,
            ext1,
            cases x with S P,
            dsimp at *,
            calc
            F._match_1 (g ∘ f) (S, P)
                = ⟨image (g ∘ f) S, P⟩              : rfl
            ... = ⟨image g (image f S), P⟩          : by rw
                                        img_comp f g S
            ... = F._match_1 g (F._match_1 f (S, P)) : rfl
        end
}
```

```
lemma Kripke_Coalgebra_hom {K₁ K₂ : Kripke φ} (φ : K₁ → K₂):
    is_homomorphism φ φ ↔
    @is_coalgebra_homomorphism F
        (Kripke_Coalgebra K₁) (Kripke_Coalgebra K₂)
            φ :=
    let 𝕂₁  := Kripke_Coalgebra K₁ in
    let 𝕂₂ := Kripke_Coalgebra K₂ in
    iff.intro
    (
    assume ⟨tr, b_a, pr⟩,
    show 𝕂₂.α ∘ φ = F.map φ ∘ 𝕂₁.α,
    begin
        ext1,
        have im_elem : ∀ s : K₂.State ,
                s ∈ image φ (K₁.T x) ↔ s ∈ K₂.T (φ x) :=
            begin
                intro s,
                split,
                intros im_φ,
                cases im_φ with s12 specS12,
                rw ← specS12.2,
                exact tr x s12 specS12.1,
                intro s_T2,
                exact b_a x s s_T2,
            end,
```

```
    have im : image φ (K₁.T x) = K₂.T (φ x) :=
        eq_sets.1 im_elem,

    have F_φ : F.map φ (𝕂₁.α x) = ⟨image φ (K₁.T x), K₁.v x⟩
            := rfl,
    simp [F_φ],
    have last : (⟨K₂.T (φ x) , K₂.v (φ x)⟩ :
        (set K₂.State) × set φ) = ⟨image φ (K₁.T x), K₁.v x⟩ :=
            by simp [eq.symm im, eq.symm (pr x)],
    simp [eq.symm last],
    refl
end )
(
assume co_hom : 𝕂₂.α ∘ φ = F.map φ ∘ 𝕂₁.α,
show (∀ a₁ a₂ :K₁ , a₂ ∈ K₁.T a₁ → (φ a₂) ∈ K₂.T (φ a₁)) ∧
     (∀ (a : K₁) (b : K₂) , b ∈ K₂.T (φ a) →
            ∃ a’: K₁ ,  a’ ∈ K₁.T a ∧ φ a’ = b) ∧
     (∀ a : K₁ , K₁.v a = K₂.v (φ a)),
begin
    split,
    intros a₁ a₂ a₂_a₁,

    have h3 : (𝕂₂.α ∘ φ) a₁ = ((F.map φ) ∘ 𝕂₁.α) a₁
                := by rw co_hom,
    have h5 : (⟨K₂.T (φ a₁) , K₂.v (φ a₁)⟩ :
        (set K₂.State) × set φ) = ⟨image φ (K₁.T a₁), K₁.v a₁⟩
            := h3,
    have h6 : K₂.T (φ a₁) = image φ (K₁.T a₁) := by tidy,
    have h7 : φ a₂ ∈ image φ (K₁.T a₁) :=
        by {use a₂, simp [a₂_a₁]},
    rw h6,
    exact h7,

    split,
    intros a₁ b b_T_φ_a,
    have h3 : (𝕂₂.α ∘ φ) a₁ = ((F.map φ) ∘ 𝕂₁.α) a₁
                := by rw co_hom,
    have h4 : 𝕂₂.α (φ a₁) = F.map φ (𝕂₁.α a₁)
                := h3,
    have h5 : (⟨K₂.T (φ a₁) , K₂.v (φ a₁)⟩ :
        (set K₂.State) × set φ) = ⟨image φ (K₁.T a₁), K₁.v a₁⟩
            := h4,
    have h6 : K₂.T (φ a₁) = image φ (K₁.T a₁) := by tidy,
```

```
        have h7 : b ∈ image φ (K₁.T a₁) := h6 ▹ b_T_φ_a,
        exact h7,
        intro a₁,
        have h3 : (𝕂₂.α ∘ φ) a₁ = ((F.map φ) ∘ 𝕂₁.α) a₁
                    := by rw co_hom,
        have h5 : (⟨K₂.T (φ a₁) , K₂.v (φ a₁)⟩ :
            (set K₂.State) × set φ) = ⟨image φ (K₁.T a₁), K₁.v a₁⟩
                := h3,
        have h6 : K₂.v (φ a₁) = K₁.v a₁ := by tidy,
        exact eq.symm h6
    end)
```

```
lemma eq_sets {A : Type u} (S T : set A)
    : (∀a : A ,a ∈ S ↔ a ∈ T) ↔
     S = T  :=
    begin
        split,
        assume h,
        have h1 : ∀s : S , s.val ∈ T :=
           λ s, (h s.val).1 s.property,
        have h2 : ∀t : T , t.val ∈ S :=
           λ t, (h t.val).2 t.property,
        simp at *,
        ext1,
        split,
        intros s,
        exact h1 x s,
        intros t,
        exact h2 x t,
        intro h,
        intro a,
        induction h,
        refl
    end
```

A.2.3. Isomorphisms and diagram lemma

```
theorem bij_inverse_of_hom_is_hom
    (φ : homomorphism 𝔸 𝔹)
    (bij : bijective φ) :
        let inv : 𝔹 → 𝔸 := some (bijective_iff_has_inverse.1 bij) in
        is_coalgebra_homomorphism inv :=
        begin
```

```
        intro inv,
        let hom :𝔹.α ∘ φ =  F.map φ ∘ 𝔸.α := φ.property,
        have has_lr_inv :left_inverse inv φ ∧ right_inverse inv φ
            := some_spec (bijective_iff_has_inverse.1 bij),

        calc
        𝔸.α ∘ inv = id ∘ 𝔸.α ∘ inv                          : rfl
        ... = (𝟙 (F.obj 𝔸)) ∘ 𝔸.α ∘ inv                     : rfl
        ... = (F.map (𝟙 𝔸)) ∘ 𝔸.α ∘ inv                     : by rw
                          ← functor.map_id'
        ... = (F.map id) ∘ 𝔸.α ∘ inv                       : rfl
        ... = (F.map (inv ∘ φ)) ∘ 𝔸.α ∘ inv                : by rw
                          id_of_left_inverse has_lr_inv.1
        ... = (F.map (inv ⊚ φ)) ∘ 𝔸.α ∘ inv                : rfl
        ... = ((F.map inv) ⊚ (F.map φ)) ∘ 𝔸.α ∘ inv        : by rw
                           ← functor.map_comp
        ... = (F.map inv) ∘ ((F.map φ) ∘ 𝔸.α) ∘ inv        : rfl
        ... = (F.map inv) ∘ (𝔹.α ∘ φ) ∘ inv                : by rw hom
        ... = (F.map inv) ∘ 𝔹.α ∘ (φ ∘ inv)                : rfl
        ... = ((F.map inv) ∘ 𝔹.α ∘ id)                     : by rw
                          id_of_right_inverse has_lr_inv.2
        ... = (F.map inv) ∘ 𝔹.α                            : rfl
    end
```

```
lemma surj_to_hom
            (f : 𝔸.carrier ⟶ 𝔹.carrier)
            (g : 𝔹.carrier ⟶ ℂ.carrier)
            (hom_gf : is_coalgebra_homomorphism (g ∘ f))
            (hom_f : is_coalgebra_homomorphism f)
            (ep : epi f)
                : is_coalgebra_homomorphism g :=

    have  h1 : (ℂ.α ∘ g) ⊚ f = (F.map g ∘ 𝔹.α) ⊚ f :=
    calc
    (ℂ.α ∘ g) ⊚ f = F.map (g ⊚ f) ∘ 𝔸.α   : hom_gf
    ... = (F.map g ⊚ F.map f) ∘ 𝔸.α       : by rw functor.map_comp
    ... = F.map g ∘ F.map f ∘ 𝔸.α         : by tidy
    ... = F.map g ∘ 𝔹.α ∘ f               : by rw [eq.symm hom_f],
    right_cancel f h1
```

```
lemma inj_to_hom (f : 𝔸.carrier ⟶ 𝔹.carrier)
                 (g : 𝔹.carrier ⟶ ℂ.carrier)
                 (hom_gf : is_coalgebra_homomorphism (g ∘ f))
                 (hom_g : is_coalgebra_homomorphism g)
```

```
                  (inj : injective g) : is_coalgebra_homomorphism f :=
begin
     cases classical.em (nonempty 𝔹) with n_em_𝔹 emp_𝔹,
     have  h1 : (F.map g) ⊚ (F.map f) ⊚ 𝔸.α = F.map g ⊚ (𝔹.α ⊚ f) :=
     calc
     ((F.map g) ⊚ (F.map f)) ∘ 𝔸.α
             = (F.map (g ⊚ f)) ∘ 𝔸.α  : by rw functor.map_comp
     ...     = F.map (g ∘ f) ∘ 𝔸.α     : rfl
     ...     = ℂ.α ∘ g ∘ f              : by rw [eq.symm hom_gf]
     ...     = (F.map g ∘ 𝔹.α) ∘ f     : by rw [eq.symm hom_g],

     haveI inh_𝔹 : inhabited 𝔹 := ⟨choice n_em_𝔹⟩,
      -- haveI is used to define an instance.
     haveI fg_mono : mono (F.map g) := mono_preserving_functor g inj,

     exact  left_cancel (F.map g) (eq.symm h1),

     exact empty_hom_codom f emp_𝔹
end
```

```
lemma empty_hom_codom (φ : 𝔸.carrier ⟶ 𝔹.carrier)
    (em_𝔹 : ¬ nonempty 𝔹):
    is_coalgebra_homomorphism φ :=
    begin
        dsimp at *,
        ext1,
        have ex : ∃ b : 𝔹 , true := exists.intro (φ x) trivial,
        exact absurd (nonempty_of_exists ex) em_𝔹
    end
```

```
lemma coalgebra_diagram (φ : homomorphism 𝔸 𝔹)
                        (ψ : homomorphism 𝔸 ℂ)
                        (sur : surjective φ)
            :   (∃! χ : homomorphism 𝔹 ℂ , χ ∘ φ = ψ) ↔
                (sub_kern φ ψ)
                :=
    iff.intro
    -- The "exists such χ" → "kern φ ⊆ kern ψ" direction:
    begin
        intro ex,
        cases ex with χ h1,

        exact eq.subst h1.left (kern_comp φ χ)
    end
```

```
    -- The "kern φ ⊆ kern ψ" → "exists a unique χ" direction:
    begin
        assume k,
        -- using the diagram lemma of Set-Category to
        -- prove the existance and uniqueness of such morphism
        have ex_uni : ∃ χ : 𝔹 → ℂ,
            (χ ∘ φ = ψ  ∧ ∀ χ₁, χ₁ ∘ φ = ψ → χ₁ = χ)
            := (diagram_surjective φ ψ sur).2 k,

        cases ex_uni with χ spec,
        -- χ : 𝔹 → ℂ
        -- spec : χ ∘ φ = ψ  ∧ ∀ χ₁, χ₁ ∘ φ = ψ → χ₁ = χ

        have hom_χ_φ : is_coalgebra_homomorphism (χ ∘ φ) :=
            (eq.symm spec.left) ▸ ψ.property,

        have hom_χ : is_coalgebra_homomorphism χ :=
            surj_to_hom φ χ hom_χ_φ φ.property
                    ((epi_iff_surjective φ).2 sur),

        have unique : ∀ (χ₁ : homomorphism 𝔹 ℂ),
                    χ₁ ∘ φ = ψ
                    → χ₁ = ⟨χ , hom_χ⟩ := by tidy,

        exact exists_unique.intro ⟨χ , hom_χ⟩
                                    spec.left unique
    end
```

A.2.4. Subcoalgebra

```
lemma subcoalgebra_unique_structure
            (S : set 𝔸)
            (h : openset S)
            : let α := some h in
            ∀ σ : S → F.obj S,
                    @is_coalgebra_homomorphism F
                        ⟨S , σ⟩
                        𝔸
                        (S ↪ 𝔸) →
                    σ = α
            :=
    begin
        intros α σ h0,
```

```
        let coS : SubCoalgebra S := ⟨α , (some_spec h)⟩,
        cases classical.em (nonempty S) with nonemp emp,

        haveI inh : inhabited S := nonemptyInhabited nonemp,
        have hom : @is_coalgebra_homomorphism F
                            coS 𝔸 (S ↪ 𝔸) := some_spec h,
        have h2 : (F.map (S ↪ 𝔸)) ∘ α = (F.map (S ↪ 𝔸)) ∘ σ :=
                calc (F.map (S ↪ 𝔸)) ∘ α
                        = 𝔸.α ∘ (S ↪ 𝔸)           : eq.symm hom
                    ... = (F.map (S ↪ 𝔸)) ∘ σ     : h0,
        haveI h3 : mono (F.map (S ↪ 𝔸)) :=
            mono_preserving_functor (S ↪ 𝔸) (inj_inclusion 𝔸 S),

        exact eq.symm (left_cancel (F.map (S ↪ 𝔸)) h2),

        have h2 : ∀ (f₁ f₂ : S → F.obj S), f₁ = f₂ :=
            map_from_empty S (F.obj S) (nonempty_notexists emp),
        exact h2 σ (some h)
    end
```

```
lemma map_from_empty (S : set A) (B : Type u) :
            (¬ ∃ s : S , true) →
            ∀ f₁ f₂  : S → B, f₁ = f₂ :=
            assume h f₁ f₂,
            show f₁ = f₂, from
            have h0 : ∀ s : S , false := by tidy,
            by tidy
```

A.2.5. Homomorphic Image

```
lemma surj_hom_to_coStructure
    (φ : homomorphism 𝔸 𝔹) (sur : surjective φ):
    let χ : 𝔹 → F.obj 𝔹 := λ b,
        let a := some (sur b) in
        ((F.map φ) ∘ 𝔸.α) a in
    𝔹.α = χ :=
    begin
        intro χ,
        have elements : ∀ b, 𝔹.α b= χ b :=
        begin
            intro b,
            let a := some (sur b),
            have a_b : φ a = b := some_spec (sur b),
```

```
            have χ_b : χ b = ((F.map φ) ∘ 𝔸.α) a := rfl,
            have hom_φ : 𝔹.α ∘ φ = (F.map φ) ∘ 𝔸.α := φ.property,
            have h_φ : ∀ a, 𝔹.α (φ a) = ((F.map φ) ∘ 𝔸.α) a :=
                λ a ,
                have h1 : (𝔹.α ∘ φ) a = ((F.map φ) ∘ 𝔸.α) a :=
                    by rw hom_φ,
                h1,
            have α_a : 𝔹.α b = ((F.map φ) ∘ 𝔸.α) a :=
                a_b ▹ (h_φ a),
            rw α_a,
        end,
        exact funext elements
    end
```

```
lemma empty_hom_dom (φ : 𝔸.carrier ⟶ 𝔹.carrier)
    (em_𝔸 : ¬ nonempty 𝔸):
    is_coalgebra_homomorphism φ :=
    begin
        dsimp at *,
        ext1,
        have ex : ∃ a : 𝔸 , true := exists.intro x trivial,
        exact absurd (nonempty_of_exists ex) em_𝔸
    end
```

```
theorem factorization {Q : Type u}
    (φ : homomorphism 𝔸 𝔹)
    (f : 𝔸.carrier ⟶ Q) (g : Q ⟶ 𝔹.carrier)
    (h : φ.val = g ∘ f)
    (sur : surjective f)
    (inj : injective g) :
        (∃! α_Q : Q ⟶ F.obj Q ,
            @is_coalgebra_homomorphism F 𝔸 ⟨Q , α_Q⟩ f)
    :=
begin
    cases classical.em (nonempty 𝔸) with n_em_𝔸 emp_𝔸,
    haveI inh : inhabited 𝔸 := ⟨choice n_em_𝔸⟩ ,
    let hom_φ := φ.property,
    /-
        Using the E-M-Square
        A        ⟶(f)⟶       Q    ⟶(𝔹.α ∘ g)⟶ F(B),
        A  ⟶(F f ∘ α_𝔸)⟶ F(Q)    ⟶(F g)⟶    F(B)
    -/
    have commute
        : (𝔹.α ∘ g) ∘ f = (F.map g) ∘ ((F.map f) ∘ 𝔸.α) :=
```

```
    calc 𝔹.α ∘ g ∘ f = 𝔹.α ∘ φ.val                      : by simp [h
]
              ...      = (F.map φ.val) ∘ 𝔸.α            : hom_φ
              ...      = (F.map (g ∘ f)) ∘ 𝔸.α          : by rw [h]
              ...      = (F.map (g ⊚ f)) ∘ 𝔸.α          : rfl
              ...      = ((F.map g) ⊚ (F.map f)) ∘ 𝔸.α  : by rw
functor.map_comp
              ...      = (F.map g) ∘ (F.map f) ∘ 𝔸.α    : by simp,

haveI inh_Q : inhabited Q := ⟨f (default 𝔸)⟩ ,

haveI epi_f : epi f := (epi_iff_surjective f).2 sur,
haveI mono_Fg : mono (F.map g) := mono_preserving_functor g inj,
/-
    we get the existance and the uniqueness of d
    the diagonal of the square and the coalgebra structure
-/
have em_square : _ := E_M_square
    f epi_f (𝔹.α ∘ g) ((F.map f) ∘ 𝔸.α)
        (F.map g) mono_Fg commute,

cases em_square with d spec,

have homomorphism_f : d ⊚ f = (F.map f) ⊚ 𝔸.α :=
    eq.symm spec.left.left,

have com_tri_epi := commutative_triangles_epi
    f (𝔹.α ∘ g) ((F.map f) ∘ 𝔸.α) (F.map g) commute d
     (eq.symm homomorphism_f),

have uni_f : ∀ (α_Q : Q ⟶ F.obj Q),
    @is_coalgebra_homomorphism F 𝔸 ⟨Q , α_Q⟩ f →
    α_Q = d :=
    assume α_Q hom_f,
    have com : α_Q ⊚ f = F.map f ∘ 𝔸.α := hom_f,
    com_tri_epi.1 α_Q (eq.symm com),

exact exists_unique.intro d
    homomorphism_f uni_f,

have A_Q : nonempty Q → nonempty 𝔸 :=
   λ n_Q, ⟨some (sur (choice n_Q))⟩,
have em_Q : nonempty Q → false :=
    λ n_Q, emp_𝔸 (A_Q n_Q),
```

```
    have n_Q : ¬ (nonempty Q) := em_Q,
    let α_Q : Q → F.obj Q := empty_map Q n_Q (F.obj Q),
    have hom_f : @is_coalgebra_homomorphism F 𝔸 ⟨Q , α_Q⟩ f
        := empty_hom_dom f emp_𝔸,
    exact exists_unique.intro α_Q hom_f (by tidy)
end
```

```
theorem factorization_hom {Q : Type u}
    (φ : homomorphism 𝔸 𝔹)
    (f : 𝔸.carrier ⟶ Q) (g : Q ⟶ 𝔹.carrier)
    (h : φ.val = g ∘ f)
    (sur : surjective f)
    (inj : injective g)
    [inhabited 𝔸] :
    let α_Q := some (factorization φ f g h sur inj) in
        @is_coalgebra_homomorphism F ⟨Q , α_Q⟩ 𝔹 g :=
    begin
        intros α_Q,

        have hom_φ := φ.property,

        have commute : (𝔹.α ∘ g) ∘ f = (F.map g) ∘ ((F.map f) ∘ 𝔸.α) :=
            calc 𝔹.α ∘ g ∘ f
                        = 𝔹.α ∘ φ.val                        : by simp [h]
                    ... = (F.map φ.val) ∘ 𝔸.α                : hom_φ
                    ... = (F.map (g ∘ f)) ∘ 𝔸.α              : by rw [h]
                    ... = (F.map (g ⊚ f)) ∘ 𝔸.α              : rfl
                    ... = ((F.map g) ⊚ (F.map f)) ∘ 𝔸.α      : by rw
   functor.map_comp
                    ... = (F.map g) ∘ (F.map f) ∘ 𝔸.α        : by simp,
        haveI inh_Q : inhabited Q := ⟨f (default 𝔸)⟩ ,
        haveI epi_f : epi f := (epi_iff_surjective f).2 sur,
        haveI mono_Fg : mono (F.map g) := mono_preserving_functor g inj,

        have spec := some_spec (factorization φ f g h sur inj),

        have com_tri_epi : _ := commutative_triangles_epi
            f (𝔹.α ∘ g) ((F.map f) ∘ 𝔸.α) (F.map g) commute α_Q
              (eq.symm spec.1),
        exact com_tri_epi.2
    end
```

```
lemma structure_existence
    (φ : homomorphism 𝔸 𝔹)
```

```
    [inhabited 𝔸]
    : ∃ α : (range φ) → F.obj (range φ),
        let ℝ : Coalgebra F :=⟨range φ , α⟩ in
        @is_coalgebra_homomorphism F 𝔸 ℝ
                (range_factorization φ) ∧
        @is_coalgebra_homomorphism F ℝ 𝔹
                ((range φ) ↪ 𝔹) :=
    begin
        haveI inh : inhabited (range φ) :=
            inhabited.mk
            ⟨φ (default 𝔸), mem_range_self (default 𝔸)⟩,
        have ex : _ := Factorization
                        φ
                        (range_factorization φ)
                        ((range φ) ↪ 𝔹)
                        (decompose φ)
                        ((epi_iff_surjective (range_factorization φ)).2
                            surjective_onto_range)
                        (inj_inclusion 𝔹 (range φ)),
        cases ex with α hom,
        exact exists.intro α hom.left
    end

def homomorphic_image_of_range
    (φ : homomorphism 𝔸 𝔹) [inhabited 𝔸]
        : ∃ α : (range φ) → F.obj (range φ),
        homomorphic_image 𝔸 ⟨range φ , α⟩  :=
        begin
            have ex : _ := structure_existence φ,
            cases ex with α hom,
            let coalg : Coalgebra F:= ⟨range φ , α⟩,
            have h : homomorphic_image 𝔸 coalg :=
                have x : true := trivial,
                exists.intro
                ⟨range_factorization φ ,  hom.left⟩
                surjective_onto_range,
            exact exists.intro α h
        end

noncomputable lemma range_is_subCoalgebra (φ : homomorphism 𝔸 𝔹)
    [inhabited 𝔸]
    : SubCoalgebra (range φ) :=
        have ex : _ := structure_existence φ,
        let α : (range φ) → F.obj (range φ) := some ex in
```

```
        ⟨α , (some_spec ex).right⟩
```

A.2.6. Coalgebra-Coequalizer

```
theorem coequalizer_is_homomorphism :
    let 𝔹_Θ := theta φ ψ in
    let π_Θ : 𝔹.carrier ⟶ 𝔹_Θ := coequalizer φ ψ in
    ∃! α : 𝔹_Θ → (F.obj 𝔹_Θ),
    @is_coalgebra_homomorphism F 𝔹 ⟨𝔹_Θ , α⟩ π_Θ
    :=
    begin
        intros 𝔹_Θ π_Θ,

        have hom_f : 𝔹.α ∘ φ = (F.map φ) ∘ 𝔸.α := φ.property,
        have hom_g : 𝔹.α ∘ ψ = (F.map ψ) ∘ 𝔸.α := ψ.property,
        let φ₁  : 𝔸.carrier ⟶ 𝔹.carrier := φ.val,
        let ψ₁ : 𝔸.carrier ⟶ 𝔹.carrier := ψ.val,
        have h : is_coequalizer φ₁ ψ₁ π_Θ := quot_is_coequalizer φ ψ,

        have h2 : (F.map π_Θ) ∘ 𝔹.α ∘ φ = (F.map π_Θ) ∘ 𝔹.α ∘ ψ :=
            calc (F.map π_Θ) ∘ (𝔹.α ∘ φ)
                      = (F.map π_Θ) ∘ (F.map φ) ∘ 𝔸.α          : by rw
                                                  hom_f
                  ... = ((F.map π_Θ) ⊚ (F.map φ)) ∘ 𝔸.α       : rfl
                  ... = (F.map (π_Θ ⊚ φ)) ∘ 𝔸.α               : by rw
                                         functor.map_comp
                  ... = (F.map (π_Θ ⊚ ψ)) ∘ 𝔸.α               : by tidy
                  ... = ((F.map π_Θ) ⊚ (F.map ψ)) ∘ 𝔸.α       : by rw
                                       ← functor.map_comp
                  ... = (F.map π_Θ) ∘ ((F.map ψ) ∘ 𝔸.α)        : rfl
                  ... = (F.map π_Θ) ∘ (𝔹.α ∘ ψ)                : by rw
                                                  ← hom_g,

        have h3 : _ := h.2 (F.obj 𝔹_Θ) ((F.map π_Θ) ∘ 𝔹.α) h2,

        let α : 𝔹_Θ → F.obj 𝔹_Θ := some h3,

        use α,

        exact some_spec h3

    end
```

```
theorem set_coequalizer_is_coalgebra_coequalizer :
    let 𝔹_Θ := theta φ ψ in
    let π_Θ : 𝔹.carrier ⟶ 𝔹_Θ := coequalizer φ ψ in
    let α : 𝔹_Θ → F.obj (𝔹_Θ):=
         some (coequalizer_is_homomorphism φ ψ) in
    let h_π := (some_spec (coequalizer_is_homomorphism φ ψ)).1 in
    let co_𝔹_Θ : Coalgebra F := ⟨𝔹_Θ, α⟩ in
    let π₁ : 𝔹 ⟶ co_𝔹_Θ := ⟨π_Θ, h_π⟩ in
    is_coequalizer
        φ ψ π₁ :=
    begin
        intros 𝔹_Θ π_Θ α h_π co_𝔹_Θ π₁,
        let φ₁ : 𝔸.carrier ⟶ 𝔹.carrier := φ.val,
        let ψ₁ : 𝔸.carrier ⟶ 𝔹.carrier := ψ.val,
        split,
        have h : is_coequalizer φ₁ ψ₁ π_Θ := quot_is_coequalizer φ ψ,
        have h1 : _ := h.1,
        exact eq_in_set.1 h1,
        intros ℚ q h,
        have h₁ : q.val ⊚ φ₁ = q.val ⊚ ψ₁ := eq_in_set.2 h,


        have com : ∃! χ : homomorphism ⟨𝔹_Θ , α⟩ ℚ ,
                          χ ∘ π_Θ = q.val :=
            begin
                have sub_ker : sub_kern π_Θ q :=
                      coequalizer_kern φ ψ ℚ q h₁,
                have hom_π : _ :=
                  (some_spec (coequalizer_is_homomorphism φ ψ)).1,

                have diag : _ := coalgebra_diagram
                            (⟨π_Θ, hom_π⟩: homomorphism 𝔹 ⟨𝔹_Θ , α⟩)
                            q (quot_is_surjective φ ψ),
                exact diag.2 sub_ker
            end,
        let χ : co_𝔹_Θ ⟶ ℚ := some com,
        use χ,
        have spec : χ ∘ π_Θ = q := (some_spec com).1,

        have h1 : (χ ⊚ π₁) = q := eq_in_set.1 spec,
        split,
        exact h1,
        intros χ₁ coeq,
```

```
        have coeq1 : χ₁ ⊚ π₁ = χ ⊚ π₁ := by simp[h1 , coeq],
        haveI ep : epi π_Θ := (epi_iff_surjective π_Θ).2
                                (quot_is_surjective φ ψ),
        have coeq2 : χ₁.val ⊚ π_Θ = χ.val ⊚ π_Θ :=
                     eq_in_set.2 coeq1,

        have coeq3: χ₁.val = χ.val := right_cancel π_Θ coeq2,

        exact eq_in_set.1 coeq3,
    end
```

A.2.7. Coalgebra-Sum

```
lemma inl_is_homomorphism :
    @is_coalgebra_homomorphism F 𝔸 (sum_of_coalgebras 𝔸 𝔹) inl
    := by {dsimp at *, refl}

lemma inr_is_homomorphism :
    @is_coalgebra_homomorphism F 𝔹 (sum_of_coalgebras 𝔸 𝔹) inr
    := by tidy
```

```
theorem set_sum_is_coalgebra_sum :
    let e₁ : 𝔸 → (𝔸 ⊕ 𝔹) := inl in
    let e₂ : 𝔹 → (𝔸 ⊕ 𝔹) := inr in
    let h_e₁ : 𝔸 ⟶ (𝔸 ⊞ 𝔹) := ⟨e₁ ,  inl_is_homomorphism 𝔸 𝔹⟩ in
    let h_e₂ : 𝔹 ⟶ (𝔸 ⊞ 𝔹) := ⟨e₂ ,  inr_is_homomorphism 𝔸 𝔹⟩ in
            is_sum (𝔸 ⊞ 𝔹) h_e₁ h_e₂
        :=
    begin
        intros e₁ e₂ h_e₁ h_e₂ ℚ φ₁ φ₂,

        let σ : 𝔸 ⊕ 𝔹 ⟶ ℚ :=
            some (disjoint_union_is_sum ℚ φ₁ φ₂),

        let γ := ℚ.α,
        let α :=  (𝔸 ⊞ 𝔹).α,

        let hom_inl :
            α ∘ inl = (F.map inl) ∘ 𝔸.α :=
                inl_is_homomorphism 𝔸 𝔹,
        let hom_inr :
            α ∘ inr = (F.map inr) ∘ 𝔹.α
                := inr_is_homomorphism 𝔸 𝔹,
```

```
let hom_φ₁ : γ ∘ φ₁.val =  (F.map φ₁.val) ∘ 𝔸.α := φ₁.property,
let hom_φ₂ : γ ∘ φ₂.val =  (F.map φ₂.val) ∘ 𝔹.α := φ₂.property,

have h0 : _ :=
    some_spec (disjoint_union_is_sum ℚ φ₁ φ₂),

have h1 : φ₁.val = σ ∘ inl ∧ φ₂.val = σ ∘ inr:=
    and.left h0,

have h2 : γ ∘ σ ∘ inl = (F.map σ) ∘ α ∘ inl :=
    calc
     γ ∘ (σ ∘ inl) = γ ∘ φ₁.val                         : by rw
                                        h1.1
        ...      = (F.map φ₁.val) ∘ 𝔸.α              : by rw
                                        hom_φ₁
        ...      = (F.map (σ ∘ inl)) ∘ 𝔸.α           : by rw
                                        h1.1
        ...      = (F.map (σ ⊚ inl)) ∘ 𝔸.α            : by simp
        ...   = ((F.map σ) ⊚ (F.map inl)) ∘ 𝔸.α      : by rw
                                   ← functor.map_comp
        ...      = (F.map σ) ∘ (F.map inl) ∘ 𝔸.α      : by simp
        ...      = (F.map σ) ∘ α ∘ inl                : by rw
                                      [hom_inl],
have h3 : γ ∘ σ ∘ inr = (F.map σ) ∘ α ∘ inr :=
    calc γ ∘ (σ ∘ inr) = γ ∘ φ₂.val                 : by rw h1.2
             ... = (F.map φ₂.val) ∘ 𝔹.α           : by rw hom_φ₂
             ... = (F.map (σ ∘ inr)) ∘ 𝔹.α       : by rw h1.2
             ... = (F.map (σ ⊚ inr)) ∘ 𝔹.α       : rfl
             ... = ((F.map σ) ⊚ (F.map inr)) ∘ 𝔹.α   : by rw
                                     ← functor.map_comp
             ... = (F.map σ) ∘ (F.map inr) ∘ 𝔹.α : by simp
             ...       = (F.map σ) ∘ α ∘ inr        : by rw
                                 hom_inr,
have h4 : ∀ ab : 𝔸 ⊕ 𝔹, (γ ∘ σ) ab = ((F.map σ) ∘ α) ab :=
                begin
                    intro ab,
                    induction ab,
                    case inl :
                      {
                            show (γ ∘ σ ∘ inl) ab =
                                ((F.map σ) ∘ α ∘ inl) ab,
                            by rw [h2]
                        },
```

```
                        case inr :
                          {
                                show (γ ∘ σ ∘ inr) ab =
                                    ((F.map σ) ∘ α ∘ inr) ab,
                                by rw [h3]
                            },
                    end,


        have h5 : @is_coalgebra_homomorphism F
                (𝔸 ⊞ 𝔹) ℚ σ
                  := funext h4,

        let ex_uni :
            ∃! s , φ₁.val = s ∘ inl ∧ φ₂.val = s ∘ inr :=
            disjoint_union_is_sum ℚ φ₁ φ₂,

        let ex :
            ∃ s ,
                (φ₁.val = s ∘ inl ∧ φ₂.val = s ∘ inr)
                ∧ @is_coalgebra_homomorphism F
                    (𝔸 ⊞ 𝔹) ℚ s :=
                        exists.intro σ
                        (and.intro h1 h5),
        let s := some ex,
        have spec_s : (φ₁.val = s ∘ inl ∧ φ₂.val = s ∘ inr) ∧
                    @is_coalgebra_homomorphism F (𝔸 ⊞ 𝔹) ℚ s
            := some_spec ex,
        use s,
        exact spec_s.2,
        split,
        exact ⟨eq_in_set.1 spec_s.1.1,
            eq_in_set.1 spec_s.1.2⟩,
        let s₁ := some ex_uni,
        have spec_s₁ : _ := some_spec ex_uni,
        have s₁_s : s₁ = s := eq.symm (spec_s₁.2 s spec_s.1),
        intros s₂ spec_s₂,

        have s₂_s₁ : s₂.val = s₁ :=  spec_s₁.2 s₂
            ⟨eq_in_set.2 spec_s₂.1,
            eq_in_set.2 spec_s₂.2⟩,

        have s₂_s : s₂.val = s := by simp [s₁_s, s₂_s₁],
```

```
        exact eq_in_set.1 s₂_s
    end
```

```
noncomputable theorem subcoalgebra_union_is_coalgebra
    {U₁ U₂ : set 𝔸}
    (S₁ : SubCoalgebra U₁)
    (S₂ : SubCoalgebra U₂)
    : SubCoalgebra (U₁ ∪ U₂) :=
    begin

        let S : Coalgebra F := ⟨U₁ , S₁.α⟩ ⊞ ⟨U₂ , S₂.α⟩,

        have ex :=
         set_sum_is_coalgebra_sum ⟨U₁ , S₁.α⟩ ⟨U₂ , S₂.α⟩ 𝔸
                    ⟨ (U₁↪ 𝔸), S₁.h⟩  ⟨(U₂ ↪ 𝔸) , S₂.h⟩,

        let φ : S ⟶ 𝔸 := some ex,

        have spec : (((U₁↪ 𝔸) = φ ∘ inl ∧ (U₂ ↪ 𝔸) = φ ∘ inr))
            := ⟨ eq_in_set.2 (some_spec ex).1.1,
                 eq_in_set.2 (some_spec ex).1.2 ⟩ ,

        have all : ∀ a : 𝔸 , a ∈ (range φ) ↔ a ∈ (U₁ ∪ U₂) :=
            λ a,
            iff.intro
            begin
                assume ar : a ∈ range φ,
                cases (mem_range.1 ar) with s φs_a,
                induction s,
                case inl :
                    begin
                        have h01 : (φ ∘ inl) s ∈ U₁ :=
                            spec.1 ▸ s.property,
                        have h02 : a ∈ U₁ := φs_a ▸ h01,
                        by simp [h02],
                    end,
                case inr :
                    begin
                        have h01 : (φ ∘ inr) s ∈ U₂ :=
                            spec.2 ▸ s.property,
                        have h02 : a ∈ U₂ :=  (φs_a ▸ h01),
                        by simp [h02],
                    end,
            end
```

```
        begin
            assume auu : a ∈ U₁ ∨ a ∈ U₂,
            apply or.elim auu,
            assume au1: a ∈ U₁,
            have h01 : a = (φ ∘ inl) ⟨a , au1⟩  :=
                spec.1 ▹ rfl,
            exact exists.intro (inl ⟨a , au1⟩) (eq.symm (h01)),
            assume au2: a ∈ U₂,
            have h01 : a = (φ ∘ inr) ⟨a , au2⟩  :=
                spec.2 ▹ rfl,
            exact exists.intro (inr ⟨a , au2⟩) (eq.symm h01)
        end,

    rw ←(eq_sets.1 all),
    exact range_is_subCoalgebra φ,

end
```

```
lemma empty_openset {S: set 𝔸} (emp : ¬ nonempty S): openset S :=
    begin
        let α : S → F.obj S := empty_map S emp (F.obj S),
        use α,
        exact empty_hom_dom (inclusion S) emp
    end
```

```
theorem subcoalgebra_intersection_is_coalgebra
    {U V : set 𝔸}
    (S₁ : SubCoalgebra U)
    (S₂ : SubCoalgebra V)
    [∀ x : 𝔸 , decidable (x ∈ U ∩ V)]
    [∀ x : 𝔸 , decidable (x ∈ V)]
    : openset (U ∩ V) :=
    begin
        let I : set 𝔸 := U ∩ V,
        cases classical.em (nonempty (I)) with n_emp emp,

        let w : I := choice n_emp ,

        let incV : I → V := set.inclusion (by simp),
        let incU : I → U := set.inclusion (by simp),

        let p_w :U → U ∩ V := λ u , if uUV : u.val ∈ U ∩ V
                    then ⟨u.val , uUV⟩
                    else ⟨w.val , w.property⟩,
```

```
let q_w :𝔸 → V := λ a , if aV : a ∈ V
            then ⟨a , aV⟩
            else ⟨w.val , and.right w.property⟩,

have h1 : ∀ u:U ,
    (incV ∘ p_w) u = (q_w ∘ (U ↪ 𝔸)) u :=
    assume u,
    @by_cases (u.val ∈ I)
          ((incV ∘ p_w) u = (q_w ∘ (U ↪ 𝔸)) u)
        begin
            intro uI,
            have pwu_u : p_w u = ⟨u.val , uI⟩:=
                by { simp [p_w , rfl, uI] },
            have qwu_u: (q_w) u.val =
                            (⟨u.val , uI.2⟩ : V)  :=
                by { simp [q_w, rfl, uI.2] },
            calc (incV ∘ p_w) u
                        = incV  ⟨u.val , uI⟩  : by rw ←pwu_u
                    ... = (⟨u.val , uI.2⟩: V) : rfl
                    ... = q_w u.val           : by rw ←qwu_u
        end
        begin
            intro uNI,
            have pwu_w : p_w u = w :=
             by { simp [p_w , uNI, rfl] },
            have qwu_u: q_w u.val =
                            (⟨w.val , (w.property).2⟩ : V)
             := begin
                    simp[q_w],
                    split_ifs,
                    exact absurd (and.intro u.property h) uNI,
                    exact rfl
                end,
            calc incV (p_w u)
                        = incV w                      : by rw
                                              pwu_w
                    ... = (⟨w.val , w.property.2⟩: V)  : rfl
                    ... = q_w u.val                   : by rw
                                              ←qwu_u
        end
    ,
have h11 : incV ∘ p_w = q_w ∘ (U ↪ 𝔸) :=
    funext h1,
```

```
    have h22 : q_w  ∘ (V ↪ 𝔸) = id :=
        funext (by {dsimp at *,  simp [q_w]}),

    let γ := (F.map p_w) ∘ S₁.α ∘ incU,

    have hom_eq : (F.map (I ↪ 𝔸)) ∘ γ = 𝔸.α ∘ (I ↪ 𝔸) :=
  calc
  (F.map ((V ↪ 𝔸) ⊚ incV)) ∘ (F.map p_w) ∘ S₁.α ∘ incU
     = ((F.map (V ↪ 𝔸)) ⊚ (F.map incV)) ∘ (F.map p_w) ∘ S₁.α
                                              ∘ incU
        : by rw functor.map_comp
  ... = (F.map (V ↪ 𝔸)) ∘ ((F.map incV) ⊚ (F.map p_w)) ∘ S₁.α
                                              ∘ incU
        : rfl
  ... = (F.map (V ↪ 𝔸)) ∘ (F.map (incV ⊚ p_w)) ∘ S₁.α ∘ incU
        : by rw ←functor.map_comp
  ... = (F.map (V ↪ 𝔸)) ∘ (F.map (incV ∘ p_w)) ∘ S₁.α ∘ incU
        : rfl
  ... = (F.map (V ↪ 𝔸)) ∘ (F.map (q_w ∘ (U ↪ 𝔸))) ∘ S₁.α ∘ incU
        : by rw ←h11
  ... = (F.map (V ↪ 𝔸)) ∘ (F.map (q_w ⊚ (U ↪ 𝔸))) ∘ S₁.α ∘ incU
        : rfl
  ... = (F.map (V ↪ 𝔸)) ∘ ((F.map q_w) ⊚ ((F.map (U ↪ 𝔸))))
                                          ∘ S₁.α ∘ incU
        : by rw functor.map_comp
  ... = (F.map (V ↪ 𝔸)) ∘ (F.map q_w) ∘ ((F.map (U ↪ 𝔸)) ∘ S₁.α)
                                          ∘ incU
        : rfl
  ... = (F.map (V ↪ 𝔸)) ∘ (F.map q_w) ∘ 𝔸.α ∘ (U ↪ 𝔸) ∘ incU
        : by rw  eq.symm S₁.h
  ... = (F.map (V ↪ 𝔸)) ∘ (F.map q_w) ∘ (𝔸.α ∘ (V ↪ 𝔸)) ∘ incV
        : rfl
  ... = (F.map (V ↪ 𝔸)) ∘ (F.map q_w) ∘ (F.map (V ↪ 𝔸)) ∘ S₂.α
                                          ∘ incV
        : by rw ← (eq.symm S₂.h)
  ... = (F.map (V ↪ 𝔸)) ∘ ((F.map q_w) ⊚ (F.map (V ↪ 𝔸))) ∘ S₂.α
                                          ∘ incV
        : rfl
  ... = (F.map (V ↪ 𝔸)) ∘ (F.map (q_w ⊚ (V ↪ 𝔸))) ∘ S₂.α ∘ incV
        : by rw ←functor.map_comp
  ... = (F.map (V ↪ 𝔸)) ∘ (F.map (q_w ∘ (V ↪ 𝔸))) ∘ S₂.α ∘ incV
        : rfl
  ... = (F.map (V ↪ 𝔸)) ∘ (F.map id) ∘ S₂.α ∘ incV : by rw h22
  ... = (F.map (V ↪ 𝔸)) ∘ (F.map (𝟙 V)) ∘ S₂.α ∘ incV
```

```
            : rfl
      ... = (F.map (V ↪ 𝔸)) ∘ (𝟙 (F.obj V)) ∘ S₂.α ∘ incV
            : by rw functor.map_id'
      ... = ((F.map (V ↪ 𝔸)) ∘ S₂.α) ∘ incV
            : rfl
      ... = 𝔸.α ∘ (V ↪ 𝔸) ∘ incV
            : by rw eq.symm S₂.h,

        exact exists.intro γ (eq.symm hom_eq),

        exact empty_openset emp
    end
```

A.2.8. Coalgebra-Pushout

```
theorem pushout_is_coalgebra :
    let S := 𝔹₁ ⊞ 𝔹₂  in
    let 𝔹_Θ := @theta 𝔸 S (inl ∘ φ) (inr ∘ ψ) in
    let π_Θ := @coequalizer 𝔸 S (inl ∘ φ) (inr ∘ ψ) in
    ∃! α : 𝔹_Θ → F.obj 𝔹_Θ,
    let P : Coalgebra F := ⟨𝔹_Θ, α⟩ in
    @is_coalgebra_homomorphism F 𝔹₁ P (π_Θ ∘ inl) ∧
    @is_coalgebra_homomorphism F 𝔹₂ P (π_Θ ∘ inr)  :=
    begin
        assume S 𝔹_Θ π_Θ,

        have po1 : π_Θ ∘ inl ∘ φ = π_Θ ∘ inr ∘ ψ
                   := (coequalizer_sum_is_pushout φ ψ).1,

        let p₁ : 𝔸 → S := (inl ∘ φ),
        let p₂ : 𝔸 → S := (inr ∘ ψ),

        have hom_p₁ : is_coalgebra_homomorphism p₁ :=
            @comp_is_hom F 𝔸 𝔹₁ S φ ⟨inl, inl_is_homomorphism 𝔹₁ 𝔹₂⟩,

        have hom_p₂ : is_coalgebra_homomorphism p₂ :=
            @comp_is_hom F 𝔸 𝔹₂ S ψ ⟨inr, inr_is_homomorphism 𝔹₁ 𝔹₂⟩,

        have co : _ := coequalizer_is_homomorphism
                        ⟨p₁, hom_p₁⟩ ⟨p₂,  hom_p₂⟩,
        let α := some co,
        have hom_π : @is_coalgebra_homomorphism F S ⟨𝔹_Θ , α⟩ π_Θ
```

```
              := (some_spec co).1,
    let P : Coalgebra F := ⟨𝔹_Θ, α⟩,

    have hom_π_inl : α ∘ (π_Θ ∘ inl) = (F.map (π_Θ ∘ inl)) ∘ 𝔹₁.α
                  := @comp_is_hom F 𝔹₁ S P
                      ⟨inl, inl_is_homomorphism 𝔹₁ 𝔹₂⟩
                      ⟨π_Θ, hom_π⟩,

    have hom_π_inr : α ∘ (π_Θ ∘ inr) = (F.map (π_Θ ∘ inr)) ∘ 𝔹₂.α
                  := @comp_is_hom F 𝔹₂ S P
                      ⟨inr, inr_is_homomorphism 𝔹₁ 𝔹₂⟩
                      ⟨π_Θ, hom_π⟩,

    use α,
    split,
    exact ⟨hom_π_inl , hom_π_inr⟩ ,

    intros α₁ hom,

    have hom1 : α₁ ∘ (π_Θ ∘ inl) = (F.map (π_Θ ∘ inl)) ∘ 𝔹₁.α
            := hom.1,

    have hom2 : α₁ ∘ (π_Θ ∘ inr) = (F.map (π_Θ ∘ inr)) ∘ 𝔹₂.α
            := hom.2,

    have α_α₁_l : α₁ ∘ π_Θ ∘ inl = α ∘ π_Θ ∘ inl :=
            by simp [hom_π_inl, hom1],

    have α_α₁_r : α₁ ∘ π_Θ ∘ inr = α ∘ π_Θ ∘ inr :=
            by simp [hom_π_inr, hom2],

    have α_α₁_π : α₁ ∘ π_Θ = α ∘ π_Θ :=
        jointly_epi (α ∘ π_Θ) (α₁ ∘ π_Θ) α_α₁_l α_α₁_r,

    let mor_π : (𝔹₁ ⊕ 𝔹₂) ⟶ 𝔹_Θ := π_Θ,
    haveI ep : epi mor_π :=
        (epi_iff_surjective mor_π).2
        (quot_is_surjective (inl ∘ φ) (inr ∘ ψ)),

    exact right_cancel mor_π α_α₁_π,
end
```

A.2.9. Coalgebra-Equalizer

```
theorem largest_subcoalgebra_equalizer
    {C : set (equalizer_set φ ψ)}
    (lar : is_largest_coalgebra C):
    let E := equalizer_set φ ψ in
    let e : E → 𝔸 := (E ↪ 𝔸) in
    let ℂ : Coalgebra F := ⟨C , some lar.1⟩  in
    let σ : ℂ ⟶ 𝔸 := ⟨(e ∘ (C ↪ E))  , some_spec lar.1⟩ in
    is_equalizer φ ψ σ :=
    begin
        intros E e ℂ σ,

        split,

        exact eq_in_set.1
            (funext (λ c, ((C ↪ E) c).property)),

        intros Q q φq_ψq,

        have is_eq := (eqaulizer_set_is_equalizer φ ψ).2 q
                    (eq_in_set.2 φq_ψq),

        let f : Q → E := some is_eq,
        have eq_f := some_spec is_eq,

        have fact : q.val = e ∘ f := eq_f.1,

        let f₁ : Q → (range f) := range_factorization f,

        let e₁ : range f → 𝔸 := e ∘ (range f ↪ E),

        have inj_e₁ : injective e₁ :=
            begin
                intros a₁ a₂ k,
                have inj_e : ∀ r₁ r₂, e r₁ = e r₂ → r₁ = r₂ :=
                    inj_inclusion 𝔸 E,
                have ra : (range f ↪ E) a₁ = (range f ↪ E) a₂ :=
                    inj_e a₁ a₂ k,
                have inj_r : ∀ q₁ q₂,
                    (range f ↪ E) q₁ = (range f ↪ E) q₂ → q₁ = q₂ :=
                        inj_inclusion E (range f),
                exact (inj_r) a₁ a₂ ra,
            end,
```

```
    have inj_σ : injective σ :=
        begin
            intros a₁ a₂ k,
            have inj_e : ∀ r₁ r₂, e r₁ = e r₂ → r₁ = r₂ :=
                inj_inclusion 𝔸 E,
            have ra : (C ↪ E) a₁ = (C ↪ E) a₂ :=
                inj_e a₁ a₂ k,
            have inj_r :
              ∀ q₁ q₂, (C ↪ E) q₁ = (C ↪ E) q₂ → q₁ = q₂ :=
                inj_inclusion E C,
            exact inj_r a₁ a₂ ra,
        end,

    have ex := Factorization q f₁ e₁ fact
        ((epi_iff_surjective f₁).2 surjective_onto_range) inj_e₁,

    let α_ℝ : range f → F.obj (range f) := some ex,

    have Rf_C : range f ⊆ C :=
        lar.2 (range f) (exists.intro α_ℝ (some_spec ex).1.2),

    let fc : Q.carrier → ℂ.carrier
        := λ q₁ , ⟨f q₁, Rf_C (f₁ q₁).property ⟩,

    have f_fc : ∀ q₁, f q₁ = fc q₁ := λ q₁, rfl,

    have q_fc_σ_el : ∀ q₁ , q.val q₁ = (σ ∘ fc) q₁ :=
        λ q₁,
        have s0 : (σ ∘ fc) q₁ = (e ∘ f) q₁ := rfl,
        by rw [fact, s0],

    have q_fc_σ : q.val = σ ∘ fc := funext q_fc_σ_el,

    have hom_σ_fc : @is_coalgebra_homomorphism F Q 𝔸 (σ ∘ fc) :=
        q_fc_σ ▹ q.property,

    have hom_fc : @is_coalgebra_homomorphism F Q ℂ fc :=
        inj_to_hom fc σ hom_σ_fc σ.property inj_σ,

    let h_fc : Q ⟶ ℂ := ⟨fc , hom_fc ⟩,

    use h_fc,
    have s1 : q.val = (σ ⊚ h_fc).val := q_fc_σ,
```

```
        split,
        exact eq_in_set.1 s1,

        intros g q_σ_g,

        let σ₁ : ℂ.carrier ⟶ 𝔸.carrier := σ.val,

        have inj_σ₁ : injective σ₁ := inj_σ,

        haveI mo : mono σ₁ := (mono_iff_injective σ).2 inj_σ₁,

        have s2 : (σ ⊚ g).val = (σ ⊚ h_fc).val :=
            q_σ_g ▹ s1,

        have s3 : σ.val ∘ g.val = σ.val ∘ h_fc.val :=
            s2,

        have s4 : g.val = h_fc.val :=  left_cancel σ₁ s3,

        exact eq_in_set.1 s4

    end
```

A.2.10. Bisimulation

```
theorem homomorphism_iff_bisimulation (f : 𝔸 → 𝔹):
    is_coalgebra_homomorphism f ↔ is_bisimulation (map_to_graph f)
    :=
    let R : set (𝔸 × 𝔹) := map_to_graph f in
    let π₁ : R → 𝔸 := graph_fst R in
    let π₂ : R → 𝔹 := graph_snd R in
    begin
        have bij : bijective π₁ := bij_map_to_graph_fst f,

        let inv : 𝔸 → R := invrs π₁ bij,
        have elements : ∀ a , (π₂ ∘ inv) a = f a :=
            begin
                intro a,
                have inv_a : inv a = ⟨⟨a, f a⟩ , by tidy⟩ :=
                    bij.1 (by simp [invrs_id π₁ bij] : (π₁ ∘ inv) a = a),
                have π₂_r : π₂ ⟨⟨a, f a⟩ , by tidy⟩ = f a := rfl,
                simp[inv_a , π₂_r]
            end,
```

```
      have f_π₂_inv : (π₂ ∘ inv) = f:= funext elements,
      split,
      intro hom,
      let ρ := (F.map inv) ∘ 𝔸.α ∘ π₁,
      use ρ,
      split,

      calc 𝔸.α ∘ π₁ = (𝟙 (F.obj 𝔸)) ∘ 𝔸.α ∘ π₁              : rfl
           ... = (F.map (𝟙 𝔸)) ∘ 𝔸.α ∘ π₁             : by rw ← (
  functor.map_id' F 𝔸)
           ... = (F.map (π₁ ⊚ inv)) ∘ 𝔸.α ∘ π₁        : by simp [
  invrs_id]
           ... = ((F.map π₁) ⊚ (F.map inv)) ∘ 𝔸.α ∘ π₁ : by rw functor
  .map_comp,

      calc 𝔹.α ∘ (π₂)  = ((𝔹.α) ∘ f) ∘ π₁                : by tidy
              ...     = ((F.map f) ∘ 𝔸.α) ∘ π₁          : by rw
                                    ← (eq.symm hom)
              ...     = (F.map (π₂ ∘ inv)) ∘ 𝔸.α ∘ π₁ : by rw
                                   ← f_π₂_inv
              ...     = (F.map (π₂ ⊚ inv)) ∘ 𝔸.α ∘ π₁ : rfl
              ...     = ((F.map π₂) ⊚ (F.map inv)) ∘ 𝔸.α ∘ π₁ :
                             by by rw functor.map_comp,

      intro bis,
      let ρ := some bis,
      let ℝ : Coalgebra F :=  ⟨R , ρ⟩,

      have hom_π := some_spec bis,
      let h_π₁ : ℝ ⟶ 𝔸 := ⟨π₁ , hom_π.1⟩,

      have hom_π₂_inv : is_coalgebra_homomorphism (π₂ ∘ inv) :=
          @comp_is_hom F 𝔸 ℝ 𝔹
          ⟨inv, bij_inverse_of_hom_is_hom h_π₁ bij⟩ ⟨π₂ , hom_π.2⟩,

      rw ← f_π₂_inv,

      exact hom_π₂_inv
  end
```

```
theorem shape_of_bisimulation :
    (Π (P : Coalgebra F) (φ₁ : P ⟶ 𝔸) (φ₂ : P ⟶ 𝔹),
        let R : set (𝔸 × 𝔹) :=
            λ ab : 𝔸 × 𝔹 , ∃ p ,φ₁ p = ab.1 ∧ φ₂ p = ab.2 in
```

```
        is_bisimulation R)
    :=
    begin
        intros P φ₁ φ₂ R,
        let φ : P.carrier → R := λ p,
            let φ_p : 𝔸 × 𝔹 := ⟨φ₁ p,  φ₂ p⟩ in
            have φ_p_R : φ_p ∈ R := exists.intro p
                ⟨(by simp : φ₁ p = φ_p.1), (by simp : φ₂ p = φ_p.2)⟩,
            ⟨φ_p,  φ_p_R⟩,
        have sur : surjective φ := λ r, by tidy,
        let μ : R → P := surj_inv sur,
        let ρ : R → F.obj R := (F.map φ) ∘ P.α ∘ μ,
        use ρ,
        let π₁ := graph_fst R,
        let π₂ := graph_snd R,
        let hom_φ₁ : 𝔸.α ∘ φ₁ = (F.map φ₁) ∘ P.α := φ₁.property,
        let hom_φ₂ : 𝔹.α ∘ φ₂ = (F.map φ₂) ∘ P.α := φ₂.property,
        have inv_id : φ ∘ μ = @id R := funext (surj_inv_eq sur),

        have hom_π₁ : (F.map π₁) ∘ ρ = 𝔸.α ∘ π₁ :=
            calc (F.map π₁) ∘ ρ =  ((F.map π₁) ⊚ (F.map φ)) ∘ P.α ∘ μ :
  rfl
                      ... = (F.map (π₁ ⊚ φ)) ∘ P.α ∘ μ  : by rw
                                   functor.map_comp
                      ... = ((F.map φ₁) ∘ P.α) ∘ μ      : rfl
                      ... = 𝔸.α ∘ φ₁ ∘ μ                : by rw ←hom_φ₁
                      ... = 𝔸.α ∘ π₁ ∘ φ ∘ μ             : rfl
                      ... = 𝔸.α ∘ π₁ ∘ id               : by rw inv_id,

        have hom_π₂ : (F.map π₂) ∘ ρ = 𝔹.α ∘ π₂ :=
            calc (F.map π₂) ∘ ρ =  ((F.map π₂) ⊚ (F.map φ)) ∘ P.α ∘ μ
  : rfl
                      ... = (F.map (π₂ ⊚ φ)) ∘ P.α ∘ μ      : by rw
                                      functor.map_comp
                      ... = ((F.map φ₂) ∘ P.α) ∘ μ           : rfl
                      ... = 𝔹.α ∘ φ₂ ∘ μ              : by rw ← hom_φ₂
                      ... = 𝔹.α ∘ π₂ ∘ φ ∘ μ                 : rfl
                      ... = 𝔹.α ∘ π₂ ∘ id               : by rw inv_id,

        exact ⟨(eq.symm hom_π₁), (eq.symm hom_π₂)⟩
    end
```

acknowledgement

Special thanks is given to Prof. Dr. H. Peter Gumm not only for guiding this work on every step, taking the time, whenever needed, and adjusting his schedule to my chaotic requests and appointments, but also for helping me figuring out what I want to do in the future and even helping me get a leg up in this direction.