

# Formalising Displayed Categories with Univalent Foundations

Anna Williams  
Supervised by Martín Escardó

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Existing Implementations . . . . .	1
1.2	A Note On How This Paper Works . . . . .	1
1.3	Declaration On AI . . . . .	2
<b>2</b>	<b>Background</b>	<b>3</b>
2.1	Agda . . . . .	3
2.2	Univalent Foundations . . . . .	4
<b>3</b>	<b>Category Theory</b>	<b>6</b>
3.1	Categories . . . . .	6
3.2	Functors . . . . .	9
3.3	Natural Transformations . . . . .	10
3.4	Adjointns . . . . .	10
<b>4</b>	<b>Displayed Category Theory</b>	<b>12</b>
4.1	Displayed Categories . . . . .	12
4.2	Displayed Functor . . . . .	13
4.3	Total Category . . . . .	13
<b>5</b>	<b>Evaluation</b>	<b>15</b>
5.1	Further Work . . . . .	15
5.2	Acknowledgements . . . . .	16

## Abstract

I formalise both categories and displayed categories in the TypeTopology library of formalisations, with a focus on ergonomics. Due to the nature of formalisation, equalities in category theory can be hard to work with. An example is Grothendieck fibrations, in which the definition relies upon equality on objects. One solution is to work with displayed categories – categories indexed by a base category – which circumvents the issue. Since TypeTopology does not have a category theory base already implemented it is important to add this. I first implement category theory before then implementing displayed categories.

# Chapter 1

## Introduction

Category theory is a widely applicable and incredibly fruitful field of mathematics that has seen many applications in fields such as topology, programming languages and domain theory. For a library of formalisations of mathematics and computer science, it is therefore useful to implement category theory. Since category theory was not already implemented in `TypeTopology` [Ec] and I had some previous knowledge in the area [Wil25], I thought this would be good for me to implement.

I decided to also implement displayed categories following the work of Ahrens and Lumsdaine [AL19]. These are useful to work with for a two reasons: building up new categories from existing ones and thus reducing the work that needs to be done, and circumventing working with equality on objects, such as those on Grothendieck fibrations.

I first implemented an ergonomic base of category theory, in the sense that it was easy to read and work with, following the theory of the HoTT Book [Uni13]. I then added displayed categories on top of this base. With this method I worked with the base category theory when implementing displayed category theory, which helped me to improve the base theory.

### 1.1 Existing Implementations

There are many existing formalisations of category theory in different libraries. Here, I look at two Agda specific implementations and the source code from [AL19].

**agda-categories** [HC21] This implementation can be useful to study, but will not work in the library because the foundations are fundamentally different.

**unimath** [Voe+] This is implemented in a different paradigm to Agda: Rocq. Whilst the type theory and foundation is similar, it cannot be directly translated easily, but is useful to reference for proofs.

**agda-unimath** [Rij+] This is closest to what I would like to implement, but unfortunately this is not easy to read. Names are very verbose, which means that you can understand their meaning without looking at the definition, but also means that text can be quite difficult to parse the meaning of when many names are combined.

### 1.2 A Note On How This Paper Works

All of the work for this paper is in the library `TypeTopology`, as such for each formalised theorem/definition/proposition/etc in this paper, there is a link to that corresponding theorem/definition/proposition/etc in `TypeTopology`.

Simultaneously, there is a file which has a listing of the theorems/definitions/propositions/etc with numberings which correspond to the numbering found in this paper. This file can be found at:

<https://martinescardo.github.io/TypeTopology/Categories.AW-MSciProject.html>

The source code for `TypeTopology` can be found at:

<https://martinescardo.github.io/TypeTopology/>

### 1.3 Declaration On AI

Throughout this project I have actively avoided the use of any generative AI assistants for ethical considerations. All work completed is my own, or done with help from discussions with other humans. (This I want to expand).

# Chapter 2

## Background

### 2.1 Agda

For this project, I worked with Agda as this is what TypeTopology is implemented with. Agda is a functional programming language with an enriched type system similar to Haskell but with some key extensions. The most important being *dependent types*, which allow us to define much stronger types and in fact means we can define mathematical structures and statements as types. Agda, as a proof assistant, helps us to then prove these statements on structures in a ‘goal’ based approach. That is, at each step of the proof we have some end goal we wish to achieve and Agda will tell us the type of this goal.

Note that as we are working with types, we write  $X : Y$  to mean that  $X$  has type  $Y$  or that  $X$  is an element of the type  $Y$ . We will also write  $A := B$  to mean that  $A$  is definitionally equal to  $B$  and  $A \equiv B$  to mean that  $A$  is equal to  $B$ .

#### 2.1.1 The Curry-Howard Isomorphism

Note that as we are working with types, we write  $X : Y$  to mean that  $X$  has type  $Y$  or that  $X$  is an element of the type  $Y$ . We will also write  $A := B$  to mean that  $A$  is definitionally equal to  $B$  and  $A \equiv B$  to mean that  $A$  is equal to  $B$ . When working with formalisation we are inherently working with the ideas of the Curry-Howard isomorphism which characterises types as propositions. We sometimes refer to this as the propositions as types mentality. More precisely, we can interpret each type as a logical statement. Thus, when Agda gives us the type of a goal, it is telling us what we have to prove; the given type represents a proposition which we have to prove. For each type regarded as a proposition, we say that the proposition holds if we can exhibit an element of that type and such an element constitutes a proof of the proposition.

Note also, that in this viewpoint, we can read  $X : \mathcal{U}$  to mean that  $X$  is in the universe  $\mathcal{U}$ , or belongs to  $\mathcal{U}$ .

#### 2.1.2 Universes

A universe is a type whose inhabitants are themselves types. To avoid paradoxes, instead of working with one universe which contains all types (including itself), we work with a hierarchy of types, where  $\mathcal{U}_{i+1}$  contains  $\mathcal{U}_i$ , as follows:

$$\mathcal{U}_0 : \mathcal{U}_1 : \mathcal{U}_2 : \dots$$

When we have a statement such as “ $A$  is a type”, we mean that  $A : \mathcal{U}_i$  for some  $i$ . Often we leave the index off and just write  $\mathcal{U}$  as we do not care which  $i$  we choose. Throughout TypeTopology, and this paper, we will use  $\mathcal{U}, \mathcal{V}, \mathcal{W}$  and  $\mathcal{T}$  as typical characters for universes.

#### Operations on Universes

We have two operations on universes to help work with them.

The first operation we have is the least upper bound of two universes,  $\mathcal{U} \sqcup \mathcal{V}$ . For all  $\mathcal{U}, \mathcal{V}$  this is the lowest index universe such that,

$$\mathcal{U} : \mathcal{U} \sqcup \mathcal{V} \text{ and } \mathcal{V} : \mathcal{U} \sqcup \mathcal{V}.$$

The second operation is the successor  $\mathcal{U}^+$ , this gives the universe directly above the current one, that is

$$\mathcal{U}_i^+ = \mathcal{U}_{i+1}.$$

### 2.1.3 Dependent Types

A dependent type is a type of the form  $A(x)$ , where  $X : \mathcal{U}$ ,  $A : X \rightarrow \mathcal{V}$  and  $x : X$ . Notice that the type of  $A(x)$  depends on which  $x : X$  we choose. This is useful in many situations but two prominent ones are *the dependent product* and *the dependent sum*.

#### Dependent Product

The dependent product type is of the form  $\prod_{a:A} B(a)$ , for which we can equivalently write  $(a : A) \rightarrow B(a)$ . This is sometimes also called *the dependent function type*, as if  $B$  is constant (i.e.  $B(a) = B$  for all  $a : A$ ) then it simply has type  $A \rightarrow B$ . Otherwise, it acts like a function in which the type of our output depends upon the input.

In the propositions as types mentality, this emulates the “for all” statement. That is,  $B$  is some predicate which we can prove for every input  $a : A$ .

#### Dependent Sum

The dependent sum type is of the form  $\sum_{a:A} B(a)$ . If  $B$  is constant it is simply the type of pairs of  $A$  and  $B$ ,  $A \times B$ . If  $B$  is not constant, then this is a pair, where the type of the second element depends on the choice of the first. To work with this type, we have two functions  $\text{pr}_1 : \sum_{a:A} B(a) \rightarrow A$  and  $\text{pr}_2 : (s : \sum_{a:A} B(a)) \rightarrow B(\text{pr}_1 s)$ , which give the first and second elements of the pair respectively, i.e.  $\text{pr}_1(a, b) = a$  and  $\text{pr}_2(a, b) = b$ .

In the propositions as types mentality, this emulates an “exists” statement. That is,  $B$  is some predicate for which we have a ‘witness’  $a : A$ , where  $B(a)$  holds.

### 2.1.4 Identification Type

Another important type is the type of identifications, which is of the form  $a =_A b : \mathcal{U}$ , where  $A : \mathcal{U}$  and  $a, b : A$ . We have one constructor for this type, the proof of reflexivity,

$$\text{refl} : \prod_{a:A} a =_A a.$$

Through propositions as types mentality this can be treated as equality and we can read the type as the statement “for all  $a : A$ ,  $a = a$ ”.

## 2.2 Univalent Foundations

The library TypeTopology follows the univalent foundations (UF) methodology. This is a subset of formalisation in which we assume the univalence axiom and reject axiom K. Axiom K is the assumption that all instances of the identity type are constructed by  $\text{refl}$ . This is not always true and the univalence axiom gives us a different solution to classify identifications. To state what univalence is, we first have to define equivalences.

### 2.2.1 Equivalences

An equivalence is similar to the idea of a bijection or isomorphism; it is a pair of functions such that they are mutually inverse. More precisely, we have the following definition.

**Definition 2.1.** An equivalence on types  $A, B$  is given by

$$A \simeq B := \sum_{f:A \rightarrow B} \sum_{g:B \rightarrow A} (f \circ g \sim \text{id}) \times (g \circ f \sim \text{id}).$$

Where  $\sim$  is pointwise equality of functions. That is, for functions  $f, g : A \rightarrow B$

$$f \sim g := \prod_{a:A} f a = g a.$$

### 2.2.2 Univalence

The univalence axiom is the following.

**Axiom (Univalence).** Let  $A, B : \mathcal{U}$ , then  $(A = B) \simeq (A \simeq B)$ .

That is to say, there are as many identifications between types as there are equivalences between them. In fact there is an equivalence between identifications and equivalences. This axiom turns out to be very useful and allows us to prove many things which are not possible with just axiom K.

### 2.2.3 Function Extensionality

An example of a property which is provable from the univalence axiom, but does not hold under axiom K is function extensionality. This is the assumption in mathematics that if  $fa = ga$  for all  $a \in A$ , then  $f = g$ , i.e. pointwise equality implies equality. With the univalence axiom, we can show that function extensionality holds [GKL11].

### 2.2.4 Working within Univalent Foundations

#### Propositions and Sets

We often wish to characterize the size of types. To do this, we look at the relation between inhabitants of the type.

**Definition 2.2.** A type is a *proposition* if every element of the type is equal. That is,  $A$  is a proposition if for all  $a b : A$ ,  $a = b$ .

**Definition 2.3.** A type is a *set* if every identification between elements of the type is equal. That is  $A$  is a set if for all  $a b : A$ , the type  $a = b$  is a proposition.

Furthermore, a 1-type is a type whose identifications are sets, a 2-type a type whose identifications are 1-types and so on.

#### Transport and dependent equality

Often, we might want to talk about equality on objects of different types. For example, if we want to show two instances of a sum type  $p q : \sum_{a:A} P(a)$  are the same, we can show that the first projections are equivalent by exhibiting an element of  $\text{pr}_1 p = \text{pr}_1 q$ . However, we can't state the type of identifications between  $\text{pr}_2 p$  and  $\text{pr}_2 q$  since  $\text{pr}_2 p : P(\text{pr}_1 p)$  and  $\text{pr}_2 q : P(\text{pr}_1 q)$  which are not equal. For this reason we introduce the idea of transport which helps us to define the required type of identifications, by leveraging the fact that we have shown  $\text{pr}_1 p = \text{pr}_1 q$ .

**Definition 2.4.** Transport is the function with type given by

$$\text{transport} : (A : X \rightarrow \mathcal{U}) \rightarrow \{x_0 x_1 : X\} \rightarrow x_0 = x_1 \rightarrow Ax_0 \rightarrow Ax_1$$

Where  $\{x_0 x_1 : X\}$  is an implicit argument, i.e. one that Agda can figure out by itself and thus we do not have to pass ourselves.

Now, if  $e : \text{pr}_1 p = \text{pr}_1 q$ , then we can state the type of identifications between  $\text{pr}_2 p$  and  $\text{pr}_2 q$  as

$$\text{transport } P e (\text{pr}_2 p) = \text{pr}_2 q.$$

For ease of writing, we can use dependent equality which is syntactic sugar for transport.

**Definition 2.5.** A dependent equality for  $Y : X \rightarrow \mathcal{U}$ ,  $a : Y x_0$ ,  $b : Y x_1$  and  $e : x_0 = x_1$  is given by

$$a = [Y, e] b :\equiv \text{transport } Y e a = b.$$

*Note.* Often, we will just write  $a =_e b$ , where  $Y$  is clear from context.

#### Structure Identity Principle

The structure identity principle (SIP) is a way in which we define equality on mathematical structures which uses univalence.

For structures of the form  $\sum S$ , we have some form of standard notion of structure (or sns) for  $S$ , which encapsulates the property that gives it univalence. This consists of

- $\iota$  - the definition of a structure preserving map for  $\sum S$ ,
- $p$  - the proof that the identity is a structure-preserving map (for example group homomorphisms) for every pair  $(X, SX)$ , and
- $\theta$  - the proof that for  $P Q : SX$ , if  $P = Q$ , then the structure preserving map on  $(X, P)$  and  $(X, Q)$  is the map given by  $p$  on  $(X, P)$ .

With these combined, we can obtain an instance of the type  $(A = B) \simeq (A \cong B)$ , where  $A B : \sum S$ . There are also ways of extending existing structures, with a proposition (i.e.  $\infty$ -Magma to Magma, by restricting the types to sets) and with a relation (i.e. from a set to a magma, by adding the binary operation).

# Chapter 3

## Category Theory

### 3.1 Categories

We first look at wild categories, which are categories in the sense we expect usually in mathematics; we do not restrict the size of the type of objects and homomorphisms.

**Definition 3.1** (*WildCategory*). A wild category [CK17],  $W$ , consists of

- a type of objects,  $\text{obj } W$ ,
- for  $A B : \text{obj } W$ , a type of homomorphisms from  $A$  to  $B$ ,  $\text{hom } A B$ ,
- for all  $A B C : \text{obj } W$ ,  $f : \text{hom } A B$  and  $g : \text{hom } B C$ , a composition operation such that  $g \circ f : \text{hom } A C$ , and
- for each  $A : \text{obj } W$  an identity homomorphism  $\text{id}_A : \text{hom } A A$  (often written just  $\text{id}$ ),

such that

- for  $A B : \text{obj } W$  and  $f : \text{hom } A B$ ,

$$\text{id}_B \circ f = f,$$

- for  $A B : \text{obj } W$  and  $f : \text{hom } A B$ ,

$$f \circ \text{id}_A = f, \text{ and}$$

- for  $h : \text{hom } C D$ ,  $g : \text{hom } B C$  and  $f : \text{hom } A B$ ,

$$h \circ (g \circ f) = (h \circ g) \circ f.$$

**Example 3.2** (*Set*). The wild category **Set** has

- objects which are sets (explicitly  $\sum_{A:\mathcal{U}} \text{is-set } A$ )
- the homomorphisms are functions between sets
- composition is the typical function composition operation
- the identity homomorphism is the identity function.

Associativity of composition and neutrality of  $\text{id}$  is given by the properties of function composition.

**Example 3.3** (*Magma*). The wild category **Magma** has

- objects which are magmas: these are  $\sum_{A:\mathcal{U}} \text{is-set } A \times (A \rightarrow A \rightarrow A)$ , where  $A \rightarrow A \rightarrow A$  is the binary operation on the underlying type,
- homomorphisms which are structure preserving maps,  $\sum_{f:A \rightarrow B} (a b : A) \rightarrow f(a \cdot b) = (fa) * (fb)$ ,
- the identity homomorphism is the pair  $(\text{id}, \text{refl})$ , and
- composition is given by composing the underlying functions, with the proof that the composite preserves the structure, for  $f : (A, \cdot) \rightarrow (B, *)$  and  $g : (B, *) \rightarrow (C, \bullet)$

$$g(f(x \cdot y)) = g((fx) * (fy)) = (g(fx)) \bullet (g(fy)).$$

The axioms follow from the properties of composition and the fact that the underlying type of the magma is a set and so all equalities on homomorphisms are equal.

From here it is useful to consider the ways in which objects and homomorphisms relate to one another. One such relation could be equality, but a more interesting notion is isomorphism. This shows that two objects have a similar structure in a looser way than equality.

**Definition 3.4 (Isomorphism).** Let  $W$  be a wild category and  $AB : \text{obj } W$ , then an isomorphism between  $A$  and  $B$ , denoted  $A \cong B$ , consists of

- a homomorphism  $f : \text{hom } AB$ , and
- an *inverse* homomorphism  $f^{-1} : \text{hom } BA$ ,

such that

$$f \circ f^{-1} = \text{id}_b \text{ and } f^{-1} \circ f = \text{id}_a.$$

*Note.* Where  $f : A \cong B$ , we will often also write  $f$  to mean the underlying morphism, with the choice given by context. We also write  $f^{-1}$  to mean the inverse of  $f$  given by the isomorphism.

**Proposition 3.5 (at-most-one-inverse).** If  $f : A \cong B$ , then the inverse  $f^{-1} : \text{hom } BA$  is unique.

*Proof.* Let  $gh : \text{hom } BA$  be inverses of  $f$ , then

$$g = g \circ \text{id} = g \circ (f \circ h) = (g \circ f) \circ h = \text{id} \circ h = h.$$

□

We now strengthen the restrictions on the type of homomorphisms and look at precategories. They are similar to the notion of locally small categories.

**Definition 3.6 (Precategory).** A wild category,  $W$ , is a precategory, if for all  $AB : \text{obj } W$  the type  $\text{hom } AB$  is a set.

**Proposition 3.7 (being-precat-is-prop).** Let  $W$  be a wild category, then the statement that  $W$  is a precategory is a proposition.

**Example 3.8 (Set, Magma).** Both the wild category **Set** and wild category **Magma** are precategories. For the proof both of these we rely on the fact that our underlying types are sets to show that the homomorphisms are also sets.

We now look at what this restriction changes about the structure of isomorphisms.

**Proposition 3.9 (being-iso-is-prop, isomorphism-type-is-set).** Let  $P$  be a precategory,  $AB : \text{obj } P$ , and  $f : \text{hom } AB$ . Then the statement that  $f$  is an isomorphism is a proposition. Furthermore, the type of isomorphisms  $A \cong B$  is a set.

*Proof.* Since the type of homomorphisms is a set, for any two proofs of being an isomorphism, the equalities  $f \circ f^{-1} = \text{id}$  and  $f^{-1} \circ f = \text{id}$  must be equal. Combining this with Proposition 3.5, we get the desired property that each proof of isomorphism must be equal. Furthermore, since the type of homomorphisms are sets and being an isomorphism is a proposition, it follows that the dependent sum of the two must be a set. □

**Corollary 3.10 (to-≅=-).** *Two isomorphisms are equal exactly when the underlying morphisms are equal.*

*Proof.* Since being an isomorphism is a proposition, the type of isomorphisms is a subtype and thus we must only show that the first arguments are equal. That is, we must show that the underlying morphisms are equal. □

Finally, we define categories. The idea here is to further bring together the two notions of equivalence between objects. We start by defining the map id-to-iso.

**Proposition 3.11 (id-to-iso).** Let  $P$  be a precategory and  $AB : \text{obj } P$ , then there is a map  $\text{id-to-iso} : A = B \rightarrow A \cong B$ .

*Proof.* By path induction  $B \equiv A$  and thus we have to prove that  $A \cong A$ . This can be constructed easily. □

With this, we can now define a category. The idea here is to bring together the notions of equality similar to the univalence axiom, by forming an equivalence between identifications on objects and isomorphisms between them.

**Definition 3.12 (Category).** Let  $P$  be a precategory, then  $P$  is a category if for all  $A B : \text{obj } P$  the map  $\text{id-to-iso } A B$  is an equivalence.

We now show a few useful properties of categories.

**Corollary 3.13 (cat-objs-form-a-1-type).** *Let  $C$  be a category, then  $\text{obj } C$  is a 1-type.*

*Proof.* Since the type of isomorphisms is a set and, isomorphisms are equivalent to identifications, the type of identifications between objects is also a set. Thus the type of objects is a 1-type.  $\square$

**Proposition 3.14 (being-cat-is-prop).** Let  $C$  be a precategory, then the property of  $C$  being a precategory is a proposition.

*Proof.* This follows from the fact that being an equivalence is a property.  $\square$

**Example 3.15 (Set).** To show that  $\text{id-to-iso } A B$  is an equivalence for  $A B : \text{obj } \mathbf{Set}$ , we show that  $(A = B) \simeq (A \cong B)$  and that our constructed isomorphism is pointwise equal to  $\text{id-to-iso } A B$ . Following [Ive18], we observe that where  $A \equiv (a, sA)$  and  $B \equiv (b, sB)$ ,

$$\begin{aligned} A = B &\equiv (a, sA) = (b, sB) \\ &\simeq a = b && (1) \\ &\simeq a \simeq b && (2) \\ &\simeq (a, sA) \cong (b, sB) && (3) \\ &\equiv A \cong B. \end{aligned}$$

For (1) the idea follows from the fact that being a set is a property and thus the type of sets is a subtype, so  $(a, sA) = (b, sB)$  if  $a = b$ .

(2) follows from the univalence axiom.

(3) follows from the fact that the structure of equivalence is similar to isomorphisms and so we can build up an equivalence between the two with some work.

Pointwise equivalence follows easily.

**Example 3.16 (Magma).** We use SIP on Magmas to show that  $(A = B) \equiv (A \cong B)$ , similar to the method used for sets and then show pointwise equality of the underlying morphism being equal to  $\text{id-to-iso } A B$ .

### 3.1.1 Notation

Initially when working with categories, I tried to work with instance arguments. These work similar to type classes in Haskell, but unfortunately when working with two categories it often would not be able to figure out which, if any, of the categories to choose for the correct instance. Due to this, it was necessary to give additional information to perform operations, which made the code more difficult to read.

Instead I took inspiration from the existing notation in `TypeTopology`, specifically the notation for underlying type. Underlying type is defined as follows:

```
record Underlying-Type {U} {V} (X : U) (Y : V) : U  $\sqcup$  V + where
  field
  ⟨_⟩ : X → Y

open Underlying-Type {{...}} public
```

That is, it is a generic function for which we specify the input and output of when we create an instance. For example, if we want the underlying type of a set we can create the following instance.

```
instance
  underlying-set : {U} → Underlying-Type (Set U) U
  ⟨_⟩ {{underlying-set}} S = pr1 S
```

Whenever we further use  $\langle \_ \rangle$  agda then checks whether any defined instance fits the correct type and will use this definition if it does. In this way we have a polymorphic function that depending on the types can choose the correct definition to apply.

Following this, I created similar records for `obj`, `hom` and composition operations, before making a record for the category as a whole. For object specifically, I defined a function similar to underlying type, where the expected argument to the function is a wild category, precategory or category which the object comes from.

## 3.2 Functors

**Definition 3.17** (Functor). Let  $P, Q$  be precategories, then a functor  $F$  from  $P$  to  $Q$ , denoted  $F : P \rightarrow Q$ , consists of

- a map on objects  $F_0 : \text{obj } P \rightarrow \text{obj } Q$ , and
- a map on homomorphisms  $F_1 : \text{hom } A B \rightarrow \text{hom } (FA) (FB)$ ,

with the following structure

- the identity is preserved,  $F_1(\text{id}) = \text{id}$ , and
- for homomorphisms,  $f : \text{hom } A B$  and  $g : \text{hom } B C$

$$F_1(g \circ f) = F_1 g \circ F_1 f.$$

**Remark 3.18.** Often we will drop the subscript on  $F_0$  and  $F_1$  and just write  $F$  where it is clear from context what we mean.

**Example 3.19** (id-functor). Let  $P$  be a precategory, then the identity functor on  $P$ , denoted  $\text{id}_P : P \rightarrow P$ , is the functor where  $F_0 = \text{id}$  and  $F_1 = \text{id}$ . Identity preservation and the distributivity laws can be easily checked.

**Definition 3.20** ( $F \circ$ ). Let  $F : A \rightarrow B$  and  $G : B \rightarrow C$  be functors, then the functor  $G \circ F$  is given by

- $(G \circ F)_0 = G_0 \circ F_0$ , and
- $(G \circ F)_1 = G_1 \circ F_1$ .

This follows the functor laws since

$$(G \circ F)\text{id} = G(F(\text{id})) = G(\text{id}) = \text{id}$$

and

$$(G \circ F)(g \circ f) = G(F(g \circ f)) = G(Fg \circ Ff) = G(Fg) \circ G(Ff) = (G \circ F)g \circ (G \circ F)f.$$

We now look at the properties of this composition operation.

**Proposition 3.21** (id-left-neutral- $F \circ$ , id-right-neutral- $F \circ$ ). The identity functor  $\text{id}_P$  is left and right neutral with respect to functor composition. That is, for all functors  $F : A \rightarrow B$ ,

$$F \circ \text{id}_A = F \text{ and } \text{id}_B \circ F = F.$$

**Proposition 3.22** (assoc- $F \circ$ ). Functor composition is associative, that is

$$H \circ (G \circ F) = (H \circ G) \circ F.$$

### 3.2.1 Notation

Following Remark 3.18, we want to add some notation such that we can write  $F$  for either  $F_0$  or  $F_1$  and Agda will work out which we mean. To do this, we have some generic notation for a map, similar to underlying type and give an instance of this for  $F_0$  and  $F_1$  as follows.

```
instance
fobj : FUNCTORMAP (obj A) (obj B)
gen-functor-map {{fobj}} = Functor.F0 F
```

```
instance
fhom : {a b : obj A}
      → FUNCTORMAP (hom a b) (hom (functor-map a) (functor-map b))
gen-functor-map {{fhom}} = Functor.F1 F
```

Then we can create a new variable ‘functor-map’ which is defined to be gen-functor-map. When opening the functor module, we simply rename functor-map to the name we wish to use for the functor, as follows.

```
open FunctorNotation F' renaming (functor-map to F)
```

The rest of the notation functions similarly to category notation, with no novel components.

### 3.3 Natural Transformations

**Definition 3.23** (Natural Transformation). Let  $F, G : A \rightarrow B$  be functors, then a natural transformation from  $F$  to  $G$  consists of a map  $\gamma : (a : \text{obj } A) \rightarrow \text{hom } Fa \ Ga$ , such that for all homomorphisms  $f : \text{hom } a \ b$  in  $A$ ,

$$Gf \circ \gamma a = \gamma b \circ Ff.$$

*Note.* Often, for a natural transformation  $\mu : F \rightarrow G$ , we will write  $\mu a$  to denote the underlying map of  $\mu$  applied to  $a$ .

**Remark 3.24.** We can show these equations using a commutative diagram. That is,  $\gamma : (a : \text{obj } A) \rightarrow \text{hom } Fa \ Ga$  is a natural transformation between  $F$  and  $G$  if the following diagram commutes.

$$\begin{array}{ccccc}
 A & & FA & \xrightarrow{\gamma^A} & GA \\
 \downarrow f & & \downarrow Ff & & \downarrow Gf \\
 B & & FB & \xrightarrow{\gamma^B} & GB
 \end{array}$$

Figure 3.1: Diagram showing the naturality condition.

**Example 3.25** (id-natural-transformation). Let  $F : A \rightarrow B$  be a functor, then the identity natural transformation on  $F$  has map  $\gamma(a) = \text{id}$ . Checking the naturality condition, we see that

$$Ff \circ \gamma a = Ff \circ \text{id} = Ff = \text{id} \circ Ff = \gamma b \circ Ff.$$

**Definition 3.26** ( $\circ$ ). Let  $\mu : G \rightarrow H$  be a natural transformation and  $F : A \rightarrow B$  be functor. Then the composite  $\mu F : G \circ F \rightarrow H \circ F$  is the natural transformation with map  $\mu \circ F$ . This satisfies naturality since, for all  $f : \text{hom } a \ b$  we have, by naturality of  $\mu$ , that

$$(H \circ F)f \circ (\mu \circ F)a = H(Ff) \circ \mu(Fa) = \mu(Fb) \circ G(Ff) = (\mu \circ F)b \circ (G \circ F)f.$$

*Note.* We can analogously define for  $\mu : F \rightarrow G$  and  $H : C \rightarrow D$  the composite  $H\mu : H \circ F \rightarrow H \circ G$ .

**Definition 3.27** ( $\circ$ ). Let  $\mu : F \rightarrow G$  and  $\gamma : G \rightarrow H$  be natural transformations. Then the composite  $\gamma \circ \mu : F \rightarrow H$  is the natural transformation with map  $(a : \text{obj } A) \rightarrow (\gamma a) \circ (\mu a)$ .

Check naturality, for  $f : \text{hom } a \ b$ , we see that

$$\begin{aligned}
 Hf \circ (\mu a \circ \gamma a) &= (Hf \circ \mu a) \circ \gamma a \\
 &= (\mu b \circ Gf) \circ \gamma a \\
 &= \mu b \circ (Gf \circ \gamma a) \\
 &= \mu b \circ (\gamma b \circ Ff) \\
 &= (\mu b \circ \gamma b) \circ Ff.
 \end{aligned}$$

### 3.4 Adjoints

**Definition 3.28** (LeftAdjoint). Let  $F : A \rightarrow B$  be a functor, then  $F$  is left adjoint if there exists

- a functor  $G : B \rightarrow A$ ,
- a natural transformation  $\mu : \text{id}_A \rightarrow GF$  (unit), and
- a natural transformation  $\varepsilon : FG \rightarrow \text{id}_B$  (co-unit),

such that

- $\varepsilon F \circ F\eta = \text{id}_F$ , and
- $G\varepsilon \circ \eta G = \text{id}_G$ .

**Remark 3.29.** Similar to natural transformations, we can define this using diagrams too. A functor is a left adjoint if there exists  $G : B \rightarrow A$  and natural transformations  $\varepsilon, \eta$  such that.

$$\begin{array}{ccc}
 F & \xrightarrow{F\eta} & FGF \\
 \searrow \text{id}_F & & \downarrow \varepsilon F \\
 & & F
 \end{array}
 \qquad
 \begin{array}{ccc}
 GFG & \xleftarrow{\eta G} & G \\
 \downarrow G\varepsilon & & \swarrow \text{id}_G \\
 & & G
 \end{array}$$

# Chapter 4

## Displayed Category Theory

### 4.1 Displayed Categories

**Definition 4.1** (Displayed Precategory). Let  $P$  be a precategory. Then a displayed precategory,  $D$ , consists of

- for each  $A : \text{obj } P$  a type  $\text{obj}_A D$ ,
- for each  $AB : \text{obj } P$ ,  $f : \text{hom } AB$  and  $X : \text{obj}_A D$ ,  $Y : \text{obj}_B D$  a set  $\text{hom}_f X Y$ ,
- an identity morphism,  $\text{id}_X : \text{hom}_{\text{id}_A} X X$ , and
- a composition operation such that for  $f : \text{hom } AB$ ,  $g : \text{hom } BC$ ,  $\bar{f} : \text{hom } XY$  and  $\bar{g} : \text{hom } YZ$ , we have

$$\bar{g} \circ \bar{f} : \text{hom}_{g \circ f} X Z,$$

such that

- for  $f : \text{hom } AB$ ,  $X : \text{obj}_A D$ ,  $Y : \text{obj}_B D$  and  $\bar{f} : \text{hom}_f X Y$

$$\text{id}_Y \circ \bar{f} =_{(\text{id} \circ f = f)} \bar{f},$$

- for  $f : \text{hom } AB$ ,  $X : \text{obj}_A D$ ,  $Y : \text{obj}_B D$  and  $\bar{f} : \text{hom}_f X Y$

$$\bar{f} \circ \text{id}_X =_{(f \circ \text{id} = f)} \bar{f}, \text{ and}$$

- for  $f : \text{hom } AB$ ,  $g : \text{hom } BC$ ,  $h : \text{hom } CD$ ,  $X : \text{obj}_A D$ ,  $Y : \text{obj}_B D$ ,  $Z : \text{obj}_C D$ ,  $W : \text{obj}_D D$ ,  $\bar{f} : \text{hom}_f X Y$ ,  $\bar{g} : \text{hom}_g Y Z$ , and  $\bar{h} : \text{hom}_h Z W$

$$\bar{h} \circ (\bar{g} \circ \bar{f}) =_{(h \circ (g \circ f) = (h \circ g) \circ f)} (\bar{h} \circ \bar{g}) \circ \bar{f}.$$

**Example 4.2** (DispPreMagma). The displayed precategory of magmas over the precategory **Set** is given by

- for each  $A : \text{obj } \mathbf{Set}$ , the type  $A \rightarrow A \rightarrow A$  of binary operations on  $A$ ,
- for each  $f : \text{hom } AB$ ,  $\cdot : \text{obj}_A \mathbf{Magma}$  and  $*$  :  $\text{obj}_B \mathbf{Magma}$ , the type  $(x y : A) \rightarrow f(x \cdot y) = (fx) * (fy)$ ,
- the identity homomorphism is the proof that  $\text{id}$  is a magma homomorphism.
- composition is the proof that the composition of two magma homomorphisms is a magma homomorphism.

The laws for composition follow from the fact that magmas are set based.

**Definition 4.3** ( $\cong$  on  $[-]_-$ ). Let  $D$  be a displayed precategory,  $X : \text{obj}_A D$ ,  $Y : \text{obj}_B D$  and  $f : A \cong B$ . Then the displayed isomorphism between  $X$  and  $Y$ , denoted  $X \cong_f Y$ , consists of

- $\bar{f} : \text{hom}_h X Y$ , and
- $\bar{g} : \text{hom}_{f^{-1}} y x$ ,

such that

$$\bar{f} \circ \bar{g} =_{(f \circ f^{-1} = \text{id})} \text{id} \text{ and } \bar{g} \circ \bar{f} =_{(f^{-1} \circ f = \text{id})} \text{id}.$$

*Note.* Alike to regular isomorphisms, we denote by  $\bar{f}^{-1}$  the inverse homomorphism.

**Proposition 4.4** (at-most-one-D-inverse). If  $\bar{f} : X \cong Y$ , then the inverse  $\bar{f}^{-1} : \text{hom}_{f^{-1}} Y X$  is unique.

*Proof.* The proof follows similarly to the proof of Proposition 3.5, but through chaining dependent equalities.  $\square$

**Proposition 4.5** (to- $\cong$ [-]- $\Rightarrow$ ). Let  $f g : X \cong Y$ , then  $f = g$  if the underlying displayed homomorphisms are equal.

*Proof.* Following Proposition 4.4 and the fact that the homomorphisms are sets, it follows that being a displayed isomorphism is a proposition. Thus an isomorphism is a subtype and we therefore only have to prove that the underlying displayed homomorphisms are equal.  $\square$

**Proposition 4.6** (D-id-to-iso). Alike to id-to-iso for categories, we can define, for  $X : \text{obj}_A$  and  $Y : \text{obj}_B$  a map  $\text{id-to-iso-disp} : (X =_e Y) \rightarrow (X \cong_{\text{id-to-iso } A B e} Y)$  for displayed categories.

**Definition 4.7** (DisplayedCategory). Let  $P$  be a displayed precategory, then  $P$  is a displayed category if  $\text{id-to-iso-disp } e x y$  is an equivalence.

**Example 4.8** (DispCatMagma). We can clearly see the use of displayed categories shining though here. There is much less work in proving univalence, because we are using the work done in proving that **Set** is a category.

## 4.2 Displayed Functor

We define displayed functors analogously to functors, with displayed precategories indexed over a functor.

**Definition 4.9** (DisplayedFunctor). Let  $F : C \rightarrow C'$  be a functor and  $D, D'$  be displayed precategories over  $C, C'$  respectively. Then a displayed functor  $\bar{F} : D \rightarrow D'$  over  $F$  consists of

- for each  $A : \text{obj } C$ , a map  $\bar{F}_0 : \text{obj}_A D \rightarrow \text{obj}_{F A} D'$ , and
- for each  $f : \text{hom } A B$ , a map  $\bar{F}_1 : \text{hom}_f X Y \rightarrow \text{hom}_{F f} (\bar{F}_0 X) (\bar{F}_0 Y)$ .

With the expected functor laws defined analogously using dependent equalities.

## 4.3 Total Category

**Proposition 4.10** (TotalPrecategory). Let  $P$  be a precategory and  $D$  be a displayed precategory over  $P$ . Then total precategory of  $D$ , denoted  $\sum D$ , with

- objects of type  $\sum_{A:\text{obj } P} \text{obj}_A D$ ,
- homomorphisms of type  $\sum_{f:\text{hom } A B} \text{hom}_f X Y$ ,
- identity homomorphism  $(\text{id}, \text{id})$ , and
- composition operation performed pointwise

is a precategory.

*Proof.* The left and right neutrality of id and associativity of composition can be all be proved pointwise using the respective properties in  $P$  and  $D$ .

To see that the homomorphisms are sets, we observe that if the first and second elements of a dependent type are sets, then the type itself is a set. Since the homomorphisms and displayed homomorphisms are sets, it is clear that the given homomorphisms form a set.  $\square$

**Proposition 4.11** (TotalCategory). Let  $C$  be a category and  $D$  be a displayed category over  $C$ . Then the total category  $\sum D$ , formed in the same way as a total precategory is a category.

*Proof.* The proof that it is a precategory follows from Proposition 4.10, so we just need to show it is a category. For given  $(a, x) (b, y) : \text{obj}(\sum D)$  we need to show that  $\text{id-to-iso}(a, x)(b, y)$  is an equivalence. To do this, we show that  $((a, x) = (b, y)) \simeq ((a, x) \cong (b, x))$  and then that the underlying function for this equivalence is pointwise equal to  $\text{id-to-iso}(a, x)(b, y)$ .

To see that  $((a, x) = (b, y)) \simeq ((a, x) \cong (b, x))$ , we use the following chain of equivalences.

$$(a, x) = (b, y) \simeq \sum_{e:a=b} \text{transport} (\lambda z \rightarrow \text{obj}_z D) e x = y \quad (4.1)$$

$$\simeq \sum_{e:a=b} x \cong_{\text{id-to-iso } a \text{ } b e} y \quad (4.2)$$

$$\simeq \sum_{f:a \cong b} x \cong_f y \quad (4.3)$$

$$\simeq (a, x) \cong (b, y) \quad (4.4)$$

(1) follows simply from equality on dependent sum types.

(2) follows from  $\Sigma$  congruence

(3) follows from a change of variable, we can use univalence of the category  $\mathbf{C}$  to exchange  $a = b$  for  $a \cong b$ , which then carries through the change of variable to the second part

(4) follows less easily, but intuitively the equivalence can be built up.  $\square$

**Example 4.12 (Magma).** The precategory and category **Magma** can easily be constructed with the total precategory and category construction and Examples 4.2 and 4.8.

# Chapter 5

## Evaluation

Overall, I feel I have achieved the goals I set out at the start of this project. Specifically, I implemented category theory basics in `TypeTopology`, including categories, functors, natural transformations and adjoints. I also introduced notation to work with these definitions such that it was easier to understand the code. I then added examples on top of this to show other users how to use the notation and work with the defined theory.

Once I completed this, I replicated the process and defined displayed categories and displayed functors, followed by notation for displayed categories. Combining this with the already-defined category theory I implemented total categories and finally an example for displayed categories.

I found total categories the hardest to define, simply because proving univalence was a difficult process. I followed the proof given in [AL19], but this was hard to read given it was written in `Coq` and uses a different proof process to `Agda`. So I built up the parts of the proof they used and pieced them together to create my own proof.

I am happy with the notation defined, which is novel in terms of category theory and something that works really well. It allows people to read the code and understand what each statement is trying to prove.

### 5.1 Further Work

#### Notation

Further work to improve notation could be done in a few different ways to improve the experience for proving with the library and reading code from the library. The first way would be to introduce a generic operation for composition such that the same symbol can be used to compose homomorphisms, functors, displayed homomorphisms, natural transformations, etc. This would make it easier to read code as the composition symbol would always look the same and it would also make working with the library easier as there wouldn't be as many symbols to remember when writing composition.

Another way to improve notation is attempting to fix the way the types expand. Often when attempting to work out the type of an expression `Agda` will expand the notation out and make it difficult to understand what is happening without cleaning up the type manually. `Agda` has opaque definitions which might help in facilitating this and could help in stopping unfolding the types unnecessarily.

#### Examples

On top of the existing examples, it would be useful to add further examples covering functors, natural transformations, adjoints and displayed functors. This will help to show others how to work with the existing notation and definitions as well as serving as a way to test that everything is working as expected.

#### Displayed Categories

On top of the existing work done, there is much more of the original displayed categories paper which has not been implemented here. This includes fibre categories, displayed natural transformations and more definitions related to fibrations, in particular those related to Grothendieck fibrations. These would be a good way to further show the usefulness of displayed categories.

#### Category Theory

Constructions on categories would be useful to add, such as products, exponentials, pullbacks, and limits. Each of their duals could be automatically defined using the opposite category to save work. With these added it would be possible to, for example, formalise the category theory paper I wrote last year [Wil25].

## 5.2 Acknowledgements

I have many many thanks to give to my supervisor Martín Escardó for meeting with me each week throughout the year to talk about my project. The discussions and guidance he has given me has been invaluable throughout the project and have improved my enjoyment of the project thoroughly.

I would also like to thank my friends Fern, Sof and Imaan for proofreading my paper and supporting me throughout the year.

# Bibliography

- [AL19] Benedikt Ahrens and Peter LeFanu Lumsdaine. “Displayed Categories”. In: *Logical Methods in Computer Science* Volume 15, Issue 1 (Mar. 2019). ISSN: 1860-5974. DOI: 10.23638/lmcs-15(1:20)2019. URL: [http://dx.doi.org/10.23638/LMCS-15\(1:20\)2019](http://dx.doi.org/10.23638/LMCS-15(1:20)2019).
- [CK17] Paolo Capriotti and Nicolai Kraus. *Univalent Higher Categories via Complete Semi-Segal Types*. 2017. arXiv: 1707.03693 [math.CT]. URL: <https://arxiv.org/abs/1707.03693>.
- [Ec] Martín H. Escardó and contributors. *TypeTopology*. Agda development. URL: <https://github.com/martinescardo/TypeTopology>.
- [Esc22] Martín Hötzel Escardó. *Introduction to Univalent Foundations of Mathematics with Agda*. 2022. arXiv: 1911.00580 [cs.LG]. URL: <https://arxiv.org/abs/1911.00580>.
- [GKL11] Nicola Gambino, Chris Kapulkin, and Peter LeFanu Lumsdaine. 2011. URL: [https://www.math.uwo.ca/faculty/kapulkin/notes/ua\\_implies\\_fe.pdf](https://www.math.uwo.ca/faculty/kapulkin/notes/ua_implies_fe.pdf).
- [HC21] Jason Z. S. Hu and Jacques Carette. “Formalizing Category Theory in Agda”. In: *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2021. Virtual, Denmark: Association for Computing Machinery, 2021, pp. 327–342. ISBN: 9781450382991. DOI: 10.1145/3437992.3439922. URL: <https://doi.org/10.1145/3437992.3439922>.
- [Ive18] Frederik Hanghøj Iversen. *Univalent Categories A formalization of category theory in Cubical Agda*. 2018. URL: <https://publications.lib.chalmers.se/records/fulltext/256404/256404.pdf>.
- [Rij+] Egbert Rijke et al. *The agda-unimath library*. URL: <https://github.com/UniMath/agda-unimath/>.
- [Uni13] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study: <https://homotopytypetheory.org/book>, 2013.
- [Voe+] Valdimir Voevodsky et al. UniMath - a computer-checked library of univalent mathematics. URL: <https://github.com/UniMath/UniMath>.
- [Wil25] Anna Williams. *Category Theory and the  $\lambda$ -calculus*. 2025. URL: <https://anna-maths.xyz/assets/papers/rsm.pdf>.