Formalisation of Display Map Categories in Univalent Foundations

Master's Thesis

$$\begin{tikzcd}
\bullet \arrow[r, dashed] \arrow[d, dashed] \arrow[dr, phantom, "\lrcorner", very near start] & \bullet \arrow[d] \\
\bullet \arrow[r] & \bullet
\end{tikzcd}$$

Csanád Farkas

Formalisation of Display Map Categories in Univalent Foundations

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Csanád Farkas
born in Budapest, Hungary

Programming Languages Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

On the cover: the pullback of a display map.

Formalisation of Display Map Categories in Univalent Foundations

Author: Csanád Farkas
Student id: 5250986

Abstract

A display map category, originally just called a class of display maps with a stability condition, can be used to model dependent type theory. There are several other constructions on categories that can serve a similar purpose, such as comprehension categories. In fact, the similarity of such concept has been well-known, and there even have been comparisons made using bicategories of such categorical notions. In this thesis, I aim to formalise one such comparison, and implement it in a proof assistant. In order to do this, I needed to formalise display map categories, some related concepts, to then construct their bicategory, and show the comparison as a pseudofunctor into the bicategory of comprehension categories. The formalisation has been done using Univalent Foundations, while the implementation has been completed using Rocq, and more specifically the UniMath library.

Acknowledgements

I would like to thank my thesis supervisor, Benedikt Ahrens for his support and guidance throughout this project. I want to thank him for always being available to answer my questions and clear up any confusion I might have had. I would also like to thank him for keeping me on track, and not letting me balloon the scope of the project. I thank Jesper Cockx and Neil Yorke-Smith for being on my thesis committee, and would like to appologise for all of the rescheduling of the defense.

I want to thank the entire UniMath team, as they have all been helpful and quick to respond, whenever I had issues or questions about the library. I would like to especially thank Niels van der Weide, for all of their feedback on my pull requests.

Last, but certainly not least, I would like to thank all my friends and family for their continued support. I truly could not have done it without you, and for that I am deeply grateful.

Csanád Farkas
Rijswijk, July 2025

Contents

Chapter 1

Introduction

Many programming languages have a type system, of varying strength. To verify the correctness of programs written in such languages, a useful tool of analysis can be type theory, which can model type systems, especially the ones of functional languages. One can also go one level further, and model type theories themselves. A way of doing this is by modelling the semantics of type theories using category theoretical structures. For those unfamiliar with either of these fields, a sufficient explanation of both will be given, as this thesis will work with both.

This thesis aims to formalise one of the correspondences given by Ahrens, Lumsdaine and North [1]. In their paper, they give foundation-agnostic correspondences between frameworks for modelling and studying dependent type theories. This left an opportunity for the arguments to be formalised in some foundation, in our case Univalent Foundations. This choice of foundation is a bit circular, as Univalent Foundations is itself a kind of *dependent type theory*. While this is not an issue in itself, but it does make it difficult to know where to start. In this thesis, when discussing the background, we will first lay out the category theoretical concepts we use, and only then discuss type theory, both how it will be our foundation, and how it is modelled by those categorical structures.

The frameworks under study in their paper were category theoretical structures. In this thesis, we will be focusing on two of them. First is the display map category, which gives a notion of stable collection of display maps, where display maps are select morphisms of a category. The other is comprehension category, which we will not study in detail, but include as it served as the basis of the comparison for all other models. The reason for them being this central comparison, is that they are a middle ground between the models that take types to be objects in some category and ones that take types to be some display map (or morphism) instead. Further, we will follow their method of comparison by using bicategories, an extension of categories. These give a way to compare not only the models, but also their analogous concepts to functors and transformations. Both display map categories and comprehension categories have been used to model *dependent type theories*. An example of both interpretations can be found in the book from Jacobs [2, Section 10.4], where Jacobs describes also the connection between the two.

Dependent type theories model mathematics using *contexts* $\Gamma, \Delta, \ldots$, *types* $A, B, \ldots$ and *terms* $a, b, \ldots$. Often, contexts can be thought of as a list of variables, mirroring what we would call context while programming. They can be used to state *judgements* that may or may not hold, for example

Γ ctx	Γ is a context;
$\Gamma \vdash A$	A is a type in context Γ;
$\Gamma \vdash a : A$	a is a term of type A in context Γ.

These judgements are similar in concept to logical statements in classical logic, for example we can state $P \wedge \neg P$, but it will never hold. Further, we have a collection of rules that can be used to derive judgements from each other. These rules further the parallels with programming, to the extent that *dependent type theory* can be implemented as a programming language. Many such languages have been created over the years, with some examples being Rocq[1], Agda[2] and Lean[3]. These languages are often referred to as 'proof assistants', as they can be used to verify proofs. To do this, they rely on the interpretation of propositions-as-types, whereby $a : A$ is equivalent to saying "a is a proof of A". For example, Rocq has been used to create a formally verified C compiler, named CompCert.[4]

The choice to use Univalent Foundations as the foundation for this thesis was due to a number of factors. The biggest one was the existence of the UniMath library.[5] While there have been countless implementations of category theory in proof assistants[6], most of these do not implement bicategories, and the few that do are still missing the bicategory of comprehension categories. Trying to also implement those would have ballooned the scope of this thesis. In contrast, UniMath has a wide variety of definitions for constructing and working with bicategories, which will help our implementation.

For this thesis, in order to reduce confusion, I will use the word 'formalisation' to refer to the act of creating a definition, theorem or proof using the theorems and axioms of a given basis. So, this will refer to the things that we have formalised on paper. The other common meaning of formalisation, being the act of writing computer code using a proof assistant, we will refer to as 'implementation'. This division should allow our formalisation to be independent of the implementation choices made specifically for UniMath, and thus allow for other implementations.

1.1 Contributions

As a part of this thesis, I have formalised display map categories, together with their maps, in Univalent Foundations. Further, I have formalised their bicategory, as a displayed bicategory over the bicategory of univalent categories. I have given a proof that this displayed bicategory is also univalent. Finally, I have constructed a pseudofunctor from the bicategory of display map categories into the bicategory of comprehension categories.

All of these have also been implemented in the UniMath library, and have been accepted to be a part of the library. Further, the implementation includes direct maps from display map categories to comprehension categories, and their related concepts. Finally, I have completed the construction of the bicategory and pseudofunctor twice, once implemented as a direct bicategory and a pseudofunctor therefrom, and once as a displayed bicategory, and corresponding pseudofunctor. I have great confidence in that these two are in fact equivalent, though I have not explicitly shown this. The reason for my confidence is due to the fact that the two implementations were largely the same, with some minor adjustments to account for the different data layout (i. e. differently ordered pairs).

The implementation of the pseudofunctor did require some 'massaging' of the types to convince Rocqof its correctness. On the other hand, as we will see in the formalisation, the conservation laws of the pseudofunctor turn out to be quite trivial as they only involve identities, thus implementing these was also trivial. A final consideration for the implementation was its compilation time as it is part of a sizeable library, so even small

[1]The Rocq Prover - Homepage
[2]Agda Wiki
[3]Lean Language - Homepage
[4]CompCert HomePage
[5]`https://unimath.github.io/UniMath/`
[6]Discussion on category theory libraries - Rocq Forum

optimisation add up over the entirety of the library. In this thesis, I will also present some of the steps I took to optimise my proofs.

1.2 Contents

Chapter 2 will present the more advanced category theory definitions that are needed for the understanding of this thesis. These will naturally include display map categories and comprehension categories, as these are the semantic frameworks, whose comparison I aim to formalise. The chapter will also include the necessary preliminaries for these structures. Furthermore, the chapter will discuss bicategories, displayed bicategories and pseudofunctors, as these serve as the frame of the comparison.

Chapter 3 will describe the foundations used in this thesis, Univalent Foundations. As Univalent Foundations are a form of type theory, the chapter will also give a brief introduction into dependent type theory. Then, Section 3.4.1 and Section 3.4.2 will describe, respectively, how display map categories and comprehension categories can be used to model dependent type theory. Finally, the chapter will conclude with a brief discussion about the concept of *univalence*.

Chapter 4 will present my formalisation of display map categories in Univalent Foundations, together with a discussion of the details of this definition. Using this definition, it then will present the definition of the displayed bicategory of display map categories. The chapter will further show proof that this bicategory is univalent, and that its total version is equivalent to the bicategory of display map categories. Finally, to show the correspondence to comprehension categories, the chapter will describe our construction of a pseudofunctor from the displayed bicategory of display map categories into the bicategory of comprehension categories, complete with proofs that it is indeed a pseudofunctor.

Chapter 5 presents the implementation of this formalisation using the UniMath library. While the prior section will contain references to the code, where appropriate, the discussion of certain implementation details will often be omitted. Therefore, this chapter will give a better overview of what I have implemented, together with some explanations of certain implementation specific details. Furthermore, the chapter will discuss a proof assistant specific issue, that being the optimisation of proofs. This is necessary, as the UniMath library is rather large, and a change in one file can require the recompilation of many of them, so the slowness of proofs can quickly add up.

Chapter 6 presents related works in both formalisation and in semantic frameworks. The prior will focus on the formalisation of the concepts used in this thesis. The latter will present not just the frameworks presented in this paper, but some other related ones as well, some which have also been in the comparison by Ahrens, Lumsdaine and North [1].

Chapter 7 will conclude by giving an overview of what I have accomplished, together with any final thoughts I have on the project as a whole. This chapter will also include a number of topics that could be target of future work.

Chapter 2

Category Theory Preliminaries

Category theory is an area of mathematics that is concerned with describing mathematical concepts in an abstract way. By abstract, we mean here that it is not concerned with the specifics, and focuses rather on the larger picture. This abstraction can allow mathematicians to make connections between seemingly unrelated parts of mathematics. It originally started as a theory for working with topology, but turned out to have wide reaching applications, such as in functional programming and in programming languages research more broadly.

While this thesis is aimed at the general computer science audience, we will require at least a cursory knowledge of category theory, concepts such as: category; functor; and natural transformation. For those unfamiliar with these concepts, we recommend checking out either the book "Category Theory for Programming" [3, Chapters 2, 4, 5], or the website "Category Theory for Programmers" [4, Part One]. Our notation for morphisms, sometimes called arrows, will most often be $f : a \to b$, marking only the objects it is between, but in case the category it belongs to may not be apparent, we will use $f : \mathcal{C}[\![a, b]\!]$.

In this chapter, we will explain some of the more advanced concepts that will be necessary for the understanding of this thesis. These concepts will be

- display map categories and related concepts in Section 2.1;
- comprehension categories and its preliminary concepts in Section 2.2;
- bicategories and pseudofunctors in Section 2.3;
- displayed bicategories, and why they are useful in formalisation in Section 2.4.

The only difference in the presentation of these definitions will be in the case of fibrations in Section 2.2.1, where we will give their definition using displayed categories, instead of functors. As we will also explain it there, this is a logically equivalent definition, but we use this version to better align with the implementations in the UniMath library.

Arrow Category. In multiple upcoming definitions, we will reference the *arrow category* $\mathcal{C}^{\to}$, which is the category of the morphisms of $\mathcal{C}$. It is defined entirely in terms of the base category, as such, every category has exactly one corresponding arrow category. As the name implies, it is defined by taking the morphisms of a category $\mathcal{C}$ as the objects of the arrow category $\mathcal{C}^{\to}$. To give the morphisms between a pair of those arrows, we need to give a pair of morphisms from the base category.

Definition 1 (Arrow Category.)**.** Given a category $\mathcal{C}$, we can define the *arrow category* $\mathcal{C}^{\to}$. The objects of this category will be the morphisms of $\mathcal{C}$, and the morphisms between two objects, $f : x \to y$ and $f' : x' \to y'$, will be a pair of morphisms $(g, g') : (x \to x') \times (y \to y')$ in $\mathcal{C}$, such that the following diagram commutes

$$\begin{tikzcd} f \arrow[r, "(g,g')"] & f' & \text{in } \mathcal{C}^{\to} \end{tikzcd}$$

$$\begin{tikzcd} x \arrow[r, "g"] \arrow[d, "f"'] & x' \arrow[d, "f'"] \\ y \arrow[r, "g'"'] & y' \end{tikzcd} \quad \text{in } \mathcal{C}$$

The identity and composition of this category can be given by using the identity and composition of $\mathcal{C}$ respectively. For the identity, it is easy to see that $(\mathsf{id}_x, \mathsf{id}_y)$ will make the square commute, as

$$f \cdot \mathsf{id}_y = \mathsf{id}_x \cdot f.$$

As for the composition, the proof is given by combining the two squares. So, given f, f', f'', and morphisms $(g_1, g_2) : f \to f'$ and $(g_1', g_2') : f' \to f''$, we have

$$\frac{f \cdot g_2 = g_1 \cdot f' \qquad f' \cdot g_2' = g_1' \cdot f''}{f \cdot (g_2 \cdot g_2') = (g_1 \cdot g_1') \cdot f''}$$

or, as a diagram

$$\begin{tikzcd} \bullet \arrow[r, "g_1"] \arrow[d, "f"'] & \bullet \arrow[r, "g_1'"] \arrow[d, dashed, "f'"] & \bullet \arrow[d, "f''"] \\ \bullet \arrow[r, "g_2"'] & \bullet \arrow[r, "g_2'"'] & \bullet \end{tikzcd}$$

A related definition will be the codomain functor $\mathsf{cod} \colon \mathcal{C}^{\to} \to \mathcal{C}$, mapping the morphisms $f : x \to y$ to their codomain or target y. Building on this definition, one can also define arrow versions of functors $F^{\to}$ and natural transformations $\alpha^{\to}$, in a manner similar to the definitions in Section 2.1.3. Due to the similarity of the definitions, we will omit them here.

2.1 Display Map Categories

Display maps are nothing more that a selected morphism in some category. Given a collection of such display maps, we can call it a class (or set) of display maps. Early definitions, such as the one given by Hyland and Pitts [5, § 2.2], does indeed define only a class of display maps, with some added conditions. A display map category is then a base category $\mathcal{C}$, a class of display maps $\mathcal{D} \subseteq \mathsf{mor}(\mathcal{C})$, and the conditions included in earlier definitions.

In order to describe this condition imposed on the class of display maps, we will need to make use of pullbacks. A pullback is similar in concept, with the difference being the (indirect) involvement of a third object. So, where the product could be defined for two objects, the pullback will be defined for two morphisms $f : B \to A$ and $g : C \to A$. The pullback itself will be made from another object P, as well as two further morphisms $f^*\, g : P \to B$ and $g^*\, f : P \to C$. The notation $f^*\, g$ should be read as "the pullback of g along f", this terminology comes from the fact that this morphism is corresponding to g, but its target is now the domain of f.

Definition 2. Given a category $\mathcal{C}$, three objects A, B, C, two morphisms $f : B \to A$ $g : C \to A$, a *pullback* is a triple made from an object P, a morphism $f^*g : P \to B$ and a morphism $g^*f : P \to C$, such that the following commutes

$$\begin{tikzcd}
P \arrow[r, "g^* f"] \arrow[d, "f^* g"'] & C \arrow[d, "g"] \\
B \arrow[r, "f"] & A
\end{tikzcd}$$

Further, for any other such triple (Q, h, k), there must exists a unique morphism $p : Q \to P$, such that this diagram commutes

$$\begin{tikzcd}
Q \arrow[dr, dashed] \arrow[drr, bend left, "k"] \arrow[ddr, bend right, "h"'] & & \\
& P \arrow[r, "g^* f"] \arrow[d, "f^* g"'] \arrow[dr, phantom, "\lrcorner", very near start] & C \arrow[d, "g"] \\
& B \arrow[r, "f"] & A
\end{tikzcd}$$

Sometimes P is instead denoted as $B \times_A C$, or often just indicated in the square diagram using $\lrcorner$.

In this second part, we can see how similar the definition of pullbacks is to products, in certain categories they do overlap. One more things of note, which we will use in the formalisation, is the fact that any two pullbacks of the same morphisms will be isomorphic. This can be proven easily, as they must both have a unique morphism to the other, and these must compose to the identity on both sides, as the unique morphism from a pullback to itself is the identity.

Remark 3. It is important to note, that any two pullbacks $P, P' : B \times_A C$ of the same morphisms f, g will be isomorphic to each other.

Proof. Since both P and P' prime are pullbacks of f, g, we get two unique morphisms (that make the diagram below commute)

$$p_1 : P \to P' \quad \text{and} \quad p_2 : P' \to P,$$

as seen in this diagram

$$\begin{tikzcd}
& P' \arrow[r] \arrow[ddl, bend left] \arrow[dl, dashed, bend left, "p_2"] \arrow[dr, phantom, "\lrcorner", very near start] & \bullet \arrow[dd, "f"] \\
P \arrow[ur, dashed, bend left, "p_1"] \arrow[urr, bend right] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & & \\
\bullet \arrow[rr, "g"'] & & \bullet
\end{tikzcd}$$

To show that they form an isomorphism, we need to show

$$p_1 \cdot p_2 = \mathsf{id}_P \tag{2.1}$$
$$p_2 \cdot p_1 = \mathsf{id}_{P'}. \tag{2.2}$$

Both of which are trivial. Equation (2.1) follows from the fact that since P is a pullback, there is a unique arrow from P to P (such that it makes the above diagram commute), and since both sides are such an arrow, they must be equal by the uniqueness. Equation (2.2) follows a similar argument, but with P'. □

2.1.1 Display Map Category

Display map categories (DMCs) are one of the semantic frameworks that can be used to model dependent type theory in category theory. While there are multiple definitions for notions similar to display map categories, for example Hyland and Pitts describe a collection of maps together with a stability condition [5], while Taylor describes *class of displays* [6]. In the formalisation presented in this thesis, we will use the version of the definition given by Ahrens et al. [1].

Definition 4 ([5, § 2.2], [6, Definition 8.3.2.], [1, Definition 11.]). A *display map category* consists of a base category $\mathcal{C}$, and a class $\mathcal{D} \subseteq \mathsf{mor}(\mathcal{C})$ of display maps (shown as $\rightarrowtriangle$), such that, for any display map $d \in D$ and morphism $f \in \mathcal{C}$ that points to the target of d, there is some display map f^*d that is the pullback of d along f:

$$\begin{tikzcd}
\bullet \arrow[r] \arrow[d, "f^*d"'] \arrow[dr, phantom, "\lrcorner", very near start] & \bullet \arrow[d, "d"] \\
\bullet \arrow[r, "f"'] & \bullet
\end{tikzcd}$$

In this definition, we require chosen pullbacks of display maps, however, for using it as a semantic framework, definitions usually require either all pullbacks explicitly, or add a *repleteness* condition. This repleteness condition takes advantage of the fact that two pullbacks of the same two morphisms will be isomorphic, as mentioned in Remark 3.

Remark 5. The *repleteness* condition states that the class of display maps $\mathcal{D}$ shall be isomorphism invariant, i. e. any morphism that is isomorphic to a display map, shall too be a display map. In our formalisation, however, we shall forgo this condition, as taking the base category $\mathcal{C}$ to be *univalent*, in the sense that will be presented in Section 3.3, guarantees this condition. This remark will be further explained in Section 4.1, when we are formalising the notion of DMCs.

We cannot yet discuss how this structure can be used as a semantic framework for type theory, as we have yet to introduce type theory. After introducing type theory in Section 3.1, we will give a brief overview of DMCs' usage as a framework in Section 3.4.1.

Another useful way of interpreting a display map category is as a subcategory of the arrow category. This interpretation will be directly used when formalising the correspondence. The formal definition of this interpretation will be given in Section 2.1.3.

2.1.2 Maps and Natural Transformations

Similar to functors and natural transformations between categories, one can define maps and natural transformations between display map categories. The former will be defined by restricting what functors we allow by adding a property that they must have, while the latter will just be defined as a natural transformation between bases.

For a functor $F : \mathcal{C} \to \mathcal{C}'$ to be a map between two display map categories $(\mathcal{C}, \mathcal{D})$ and $(\mathcal{C}', \mathcal{D}')$, it is a natural condition that it should preserve display maps in D, i. e. it should map display maps in $\mathcal{D}$ to display maps in $\mathcal{D}'$. Further, it should also preserve the pullbacks of the display maps.

Definition 6. A *map* between two display map categories $(\mathcal{C}, \mathcal{D})$ and $(\mathcal{C}', \mathcal{D}')$ consists of a functor F, such that it preserves display maps, as well as their pullbacks. We will use the notation $F : \mathcal{D} \to \mathcal{D}'$ to show this fact.

It will be useful in our formalisation to show that this definition will include identity and composition, as those will be needed when constructing the bicategory. For the identity functor, it is trivial to show preservation, as it is just the identity, giving us the identity map for a display map category.

Lemma 7 (identity map)**.** Given a display map category $(\mathcal{C}, \mathcal{D})$, the identity functor $\mathbb{1}_{\mathcal{C}}$ is also an identity map

$$\mathbb{1}_{\mathcal{D}} : \mathcal{D} \to \mathcal{D}.$$

Proof. Since the identity functor preserves all objects and morphisms, it will preserve display maps, as well as their pullbacks. □

Showing that the composition of two functors, which both are maps, is also a map is similarly trivial. Since both functors are maps, we can use their preservation properties to show the preservation for the composition.

Lemma 8 (map composition)**.** Given three display map categories $(\mathcal{C}_1, \mathcal{D}_1)$, $(\mathcal{C}_2, \mathcal{D}_2)$, $(\mathcal{C}_3, \mathcal{D}_3)$, and two maps $F : \mathcal{D}_1 \to \mathcal{D}_2, G : \mathcal{D}_2 \to \mathcal{D}_3$, the composition of the underlying functors is also a map

$$F \cdot G : \mathcal{D}_1 \to \mathcal{D}_3.$$

Proof. Since F and G are both maps, both of their underlying functors preserve display maps and their pullbacks. It then naturally follows that the composition of the underlying functors shall also preserve maps and their pullbacks. □

These two lemmas will be useful when organising DMCs into a bicategory (see Definition 24), as we will need identity and composition of maps. The proofs of these can also be found in the implementation, named `display_map_class_functor_identity` and `display_map_class_functor_composite` respectively.

As mentioned previously, we have no requirements for transformations between maps, and can just use natural transformations directly.

Definition 9. A *transformation* between two maps $F, G : (\mathcal{C}, \mathcal{D}) \to (\mathcal{C}', \mathcal{D}')$ is simply a natural transformation between the underlying functors $\alpha : F \Rightarrow G$.

2.1.3 As a Subcategory of the Arrow Category

As the class of display maps $\mathcal{D}$ underlying a display map category is just a subset of the set of morphisms $\mathsf{mor}(\mathcal{C})$, and since the arrow category $\mathcal{C}^{\rightarrow}$ is a categorical version of the set of morphisms, we can also define a subcategory of the arrow category $\mathcal{C}^{\rightarrow}$ that will only have the display maps as its objects. Further, this will also allow to define arrow versions of both maps and transformations. As mentioned prior, the structure of these definitions is very similar to the arrow category.

Definition 10. Given a display map category $(\mathcal{C}, \mathcal{D})$, we can define a subcategory $\mathcal{D}^{\rightarrowtriangle}$ of the arrow category $\mathcal{C}^{\rightarrow}$. The elements of this subcategory are the display maps included in $\mathcal{D}$, and a morphism between two of them d, d' consists of a pair of morphisms (f, g) in $\mathcal{C}$, such that the following diagram commutes

$$\begin{tikzcd}
\bullet \arrow[r, "f"] \arrow[d, "d"'] & \bullet \arrow[d, "d'"] \\
\bullet \arrow[r, "g"'] & \bullet
\end{tikzcd}$$

Identity and composition of such morphisms is given by identity and composition in the base $\mathcal{C}$ respectively.

When we have a subcategory, we can define an inclusion functor, mapping all objects and morphisms to the same objects and morphisms in the category of which this is a subcategory. This functor is effectively just the identity functor, but with a restricted domain.

Definition 11. Given the display map subcategory $\mathcal{D}^{\rightarrowtriangle}$, we can define an inclusion functor $\iota : \mathcal{D}^{\rightarrowtriangle} \to \mathcal{C}^{\to}$. It will have the following underlying functions

$$\iota_o(d) := d$$
$$\iota_m(f, g) := (f, g).$$

Furthermore, if we have a map of DMCs $F : (\mathcal{C}, \mathcal{D}) \to (\mathcal{C}', \mathcal{D}')$, we can use it to define a functor between the subcategorical versions of $\mathcal{D}$ and $\mathcal{D}'$. This definition is basically identical to the definition of an arrow functor between arrow categories.

Definition 12. Given a map F of DMCs $(\mathcal{C}, \mathcal{D}), (\mathcal{C}', \mathcal{D}')$, we can define a functor $F^{\rightarrowtriangle}$, between the corresponding display map subcategories $\mathcal{D}^{\rightarrowtriangle}, \mathcal{D}'^{\rightarrowtriangle}$. This functor $\bar{F}$ consists of the following functions

$$F_o^{\rightarrowtriangle}(d) := F_m(d)$$
$$F_m^{\rightarrowtriangle}(f, g) := (F_m(f), F_m(g))$$

All necessary properties can be shown using the properties of F as a functor.

Finally, we can also create arrow versions of natural transformations, as the are just regular natural transformations, this definition will be the same as the one for $C^{\to}$.

Definition 13. Given a natural transformation $\alpha : F \Rightarrow G$, between maps of DMCs, we can define a natural transformation $\alpha^{\rightarrowtriangle} : F^{\rightarrowtriangle} \Rightarrow G^{\rightarrowtriangle}$. We define the underlying function $\alpha_0^{\rightarrowtriangle}$ as mapping display maps $d : x \to y$, to the pair of morphisms $(\alpha_0(x), \alpha_0(y))$, as this will be the morphism between $F_o^{\rightarrowtriangle}(d)$ and $G_o^{\rightarrowtriangle}(d)$, as can be seen in this diagram

$$\begin{tikzcd}
x \arrow[d, "d"] & F_o(x) \arrow[r, "\alpha_0(x)"] \arrow[d, "F_o^{\rightarrowtriangle}(d)"'] & G_o(x) \arrow[d, "G_o^{\rightarrowtriangle}(d)"] \\
y & F_o(y) \arrow[r, "\alpha_0(y)"] & G_o(y)
\end{tikzcd}$$

The naturality condition will be given by the naturality of the transformation α.

2.2 Comprehension Categories

Comprehension category (CC), introduced by Jacobs [7, Def. 4.1.], is the central comparison for models of dependent type theory in the paper we are following [1]. They consist of two categories, a fibration and a comprehension functor, hence the name. In this section, we will start by discussing what fibrations are, then give the definition for comprehension categories, and finish by giving the pseudo maps and transformations for CCs.

2.2.1 Grothendieck Fibration.

First described by Grothendieck, fibrations (and the underlying cartesian arrows) encode a notion of a category 'lying over' another category, in some well behaved way. Given an object A in the base category $\mathcal{C}$, the fibration will give us a fibre of objects $\bar{A}, \ldots : \mathcal{D}_A$ over that object. Further, for any morphism $f : A \to B$ in the base, we will have a corresponding 'lifted' morphism $\bar{f} : \bar{A} \to \bar{B}$ in the above category. A related concept is displayed categories, introduced by Ahrens and Lumsdaine [8]. They are defined, given some category, which we'll refer to as the base, by assigning a collection of objects to every object in that category, and assigning a set of morphisms to every morphism in the base. Naturally, a set of suitable laws is required. Displayed categories are more relaxed, as such they correspond more to functors, but Ahrens and Lumsdaine also gave definitions of when a displayed category is a fibration. In this section, we will present their definitions, as in the implementation, it will be this version that will be used.

Definition 14 ([8, Definition 3.1])**.** Given a category $\mathcal{C}$, a *displayed category* $\mathcal{D}$ over $\mathcal{C}$ consists of

1. for every object x in $\mathcal{C}$, a collection of objects $\bar{x} : \mathcal{D}_x$;
2. for every morphism $f : x \to y$ in $\mathcal{C}$, a set of morphisms $\bar{f} : \bar{x} \underset{f}{\to} \bar{y}$;
3. for every object $\bar{x} : \mathcal{D}_x$, a morphisms $\mathsf{id}(\bar{x}) : \bar{x} \xrightarrow[\mathsf{id}(x)]{} \bar{x}$;
4. for every pair of morphisms $\bar{f} : \bar{x} \underset{f}{\to} \bar{y}$ and $\bar{g} : \bar{y} \underset{g}{\to} \bar{z}$, their composition $\bar{f} \cdot \bar{g} : \bar{x} \xrightarrow[f \cdot g]{} \bar{y}$.

Such that the following *dependent* equalities hold

$$\bar{f} \cdot \mathsf{id}(\bar{y}) =_* \bar{f} \tag{2.3}$$
$$\mathsf{id}(\bar{x}) \cdot \bar{f} =_* \bar{f} \tag{2.4}$$
$$\bar{f} \cdot (\bar{g} \cdot \bar{h}) =_* (\bar{f} \cdot \bar{g}) \cdot \bar{h} \tag{2.5}$$

In this definition, the reason for the dependent equalities is the difference in types on the two sides, when formalising this in Univalent Foundations. For example $\bar{f} \cdot \mathsf{id}$ will be of a type $\bar{x} \to_{f \cdot \mathsf{id}} \bar{y}$, while $\bar{f}$ will have a type $\bar{x} \to_f \bar{y}$. The equality of these types will be *dependent* on the corresponding identity law of $\mathcal{C}$. This dependence will be hidden in the remaining definitions in this section.

A simple example of a displayed category could be the displayed category of monoids Mon over the category of sets Set. The collection of objects over a given set will correspond to all possible monoids made from its elements, and the collection above a morphism (function) will have a single element over functions that preserve the monoid structure, and will be empty otherwise.

There is a natural connection between the notion of displayed categories, and functors. Given a displayed category, one can define a functor $F : \mathcal{D} \to \mathcal{C}$, that maps all objects and morphisms in $\mathcal{D}$ to their base in $\mathcal{C}$. Furthermore, given a functor $F : \mathcal{E} \to \mathcal{C}$, one can define a corresponding displayed category over $\mathcal{C}$.

Example 15. Given a functor $F : \mathcal{E} \to \mathcal{C}$, we can define a displayed category $\mathcal{D}$ over $\mathcal{C}$. This will consist of

- the collection of objects over $c : \mathcal{C}$, will be all objects $x : \mathcal{E}$, such that $F_o(x) = c$;
- the collection of morphisms over $f : x \to y$, will be all morphisms $g : \bar{x} \to \bar{y}$, such that $F_m(g) = f$;
- for the identities, we can take the identities of $\mathcal{E}$, since F preserves them ($F(\mathsf{id}) = \mathsf{id}$);
- similarly, the composition will just be the composition in $\mathcal{E}$, since F also preserves composition.

The laws can be shown to hold, using the laws of $\mathcal{E}$.

Doing this the other direction is also possible, and given a displayed category $\mathcal{D}$, there is always a corresponding functor $\pi_1^{\mathcal{D}} : \int \mathcal{D} \to \mathcal{C}$, where $\int \mathcal{D}$ is the total version of $\mathcal{D}$. The total version is created by pairing up objects and morphisms in $\mathcal{D}$ with their base, for example $(x, \bar{x})$ would be an object, where $\bar{x}$ is over x. The functor $\pi_1^{\mathcal{D}}$ simply maps these pairs to their first element.

This is the correspondence we mentioned at the top of this section, and using it, we can use displayed categories rather than functors to define fibrations. To do this, first, we need to present the definition of cartesian arrows, as they are a prerequisite for fibrations.

Definition 16. Given a displayed category $\mathcal{D}$ over $\mathcal{C}$, a morphism $\bar{f} : \bar{y} \xrightarrow[f]{} \bar{z}$ in $\mathcal{D}$ is called *cartesian* if for any morphism $\bar{g} : \bar{x} \xrightarrow[g]{} \bar{z}$ in $\mathcal{D}$ and $w : x \to y$ in $\mathcal{C}$ such that $w \cdot f = g$, there exists a unique morphism $\bar{w} : \bar{x} \to \bar{y}$ in $\mathcal{D}$, such that $\bar{w} \cdot \bar{f} = \bar{g}$.

The corresponding diagram is the following:

$$\begin{tikzcd}
\bar{x} \arrow[dr, dashed, "\bar{w}"'] \arrow[drr, "\bar{g}"] & & & \\
& \bar{y} \arrow[r, "\bar{f}"'] & \bar{z} & \mathcal{D} \arrow[dd] \\
x \arrow[dr, "w"'] \arrow[drr, "g"] & & & \\
& y \arrow[r, "f"'] & z & \mathcal{C}
\end{tikzcd}$$

One relevant example of a cartesian morphism will be in the case of the codomain functor $\mathsf{cod} : \mathcal{C}^{\to} \to \mathcal{C}$. This functor maps elements $f : x \to y$ of the arrow category $\mathcal{C}^{\to}$ (which are just the morphisms of the base category $\mathcal{C}$) to their codomain y. For this functor, a cartesian arrow corresponds to a pullback in the base category. To see why this is, consider the following diagram.

$$\begin{tikzcd}
\bullet \arrow[dd, "\bar{z}"'] \arrow[dr, dashed, "\bar{w}"'] \arrow[drr, "\bar{g}"] & & \\
& \bullet \arrow[r, "\bar{f}"] \arrow[dd, "\bar{y}"] \arrow[dr, phantom, "\lrcorner", very near start] & \bullet \arrow[dd, "\bar{x}"] \\
x \arrow[dr, "w"'] \arrow[drr, "g"] & & \\
& y \arrow[r, "f"] & z
\end{tikzcd}$$

Here, $\bar{x}, \bar{y}$ and $\bar{z}$ are objects of $\mathcal{C}^{\to}$, $(\bar{f}, f)$ would be an arrow over f. This arrow is cartesian if the square in the diagram is a pullback. The reason for this is, that for any other arrow $(\bar{g}, g)$ in $\mathcal{C}^{\to}$, and morphism w in $\mathcal{C}$, such that $g = w \cdot f$, and object $\bar{z}$, we will have a unique morphism $\bar{w}$, given by the pullback, which will give the unique arrow $(\bar{w}, w)$ in $\mathcal{C}^{\to}$. Similarly, one can also show this implication also holds in the other direction.

A fibration will then be defined by having a cartesian arrow above every morphism in $\mathcal{C}$. This cartesian arrow will then be called the *cartesian lift* of a given morphism.

Definition 17. A displayed map category is a fibration, if for all objects $\bar{z} : \mathcal{D}_z$ and morphisms $f : y \to z$ in $\mathcal{C}$, there exists a cartesian morphism $\bar{f} : \bar{y} \xrightarrow[f]{} \bar{z}$. The functor version will be denoted as $F : \mathcal{D} \twoheadrightarrow \mathcal{C}$.

It is important to note here that the above definition only asks for the existence of the cartesian morphism above f. If we want to have a function that lifts morphisms to the cartesian morphisms, that will be called a *cleavage* and a fibration together with a cleavage will be called a *cleaving*. This may seem unnecessary, but the foundations (see Chapter 3) we will be working with do not have the *axiom of choice*, or have only a modified version, therefore, if one wants to use specific cartesian arrows above f, we will need to use a *cleaving*. In fact, in the formalisation of comprehension categories, the fibration has been replaced by a cleaving in the UniMath library.

Building on the connection between pullbacks in the base category and cartesian arrows for the cod functor, if a category has all pullbacks, then cod is a fibration. A simpler example of a fibration would be the related domain functor $\mathsf{dom} : \mathcal{C}^{\to} \to \mathcal{C}$, mapping morphisms to their domain.

Example 18. The functor $\mathsf{dom} : \mathcal{C}^{\to} \to \mathcal{C}$ is a *fibration*.

Proof. In order for it to be a fibration, it must have a cartesian morphism f, for all objects $\varphi : \mathcal{C}^{\to}$ and morphism $f_0 : y \to \mathsf{dom}(\varphi)$, such that $\mathsf{dom}(f) = f_0$. Our candidate for this cartesian morphism will be $(f_0, f_0 \cdot \varphi)$, as seen below

$$\begin{tikzcd}
y \arrow[r, "f_0"] \arrow[d, "f_0 \cdot \varphi"'] & x \arrow[d, "\varphi"] \\
x' \arrow[r, dash, "\mathsf{id}"] & x'
\end{tikzcd}$$

Then, we must show that for any $\gamma : \mathcal{C}^{\to}$, $g : \gamma \to \phi$ and $w : \mathsf{dom}(\gamma) \to y$, such that $\mathsf{dom}(g) = w \cdot f_0$, we must have a unique morphism $\bar{w}$, such that $w = \mathsf{dom}(\bar{w})$ and $g = \bar{w} \cdot f$. Let us make a new diagram for this

$$\begin{tikzcd}
z \arrow[dr, "w"] \arrow[drr, "g_1"] \arrow[dd, "\gamma"'] & & \\
& y \arrow[r, "f_0"] \arrow[dd, "f_0 \cdot \varphi"] & x \arrow[dd, "\varphi"] \\
z' \arrow[dr, dashed] \arrow[drr, "g_2"] & & \\
& x' \arrow[r, "\mathsf{id}"] & x'
\end{tikzcd}$$

From the first condition, we already know that $\bar{w}_1 = w$, and from the diagram we can see that

$$g_2 = \bar{w}_2 \cdot \mathsf{id} \qquad \textit{(using the identity rules of } \mathcal{C}\textit{)}$$
$$g_2 = \bar{w}_2$$

So, $\bar{w} = (w, g_2)$. The uniqueness of this arrow can be shown easily. □

2.2.2 Comprehension Category

We can now discuss the definition of the comprehension category, consisting of two categories C and T, together with a fibration $p : \mathcal{T} \longrightarrow \mathcal{C}^{\to}$, and a comprehension functor from $\mathcal{T}$ to $\mathcal{C}^{\to}$. For a functor to be a comprehension functor, it needs to preserve cartesian arrows. This comprehension will then mirror the definition of DMC, as it will map elements of $\mathcal{T}$ to arrows, and these arrows will have pullbacks, since cartesian arrows in $\mathcal{C}^{\to}$ correspond to pullbacks in $\mathcal{C}$.

Definition 19 ([7, Def. 4.1], comprehension_cat_structure)**.** A *comprehension category* consists of a base category $\mathcal{C}$, a fibration $p : \mathcal{T} \to \mathcal{C}^{\to}$, and a *comprehension* χ (i. e. it preserves cartesian arrows) from $\mathcal{T}$ to $\mathcal{C}^{\to}$, such that the following commutes:

$$\begin{tikzcd}
\mathcal{T} \arrow[rr, "\chi"] \arrow[dr, "p"'] & & \mathcal{C}^{\to} \arrow[dl, "\mathsf{cod}"] \\
& \mathcal{C}. &
\end{tikzcd}$$

The most simple example of a comprehension category is a base category $\mathcal{C}$ with all pullbacks, together with the fibration $\mathsf{cod} : \mathcal{C}^{\to} \to \mathcal{C}$ (it is a fibration because we have all pullbacks), and the comprehension $\mathbb{1}_{\mathcal{C}^{\to}} : \mathcal{C}^{\to} \to \mathcal{C}^{\to}$.

By restricting the allowed comprehensions, we can refine the kinds of CCs. Of particular interest to us is the full comprehension category, defined as follows.

Remark 20. A comprehension category is considered *full*, if its comprehension χ is *fully faithful*.

A functor is said to be *fully faithful* if the underlying function mapping morphisms is bijective.

Just like with the DMCs, we will only discuss the usage of CCs as a semantic framework after introducing the concepts of dependent type theory in Section 3.1.

2.2.3 Pseudo Map and Transformation

Pseudo maps fill a similar role for comprehension categories to what functors are for categories. While these do not play a role when using comprehension categories as a semantic framework, they will be nonetheless needed when formalising the correspondence with display map categories.

Since the comprehension category involves two categories, we will need two functors $F : \mathcal{C} \to \mathcal{C}'$ and $\bar{F} : \mathcal{T} \to \mathcal{T}'$ to define a pseudo map between them. Further, these two will have to map elements in a congruent manner to each other, for this reason, the definition includes an isomorphism, corresponding to the two functors agreeing up to isomorphism.

Definition 21. A *pseudo map* between two comprehension categories $(\mathcal{C}, \mathcal{T}, \chi)$ and $(\mathcal{C}', \mathcal{T}', \chi')$ consists of two functors $F : \mathcal{C} \to \mathcal{C}'$ and $\bar{F} : \mathcal{T} \to \mathcal{T}'$, as well as a natural isomorphism $\varphi : \chi \cdot F^{\to} \implies \bar{F} \cdot \chi'$.

$$\begin{tikzcd}
\mathcal{T} \arrow[r, "\chi"] \arrow[d] \arrow[drr, "\bar{F}"] & \mathcal{C}^{\to} \arrow[dl] \arrow[drr, "F^{\to}"] \arrow[d, phantom, "\overset{\varphi}{\Longrightarrow}"] & & \\
\mathcal{C} \arrow[drr, "F"'] & {} & \mathcal{T}' \arrow[r, "\chi'"] \arrow[d] & \mathcal{C}'^{\to} \arrow[dl] \\
& & \mathcal{C}' &
\end{tikzcd}$$

This definition can be further restricted to require the two functors to agree on the nose. It is then called a strict map. In the formalisation part of this thesis, we will see an example for such a map.

Remark 22. If the isomorphism is just the identity, and we have an equality

$$\chi \cdot F^{\to} = \bar{F} \cdot \chi'$$

then we call the *pseudo map* a *strict map* instead.

Finally, we can describe transformations between the pseudo maps. As a pseudo map consists of two functors, this transformation will need to transform both, requiring two natural transformations. Further, the two transformations will need to be coherent, this time up to equality.

Definition 23. A *transformation* between two pseudo maps $(F, \bar{F}, \varphi)$ and $(F', \bar{F}', \varphi')$ consists of two natural transformations $\alpha : F \Rightarrow F'$ and $\bar{\alpha} : \bar{F} \Rightarrow \bar{F}'$, such that for each $A \in \mathcal{T}$

$$\varphi'_A \; \chi'(\bar{\alpha}_A) = \alpha_{\varphi.A} \; \varphi_A.$$

That last equation might be a little hard to parse, so here's an illustration

$$\begin{tikzcd}
\mathcal{T} \arrow[rr, "\iota"] \arrow[dr, bend right, no head] \arrow[r, bend right=10, no head] & \bar{F}' \arrow[r, phantom, "= \varphi'"] \arrow[dr, bend left=10] & \mathcal{C}^{\to} \arrow[dr, bend right, no head] \arrow[r, bend left=10, no head] & F'^{\to} \arrow[dr, bend left=10] & \\
& \bar{F} \arrow[u, Rightarrow, "\bar{\alpha}"] \arrow[r, phantom, "= \varphi"] \arrow[r, bend right=10] & \mathcal{T}' \arrow[rr, "\iota'"'] & F^{\to} \arrow[u, Rightarrow, "\alpha^{\to}"'] \arrow[r, bend right=10] & \mathcal{C}'^{\to}
\end{tikzcd}$$

The equation just states that the natural transformations must commute.

2.3 Bicategory

To show the correspondence between the above concepts, we will make use of bicategories and pseudofunctors between them. This will not only show that we can create a function from one to the other, but also that said function behaves well. Bicategories are an extension of categories, adding so called 2-cells that go between morphisms, now referred to as 1-cells. Similarly, pseudofunctors are the extension of functors, now between the bicategories. Important to note, that this extension will mean that laws for 1-cells will be now given as invertible 2-cells instead, this is why a bicategory is sometimes referred to as *weak 2-categories*, whereas requiring equalities would give a strict 2-category.

Definition 24 ([9, Local definition (1.1)], [10])**.** A *bicategory* $\mathcal{B}$ consists of

- A collection of *objects* $x, y, z, ... : \mathcal{B}_0$;
- for each pair of objects x, y, a collection of *1-cells* $x \to y$;
- for each pair of objects x, y and pair of 1-cells $f, g : x \to y$, a collection of *2-cells* $f \Rightarrow g$;

and a number of operators on these cells

- for each object x, an *identity* $\mathsf{id}_1(x) : x \to x$;
- for each $f : x \to y$ and $g : y \to z$ a *composite* $f \cdot g : x \to z$;
- for each $f : x \to y$, an *identity* $\mathsf{id}_2(f) : f \Rightarrow f$;
- for each $\theta : f \Rightarrow g$ and $\gamma : g \Rightarrow h$, a *vertical composite* $\theta \bullet \gamma : f \Rightarrow h$;

	objects	1-cells	2-cells
bicategory of comprehension categories	comprehension categories	pseudo maps	transformations
bicategory of display map categories	display map categories	maps	natural transformations

Table 2.1: Bicategory Data

- for each $f : x \to y, g, h : y \to z$ and $\theta : g \Rightarrow h$, a *left whiskering* $f \lhd \theta : f \cdot g \Rightarrow f \cdot h$;
- for each $f, g : x \to y, h : y \to z$ and $\theta : f \Rightarrow g$, a *right whiskering* $\theta \rhd h : f \cdot h \Rightarrow g \cdot h$;
- a *left unitor* $\lambda(f) : \mathsf{id}_1(x) \cdot f \Rightarrow f$ and its inverse $\lambda^{-1}(f) : f \Rightarrow \mathsf{id}_1(x) \cdot f$;
- a *right unitor* $\rho(f) : f \cdot \mathsf{id}_1(x) \cdot f \Rightarrow f$ and its inverse $\rho^{-1}(f) : f \Rightarrow f \cdot \mathsf{id}_1(x)$;
- a *left* $\alpha(f, g, h) : f \cdot (g \cdot h) \Rightarrow (f \cdot g) \cdot h$ and a *right associator* $\alpha(f, g, h)^{-1} : (f \cdot g) \cdot h \Rightarrow f \cdot (g \cdot h)$;

together with a set of laws, omitted here.

An example of bicategories is the bicategory of categories, with categories as objects, functors as 1-cells and natural transformations as 2-cells. This is also a good example for the way in which we will be using bicategories, as we can organise the prior structures into bicategories with the data in Table 2.1.

The bicategory of comprehension categories was already implemented in the UniMath library (`bicat_comp_cat`), so we will not discuss its details in this thesis. The bicategory of display map categories was not yet implemented, so in this thesis we will give its formalisation in Section 4.2, and its implementation details will be discussed in Section 5.1.

As mentioned prior, pseudofunctors between bicategories are like functors between categories, but we also need to map 2-cells, and adjust the laws that must hold for it. The laws involving 1-cells will require invertible 2-cells and not equalities.

Definition 25. A *pseudofunctor* $F : \mathcal{B} \to \mathcal{C}$ consists of:

- A *function* $F_0 : \mathcal{B}_0 \to \mathcal{C}_0$;
- a *function* $F_1 : (x \to y) \to (F_0(x) \to F_0(y))$;
- a *function* $F_2 : (f \Rightarrow g) \to (F_1(f) \Rightarrow F_1(g))$;
- for each $x : \mathcal{B}_0$, an *invertible 2-cell* $F_i(x) : \mathsf{id}_1(F_0(a)) \Rightarrow F_1(\mathsf{id}_1(x))$;
- for each $f : x \to y$ and $g : y \to z$, an *invertible 2-cell* $F_c(f, g) : F_1(f) \cdot F_1(g) \Rightarrow F_1(f \cdot g)$;

together with a set of laws, omitted here.

The laws omitted for the pseudofunctor are all preservation laws, up to invertible 2-cells for the 1-cells, and up to equality for the 2-cells. There is a lot more of them, as we have more operators which all need to be preserved. As we will see in Section 4.4.4 and Section 4.4.5, in our case most of these laws will be simple to prove.

2.4 Displayed Bicategory

Introduced by Ahrens and Lumsdaine [8, Definition 3.1], *displayed categories* are a concept that aims to formalise the notion of a category being 'over' another, as discussed in Section 2.2.1. It is also a useful definition when creating new categories using existing ones. This latter property is why we will be discussing the bicategorical version, *displayed bicategories*, as they will provide a modular approach to our formalisation. Further details on this modularity will be given in Section 5.1.

Displayed bicategories are similar enough in concept to displayed categories, but will necessarily require the addition of a new collection for displayed 2-cells over the 2-cells of the base. It will further require extra laws and operators, corresponding to those of the bicategory. Similar to displayed categories, the laws will be dependent on the base, and similar to bicategories, laws relating to 1-cells will take the form of invertible 2-cells.

Definition 26 ([11, Definition 6.1])**.** Given a *bicategory* $\mathcal{B}$, a displayed bicategory $\mathcal{D}$ over $\mathcal{B}$ consists of

- for each $a : \mathcal{B}_0$, a collection $\mathcal{D}_a$ of *displayed 0-cells* $\bar{a}$ over a;
- for each $f : a \to b$ and $\bar{a} : \mathcal{D}_a$, $\bar{b} : \mathcal{D}_b$,
  a collection $\bar{a} \underset{f}{\to} \bar{b}$ of *displayed 1-cells* $\bar{f}$ over f;
- for each $\theta : f \Rightarrow g$ and $\bar{f} : \bar{a} \underset{f}{\to} \bar{b}$, $\bar{g} : \bar{a} \underset{g}{\to} \bar{b}$,
  a collection $\bar{\theta} : \bar{f} \underset{\theta}{\Rightarrow} \bar{g}$ of *displayed 2-cells* over θ;

together with *dependent* versions of the operations and laws of a *bicategory*.

Building on our example for the bicategory (bicategory of categories), an example of a displayed bicategory could be the bicategory of categories with a terminal object. For this, the collection of displayed 0-cells would be the collection of terminal objects in a given category, and the collection of displayed 1-cells will be a single element over 1-cells (functors) that preserve terminal objects, and empty otherwise. The collection of displayed 2-cells will all have a single element over them, as there is no condition placed on them.

Given a displayed bicategory, its elements can be paired up with their counterparts in the base bicategory, in order to form a new bicategory, the so called *total bicategory*. In the following definition, we will use the dependant pair notation, which is often used in dependent type theory, for its definition see Section 3.1.

Definition 27 ([11, Definition 6.2])**.** Given a *displayed bicategory* $\mathcal{D}$ over a bicategory $\mathcal{B}$, the *total bicategory* $\int \mathcal{D}$ consists of:

- the type of *0-cells* is the type of tuples $(a, \bar{a}) : \sum_{a:\mathcal{B}_0} \mathcal{D}_a$;
- the type of *1-cells* is the type of tuples $(f, \bar{f}) : \sum_{f:a\to b} \bar{a} \underset{f}{\to} \bar{b}$;
- the type of *2-cells* is the type of tuples $(\theta, \bar{\theta}) : \sum_{\theta:f\Rightarrow g} \bar{f} \underset{\theta}{\Rightarrow} \bar{g}$;

Operators are defined using the operators of $\mathcal{D}$ and $\mathcal{B}$. The laws will hold since they hold in both $\mathcal{D}$ and $\mathcal{B}$.

Continuing the example, the total category would have pairs of categories and their terminal objects as 0-cells, terminal preserving functors paired with a singleton as 1-cells, and natural transformations paired with singletons as 2-cells. For the implementation of this displayed bicategory, see `disp_bicat_terminal_obj`, and for its total version see `univ_cat_with_terminal_obj`.

2.4.1 Product of Displayed Bicategories

One of the useful constructions using displayed bicategories is combining two of them to get their 'product'. To be able to define one, we need a bicategory $\mathcal{B}$, with two displayed bicategories $\mathcal{D}$ and $\mathcal{D}'$ over it. The 0-cells over a 0-cell $x : \mathcal{B}$ will be defined as pairs made from the 0-cells in the two displayed bicategories $\bar{x} : \mathcal{D}_x$ and $\bar{x}' : \mathcal{D}'_x$. The process is similar for 1-cells and 2-cells as well.

Definition 28 ([11, Definition 6.6 (1)])**.** Given a bicategory $\mathcal{B}$, and two displayed bicategories $\mathcal{D}$ and $\mathcal{D}'$ over it, the data for the product of the displayed bicategories is defined as

- for each 0-cell $x : \mathcal{B}$, the collection of displayed 0-cells is given by pairs $(\bar{x}, \bar{x}')$, made from displayed 0-cells $\bar{x} : \mathcal{D}_x$ and $\bar{x}' : \mathcal{D}'_x$;
- for each 1-cell $f : x \to y$, the collection of displayed 1-cells is given by pairs $(\bar{f}, \bar{f}')$, made from displayed 1-cells $\bar{f} : \mathcal{D}_f[\![\bar{x}, \bar{y}]\!]$ and $\bar{f}' : \mathcal{D}'_f[\![\bar{x}', \bar{y}']\!]$;
- for each 2-cell $\theta : f \Rightarrow g$, the collection of displayed 2-cells is given by pairs $(\bar{\theta}, \bar{\theta}')$, made from displayed 2-cells $\bar{\theta} : \mathcal{D}_\theta[\![\bar{f}, \bar{g}]\!]$ and $\bar{\theta}' : \mathcal{D}'_\theta[\![\bar{f}', \bar{g}']\!]$.

The identities, composition and other operators are given by using the corresponding operators in the two displayed bicategories. For example, the composition of 1-cells would be:

$$(\bar{f}, \bar{f}') \cdot (\bar{g}, \bar{g}') := (\bar{f} \cdot \bar{g}, \bar{f}' \cdot \bar{g}')$$

This has been already been implemented in the UniMath library as `disp_dirprod_bicat`. We will use this later, as it makes it easier to prove certain properties, such as univalence, as in Definition 32. For more information on how we use this in the implementation, check Section 5.1.

Chapter 3

Univalent Foundations

Univalent Foundations (UF), sometimes called Homotopy Type Theory, is a foundation for mathematics, that is, it gives a set of axioms (statements accepted without proofs) to serve as the basis of mathematical formalisation. Another such foundation is *set theory with classical logic* or *classical foundations*, most readers should be, perhaps not consciously, familiar with these foundations, as they are what is commonly understood as the base of mathematics. Univalent Foundations combines *type theory* and *homotopy theory*, as well as the *univalence axiom* to serve as the base, which is where the two names come from.

There have been multiple implementations of UF in proof assistants, one of them being the UniMath library, written in Rocq, that "aims to formalize a substantial body of mathematics using the univalent point of view."[1] When implementing our work, we will be relying on, and extending this library. For further implementation details, refer to Chapter 5.

As the second name, Homotopy Type Theory implies, UF is a kind of type theory. We will give an informal overview of what this means in Section 3.1, in order to be able to present the univalence axiom in Section 3.2, and the extension of the same principle to categories, resulting in univalent categories, as presented in Section 3.3. Finally, we will finish this chapter by giving an insight into formal dependent type theory in Section 3.4, followed by the interpretation of display map category and comprehension category as a model for dependent type theory in Sections 3.4.1 and 3.4.2 respectively. For the purpose of this thesis, the information presented in this chapter should be enough, but for those looking for a more comprehensive description, we recommend the book [12].

3.1 Dependent Type Theory

In this section, we will give an overview of dependent type theory, in a relaxed way. This should give enough context to accurately describe concepts, but will not be rigorous enough to act as foundation. For the more formal version, check Section 3.4.

As the central concept, we will have *types* denoted with a capital letter A, B, which are not unlike their programming counterparts and will contain *elements* denoted with a lowercase letter and using colon for the relation $a, \ldots : A$. Some examples for types are the empty type $\mathbf{0}$; the unit type $\mathbf{1}$ and the type of natural numbers nat. Given two types A, B, we can have a function type $A \to B$ whose elements will be the functions mapping elements of A to elements of B.

Importantly, in dependent type theory we can also have *dependent types*, also called families of types. Before we can introduce them, we need to make a small detour and define *universes*. Universes are essentially a *type of types*, with some crucial restrictions.

[1] UniMath repository

They follow a strict hierarchy, where each universe contains the smaller ones.

$$\mathcal{U}_0 : \mathcal{U}_1 : \mathcal{U}_2 : \mathcal{U}_3 : \dots$$

Note that the suffixes are not of the type nat, they are just to show the hierarchy. Notably, this hierarchy does not allow self-containing universes $\mathcal{U}_i : \mathcal{U}_i$, thereby avoiding Russell's paradox[2], among others. If a type is an element of some universe $A : \mathcal{U}_i$, then it is also an element of all larger universes $A : \mathcal{U}_j, i < j$. Due to this inclusion, the subscript is often left off, since we will not dabble with universes directly, we too shall do this. This style is called *typical ambiguity*.

Returning to *dependent types*, we define them as $B : A \to \mathcal{U}$, mapping the elements of a type to types in some universe, hence the name family of types. An example of such a type could be $\mathsf{IsEven} : \mathsf{nat} \to \mathcal{U}$, mapping even numbers to the type $\mathbf{1}$ and odd numbers to $\mathbf{0}$. Note that this is different from the function $\mathsf{even} : \mathsf{nat} \to \mathsf{bool}$, which maps even numbers to true and odd numbers to false.

It might be that we only want to map to a single element within the target type, which we can do using *dependent functions*. Given a family of types $B : A \to \mathcal{U}$, we can define *dependent functions*

$$f : \prod_{a:A} B(a).$$

These functions map elements of A to elements in the corresponding dependent type $B(a)$. An example of this would be a function mapping elements of $n : \mathsf{nat}$ to the tuple made of $n + 1$ copies of themselves, as follows:

$$\begin{aligned}
0 &\mapsto 0 && : \mathsf{nat} \\
1 &\mapsto (1, 1) && : \mathsf{nat} \times \mathsf{nat} \\
2 &\mapsto (2, 2, 2) && : \mathsf{nat} \times \mathsf{nat} \times \mathsf{nat} \\
n &\mapsto (n, n, n, \dots) && : \mathsf{nat} \times \mathsf{nat} \times \mathsf{nat} \times \cdots
\end{aligned}$$

Finally, we could also just create pairs out of corresponding elements from A and $B(a)$. Given a family of types $B : A \to \mathcal{U}$, we can also create *dependent pairs*

$$(a, b) : \sum_{a:A} B(a)$$

which will do this exact thing. A simple example would be a type that pairs up natural numbers n, with tuples that have n elements. A special example could be $\sum_{n:\mathsf{nat}} \mathsf{IsEven}(n)$, a type that would only have pairs of even numbers and an element of the unit type, e. g. $(0, t), (2, t), (4, t), \dots$ This construction is a special one named *subtype*, as it has only some elements of the type A and we will make use of this construction throughout the implementation. Further, a more precise definition of a subtype will be given at the end of this section.

Usually in type theory, elements can be compared using definitional equality, but this is cumbersome for reasoning within type theory. Instead, we have a special type, the equality or identity type between two elements $a, a' : A$

$$a =_A a'.$$

This type will exist for any two elements of the same type, and never between elements of different types. For this reason, in the implementation the computer sometimes needs to be convinced that two elements have the same type, to this end we use transports. In the

[2]https://plato.stanford.edu/entries/russell-paradox/

formalisation section we will instead denote this as equality with a subscript $=_*$. The identity type can have any number of elements, but we must have at least the element given by reflexivity

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

Certain types can have more than one path between equal elements, which can be useful when working with topology. For two non-equal types, their equality type will be empty.

This flexibility with the identity type then allow us to order types into so-called h-levels. These should not be confused with universes, as h-types describe a structure within a type, while universes enforce an external hierarchy. The higher h-levels are most useful for modelling topological spaces, therefore, we will only discuss the lowest three levels. The lowest level is the level of contractible types, or types with a single element.

$$\mathsf{iscontr}(X : \mathcal{U}) := \sum_{(x:X)} \prod_{(x':X)} x =_X x' \tag{3.1}$$

The dependent pair enforces that we do have a term in X, and the rest enforce that there is only one unique element. Due to univalence, any contractible type will be equal to the unit type $\mathbf{1}$.

Then, the next level will be *propositions*, that is, types that have either no or just a single element. We can state what it means to be a proposition as follows

$$\mathsf{isaprop}(X : \mathcal{U}) := \prod_{(x,x':X)} \mathsf{iscontr}(x =_X x') \tag{3.2}$$

If we unfold iscontr, this will state that for any two elements x, x' in X, their identity type $x =_X x'$ must be inhabited by a single unique element. This will also allow us to introduce the interpretation of 'propositions-as-types', where we say that $a : A$ is logically equivalent to saying "a is a proof of A". Similarly, we can interpret uninhabited types as being false. The only issue with this second part is that we cannot directly show that a type is uninhabited, instead we can construct the function $A \to \mathbf{0}$, which should only exist for empty types, otherwise there would be an element in $\mathbf{0}$.

Further, we can follow the same pattern to define sets in these foundations. Sets will therefore have identity types between their elements be propositions.

$$\mathsf{isaset}(X : \mathcal{U}) := \prod_{(x,x':X)} \mathsf{isaprop}(x =_X x') \tag{3.3}$$

In fact, this pattern can be continued to inductively define what it means to be of a certain h-level. As noted prior, the unit type $\mathbf{1}$ is both contractible and a proposition, further it is also a set. This is the case for all types, if they are of some level, then they are also of every subsequent level. This is similar to how universes accumulate types, though these two are not the same. As mentioned prior, these levels describe the internals of types, while universes are more external.

There are some related concepts that we will use, when doing the formalisation. The first of these will be the type of propositions.

$$\mathsf{hProp} := \sum_{X:\mathcal{U}} \mathsf{isaprop}\ X \tag{3.4}$$

The h in the name of both refers again to homotopy, and is included as proof assistants often provide their own versions of these types, and thus would conflict with these. Elements (X, H) of this type will be equal to the type contained within X, this is a consequence of isaprop being a proposition as well, and since univalence will allow us to equate equivalent types (see Section 3.2). Further, we can define what it means for a type family to be a predicate.

$$\mathsf{isPredicate}(B : A \to \mathcal{U}) := \prod_{(a:A)} \mathsf{isaprop}(B(a)) \tag{3.5}$$

It just means that all of the types in the family is a proposition. This will be useful when dealing with subtypes, which are of the form $\Sigma_{a:A} P(a)$, where P is a predicate. Any two elements of such a type will be equal if, and only if the first element of the pair is the same, this can be proven as a result of $P(a)$ being a proposition, thus the second element must be the single unique element in $P(a)$. This is also the reason for the name subtype, as this new type $\Sigma_{a:A} P(a)$ will have a similar structure to A, but with (potentially) fewer elements.

3.2 Univalence Axiom

Finally, we can discuss one of the more special aspects of Univalent Foundations, that being the *univalence axiom*. As mentioned prior, UF is a basis for mathematics, and as such, defines a set of axioms on top of which it aims to build everything else. One of these is the univalence axiom, and it concerns notions of sameness for types. The first notion will be (propositional) equality, i. e. the identity type $=_{\mathcal{U}}$ between two types. The other notion is equivalence $\simeq$, which is a function that has an inverse. Naturally, if two types are equal, it should be possible to create an equivalence between them.

Definition 29. Given two types A, B in a universe $\mathcal{U}$, we can define a function

$$\mathsf{idtoeqv} : (A =_{\mathcal{U}} B) \to (A \simeq B)$$

that turns the equality $(=_{\mathcal{U}})$ of two types into an equivalence $(\simeq)$. An equivalence is just a function that has an inverse on both sides, i. e. e is an equivalence if there exist two functions f and g, such that, $e \cdot f = \mathsf{id}$ and $g \cdot e = \mathsf{id}$.

We can informally define this function as mapping equalities to the identity, since if two types are equal, then their identity function will be an equivalence, since $\mathsf{id} \cdot \mathsf{id} = \mathsf{id}$.

What the *univalence axiom* states, is that this function idtoeqv should be an equivalence. Meaning that an equivalence can be turned into an equality.

Definition 30 ([12, Axiom 2.10.3])**.** For any $A, B : \mathcal{U}$, the function idtoeqv is an equivalence. Meaning that we have

$$(A =_{\mathcal{U}} B) \simeq (A \simeq B).$$

As a consequence of this, any type that has only a single term can be replaced by the unit type $\mathbf{1}$, as they are equivalent. Similarly, the types $A \times B$ and $B \times A$ will be equal, as the function

$$\mathsf{swap} : A \times B \to B \times A := (a, b) \mapsto (b, a)$$

is an equivalence between the two of them. This may appear strange at first, but there is no reason why our labelling of the terms should make these types different.

Another result of the univalence axiom is functional extensionality, i. e. that two functions are equal if and only if they are point-wise equal. We mention this example to illustrate that the univalence axiom does agree with some of our expectations of what equality means. The full derivation of this result is beyond the scope of this thesis, and is not needed for it, but for those interested, we recommend checking out the Homotopy Type Theory Book [12, Section 4.9].

3.3 Univalent Categories

Introduced by Ahrens, Kapulkin and Shulman [13], *univalent categories* are categories that satisfy a property similar to the *univalence axiom*. Ahrens, Kapulkin and Shulman consider it "to be the 'correct' definition of *category* in UF", and it has indeed been widely used in the formalisation efforts in UniMath.

Definition 31 ([13, Definition 3.6]). A category $\mathcal{C}$ is said to be univalent if for any two objects A, B, the map $\mathsf{idtoiso} : A = B \to A \cong B$ (mapping equivalences to isomorphisms), is an equivalence.

The notion has later been extended to bicategories as well, by Ahrens et al. [11]. Since in a bicategory we have two types of 'arrows', 1-cells and 2-cells, the notion of univalence can be extended to isomorphisms of either, giving global and local univalence respectively.

Definition 32 ([11, Definition 3.1]). A bicategory $\mathcal{B}$ is said to be univalent, if it is univalent both *locally* and *globally*.

- It is *locally univalent*, if for any $f, g : a \to b$, the function $\mathsf{idtoiso}^{2,1} : f = g \to f \cong g$ (mapping identities to invertible 2-cells) is an equivalence;
- It is *globally univalent*, if for any $a, b : \mathcal{B}_0$, the function $\mathsf{idtoiso}^{2,0} : a = b \to a \simeq b$ (mapping identities to adjoints) is an equivalence;

We will use the above definition when describing the bicategory we construct in Section 4.2.

As mentioned in the previous chapter, we can show that given two univalent displayed bicategories, their product (as defined in Definition 28) is also univalent. To construct the required equivalences directly in the implementation can be cumbersome. Instead, a pattern in the UniMath library is to break up the equivalence into smaller ones that are easier to show. Then, an equivalence can be created by concatenating them, but this leaves a final step, to show that this new function is equal to idtoiso, which is often done using functional extensionality. This makes the proofs rather lengthy, so we will omit them here, but they can be found in the UniMath library as `is_univalent_2_1_dirprod_bicat` and `is_univalent_2_0_dirprod_bicat`. We will also use a similar process to create one of the equivalences required in Section 4.3.

3.4 Formal Type Theory

So far, we have discussed Univalent Foundations in a somewhat informal manner. However, to be able to discuss how the categorical structures in Sections 2.1.1 and 2.2.2, we first need to discuss the formal version of dependent type theory. For those interested in more details about the formal version of UF, we recommend checking out the appendix "Formal type theory" in the Homotopy Type Theory book [12], as we use their second presentation as the reference for this section. The book gives a more comprehensive view of the formal version of UF, while this section is meant only to give enough context to describe how display map category and comprehension category model dependent type theories.

Formally, types exist in *contexts* Γ, Δ, which are effectively just a list of variables that belong to some type. The only restriction on this list is that its members must be well-typed, where being well-typed means the type of the variable exists in the context given by the other variables before it. This is similar to how we use the word context while programming, referring to currently declared variables. Further, we can state various judgements about contexts, types, elements and their relations.

- Γ ctx — Γ is a context;
- $\Gamma \vdash A$ — the type A exists in some context Γ;
- $\Gamma \vdash a : A$ — there is an element a of type A in context Γ;
- $\Gamma \vdash A \equiv B$ — the types A and B are equal by definition;
- $\Gamma \vdash a \equiv a' : A$ — the elements a and a' are equal by definition;

Just stating a judgement does not mean it necessarily holds, for that we need to also have rules. These rules are akin to rules of computation, and this is one of the reasons we use type theory in formalisation with proof assistants. An example of a rule looks like

$$\frac{\mathcal{J}_0 \qquad \mathcal{J}_1 \qquad \mathcal{J}_2 \qquad \cdots \qquad \mathcal{J}_n}{\mathcal{J}}\ \textsc{example}$$

and can be read as "if $\mathcal{J}_0, \mathcal{J}_1, \ldots$ all hold, then $\mathcal{J}$ will also hold." There are some general rules for handling contexts and, for Univalent Foundations there are also rules for universes. In the remainder of this section, we will present some of the rules for dependent type theory. This will include general rules included in most dependent type theories, some Univalent Foundations specific rules for universes and finally some examples of types, namely the empty and the unit type.

Contexts. We have a number of rules governing how to form and use contexts.

$$\frac{}{\cdot\ \mathsf{ctx}}\ \textsc{empty} \qquad \frac{\Gamma\ \mathsf{ctx} \qquad \Gamma \vdash C : \mathcal{U}_i}{\Gamma, x : C\ \mathsf{ctx}}\ \textsc{ext} \qquad \frac{\Gamma, x : C\ \mathsf{ctx}}{\Gamma, x : C \vdash x : C}\ \textsc{var}$$

The first of these states that given nothing else, we can create the empty context. The second states that given a context, and a type in that context, we can extend the context with a variable in that type. Note that the rule does not require an element in said type. The final one states that if we have a variable in our context, then we can use it as a term, importantly, this term is still dependent on the variable in the context.

Weakening. If we have some judgement $\mathcal{J}$, given some context Γ, it stands that we must still have that judgement, even if we extend the context. This is formalised in the *weakening* rule as follows

$$\frac{\Gamma, \Delta \vdash \mathcal{J} \qquad \Gamma \vdash C}{\Gamma, x : C, \Delta \vdash \mathcal{J}}\ \textsc{weak}$$

This rule can then be combined with others to enable greater manipulation of contexts, when using them in derivations of judgements. An example will be combining it with the following rule.

Substitiution. Another important rule is substitution, where we can replace variables with well-typed terms.

$$\frac{\Gamma, x : A \vdash \mathcal{J} \qquad \Gamma \vdash a : A}{\Gamma \vdash \mathcal{J}[a/x]}\ \textsc{subst}$$

In the above rule $\mathcal{J}$ is some judgement in context $\Gamma, x : A$, and $\mathcal{J}[a/x]$ is the same judgement, but all occurrences of x have been replaced by the term a. For example,

$$\frac{x : \mathsf{nat} \vdash x + x : \mathsf{nat} \qquad \cdot \vdash 2 : \mathsf{nat}}{\cdot \vdash 2 + 2 : \mathsf{nat}}\ \textsc{subst}$$

As mentioned for the weakening rule, it can be combined with substitution and other rules to allow for simultaneous substitution of multiple variables in context. Given two contexts Γ and Δ, such that, Δ is of the form $(x_0 : A_0, x_1 : A_1, \dots, x_n : A_n)$, the *context morphism* s from Γ to Δ will be a list of terms $t_0, t_1, \dots, t_n$. For it to be a substitution, it must hold that

$$\Gamma \vdash t_i[t_0/x_0,\ t_1/x_1, \dots,\ t_{i-1}/x_{i-1}].$$

Given such a context morphism, one can simultaneously substitute multiple variables. These context morphisms are used in the semantic frameworks to model the effects of substitutions.

Universes. As given in the book [12], universes can also be formalised to give rules for them. The first rule introduces that we have universes in all contexts, and that they have a certain hierarchy. The second rule introduces that they must accumulate types, so any type in a 'smaller' universe is also a type in all 'bigger' universes.

$$\frac{\Gamma\ \mathsf{ctx}}{\Gamma \vdash \mathcal{U}_i : \mathcal{U}_{i+1}}\ \mathcal{U}\text{-}\textsc{intro} \qquad\qquad \frac{\Gamma \vdash A : \mathcal{U}_i}{\Gamma \vdash A : \mathcal{U}_{i+1}}\ \mathcal{U}\text{-}\textsc{cumul}$$

Universes are incompatible (as is) with the models we will present, due to them being both types and elements, while the models will make clear distinctions between the two. There have been more complex models that do include universes, such as the one given by Lumsdaine and Warren [14].

Types. Types are usually declared by giving a set of rules that fall under the following kinds

- *type formation rules* —stating when the type exists;
- *introduction rules* —stating how to derive terms of the type;
- *elimination rules* —stating how to use terms of the type;
- *computation rules* —stating how the introduction and elimination rules interact;
- *uniqueness principles* —optional, stating the uniqueness of the terms.

A type does not have to have all of these to be a type, and it can have multiple rules that fall into the same category. To see how they work, we will give the two simplest examples, the empty and the unit type.

Example: empty type. This is one of if not the simplest types we can create, as it should have no terms. This means that there is no need to define any introduction rules, which will also mean no computation rules.

$$\frac{\Gamma\ \mathsf{ctx}}{\Gamma \vdash \mathbf{0} : \mathcal{U}_i}\ \mathbf{0}\text{-}\textsc{form} \qquad\qquad \frac{\Gamma, x : \mathbf{0} \vdash C : \mathcal{U}_i \qquad \Gamma \vdash a : \mathbf{0}}{\Gamma \vdash \mathsf{ind}_0(x.C, a) : C[a/x]}\ \mathbf{0}\text{-}\textsc{elim}$$

The first rule here states that the empty type exists in all contexts. Technically, this rule could state that it exists only given the empty context, but the current form could then be derived using the weakening rule. Similarly, it would suffice to only state that **0** is a term of the universe smallest $\mathcal{U}_0$. The second rule corresponds to the existence of the function $\prod_{x:0} C$, from the empty type to any other type. This function always exists as it has to map no elements, since **0** is empty.

Example: unit type. A similarly simple type is the type **1** with only one term $\star$.

$$\frac{\Gamma\ \mathsf{ctx}}{\Gamma \vdash \mathbf{1} : \mathcal{U}_i}\ \mathbf{1}\text{-FORM} \qquad \frac{\Gamma \vdash \mathbf{1} : \mathcal{U}_i}{\Gamma \vdash \star : \mathbf{1}}\ \mathbf{1}\text{-INTRO}$$

$$\frac{\Gamma, x : \mathbf{1} \vdash C : \mathcal{U}_i \qquad \Gamma \vdash c : C[\star/x] \qquad \Gamma \vdash a : \mathbf{1}}{\Gamma \vdash \mathsf{ind}_1(x.C, c, a) : C[a/x]}\ \mathbf{1}\text{-ELIM}$$

$$\frac{\Gamma, x : \mathbf{1} \vdash C : \mathcal{U}_i \qquad \Gamma \vdash c : C[\star/x]}{\Gamma \vdash \mathsf{ind}_1(x.C, c, \star) \equiv c : C[\star/x]}\ \mathbf{1}\text{-COMP}$$

The formation rule is essentially the same as the empty type, but this time we do also have a single introduction rule, giving the term $\star$. This also means we need corresponding elimination and computation rules. The elimination states that if we have a type dependent on **1**, then if a term exists in the type dependent on $\star$, then there is a term for all elements in **1**, and the computation rule guarantees that this element will be the same as the one for $\star$, since it is the only element in **1**.

In the following two sections, we will present an overview of how display map category and comprehension category get used to model dependent type theories. We will not discuss universes in these sections, as both models make clear distinctions between terms and types, which causes problems since universes (and also all types) are both terms and types in UF. On a similar note, it is worth to mention that the UniMath library does not have multiple universes, and relies on typical ambiguity. This is due to the internals of the theory behind Rocq, the details of this are beyond the scope of this thesis.

3.4.1 Display Map Category as a Semantic Framework

We are formalising display map categories in order to show their correspondence with other semantic frameworks. For this comparison to make sense, we also have to discuss how DMCs get used as a semantic framework. It is formalising the latter, formal version of type theory presented above. In the following, we will be considering a display map category $(\mathcal{C}, \mathcal{D})$.

Contexts are modelled to be objects $\Gamma, \Delta, \ldots$ of the base category $\mathcal{C}$. Sometimes this interpretation is denoted with $[\Gamma]$ instead, but we will use the prior for simplicity. *Types* that exist given a context Γ are taken to be a display map A into the object Γ.

$$\begin{tikzcd}\Gamma.A \arrow[d, "A"] \\ \Gamma\end{tikzcd}$$

The domain of this display map A is the context extended with some term $a : A$. *Terms* of some type A are the sections of the display map A.

$$\begin{tikzcd}\Gamma \arrow[r, "a"] \arrow[dr, "\mathsf{id}"'] & \Gamma.A \arrow[d, "A"] \\ & \Gamma\end{tikzcd}$$

Section here means that the above diagram commutes, i. e. $a \cdot A = \mathsf{id}$.

The assumed pullbacks of display map categories will model the effects of context morphisms. A *context morphism* s from Γ to Δ will be interpreted as the morphism

$$\Gamma \xrightarrow{s} \Delta.$$

So, given a type A in context Δ, the substitution will be given by the pullback

$$\begin{tikzcd}
\Gamma.A[s] \arrow[r] \arrow[d, "A[s]"'] \arrow[dr, phantom, very near start, "\lrcorner"] & \Delta.A \arrow[d, "A"] \\
\Gamma \arrow[r, "s"'] & \Delta
\end{tikzcd}$$

Where $A[s]$ is the new type, after the substitution. Furthermore, the pullback of a display map along another display map will correspond to the weakening rule as follows.

$$\begin{tikzcd}
\Gamma.A.B \arrow[r, dashed] \arrow[d, "B"'] \arrow[dr, phantom, very near start, "\lrcorner"] & \Gamma.B \arrow[d, "B"] \\
\Gamma.A \arrow[r, "A"] & \Gamma
\end{tikzcd}$$

We do not need to substitute anything in B, as it does not depend on A, and nothing else changed in the context.

Depending on the exact type theory being modelled, extra conditions can be added to this model. One such condition that is often added, is the need for a terminal object, which will represent the empty context. The various types presented can also be modelled, using different structures.

Empty Type. To model the empty type, for any object Γ we need to have a display map $\mathbf{0}$, with no sections. The lack of sections will mean that this type will not have any terms. We do not explicitly model the elimination rule.

Unit Type. For the unit type, we need to have, for all objects Γ, a display map $\mathbf{1}$, with exactly one section. This will mean that, for any other type $\Gamma \vdash A$, there will be a unique morphism from $\Gamma.A$ to $\Gamma.\mathbf{1}$, such that the following diagram commutes.

$$\begin{tikzcd}
\Gamma.A \arrow[rr, dashed] \arrow[dr, "A"'] & & \Gamma.\mathbf{1} \arrow[dl, bend left, "\mathbf{1}"] \\
& \Gamma \arrow[ur, dashed, bend left, "\star"] &
\end{tikzcd}$$

3.4.2 Comprehension Category as a Semantic Framework

Now, given that we will be showing the correspondence between these two structures, this section will describe a fairly similar interpretation to the one for DMCs. For the interpretation we will be considering a comprehension category with base $\mathcal{C}$, fibration $p : \mathcal{T} \to \mathcal{C}$ and comprehension $\chi : \mathcal{T} \to \mathcal{C}^{\to}$.

Contexts will be once more objects in the base category $\mathcal{C}$. *Types* will now be represented by objects in the category $\mathcal{T}$, such that, the judgement $\Gamma \vdash A$ corresponds to $p(A) = \Gamma$ or $A : \mathcal{T}_\Gamma$, depending on notation. This will mean that the comprehension χ will map this object to a morphism

$$\begin{tikzcd}
\Gamma.A \arrow[d, "\pi_A"] \\
\Gamma.
\end{tikzcd}$$

Terms of a type will once again be sections, but this time of the morphism π_A.

$$\begin{tikzcd}
\Gamma \arrow[r, "a"] \arrow[dr, "\mathsf{id}"'] & \Gamma.A \arrow[d, "\pi_A"] \\
& \Gamma.
\end{tikzcd}$$

Finally, the *context morphisms* will once more be morphisms in the category $\mathcal{C}$. Since p is a fibration, and we require the comprehension χ to preserve cartesian morphisms, and since a cartesian arrow for the functor cod corresponds to a pullback in the base, we once again get a pullback square for substitutions.

$$\begin{tikzcd}
\Gamma.A[s] \arrow[r] \arrow[d, "\pi_{A[s]}"'] \arrow[dr, phantom, "\lrcorner", very near start] & \Delta.A \arrow[d, "\pi_A"] \\
\Gamma \arrow[r, "s"'] & \Delta
\end{tikzcd}$$

The weakening rule will now be given by the cartesian lift of the comprehension of a type. So, given a type $\Gamma \vdash A$ and another type $\Gamma \vdash B$, we have the following

$$\begin{tikzcd}
\mathcal{T} \arrow[d, "p"'] & B_{\Gamma.A} \arrow[r] & B_{\Gamma} & \Gamma.A.B \arrow[r, dashed] \arrow[d, "\pi_B"'] \arrow[dr, phantom, "\lrcorner", very near start] & \Gamma.B \arrow[d, "\pi_B"'] \\
\mathcal{C} & \Gamma.A \arrow[r, "\pi_A"'] & \Gamma & \Gamma.A \arrow[r, "\pi_A"'] & \Gamma
\end{tikzcd}$$

The comprehension of the resulting cartesian arrow will be a familiar pullback square.

Furthermore, this model can also be extended by adding extra properties that the CC must satisfy. For example, a terminal object in the category $\mathcal{C}$ will once more correspond to the empty category.

The empty type **0**, or rather its comprehension, will have no sections once again. The unit type **1** will have a single section once more, which leads to a unique morphism from any other type in the same context, meaning that **1** will be a terminal object in $\mathcal{T}_\Gamma$, i. e. the subcategory of $\mathcal{T}$ over a context Γ.

Chapter 4

Formalisation

By formalisation, I here mean formalising the definition of display map category and related definitions, using the notation and definitions of Univalent Foundations. This is not to be confused with the act of what I will call implementation (in a proof assistant), which is often also called formalisation, and hence can lead to confusion. I make this distinction as this chapter will aim to give a language agnostic argumentation for my results, relying only on the notation of Univalent Foundations. Hopefully, this should enable potential verification of my results in other proof assistants or libraries easier. Further, this will also allow me to make arguments easier, as I will be able to leave out transports, which are a special 'identity' function that 'transports' a term into an equal type to its own (e. g. `transportf` $: A =_{\mathcal{U}} B \to A \to B$), and are used throughout the implementation. Finally, to make it easy to reference the corresponding implementation to each formalisation, I will add references to the code, where appropriate. For more detail on the implementation, check Chapter 5.

We will begin by formalising the central concept of this thesis, display map category together with some properties of this definition in Section 2.1. Further, the same section will also include formalisation of the condition of what it means to be a map between two display map categories, together with a proof that this is a predicate as per Equation (3.5). Then, we will construct the bicategory of display map categories as a displayed bicategory over the bicategory of univalent categories in Section 4.2. Further, as a bit of housekeeping, we will give proof that the bicategory we defined is univalent, and thus conforms to the 'correct' version of bicategories in UF. This is done in Section 4.3 and involves the creation of two equivalences, one trivial and one that will be constructed as a composition of smaller equivalences. Finally, to finish off this chapter, we will give the construction of a pseudofunctor from the above bicategory into the bicategory of full comprehension categories in Section 4.4. The most important parts of this pseudofunctor are the first two underlying functions, mapping the 0- and 1-cells, the rest of the section is just completing all other necessary proofs to complete the pseudofunctor.

4.1 Display Map Category

The central concept of this thesis is DMCs, which we will formalise in Univalent Foundations in this section. For this, we first give our formal definition, and discuss some of its details and differences to Definition 4. Furthermore, we will show some of the properties of this definition that will come handy in the rest of this chapter. Finally, we finish this section with the formalisation of maps between DMCs (see Definition 6). The reader might notice the lack of a definition for the transformation between these maps, which is due to the fact that they are just natural transformations, and those have already been formalised in Univalent Foundations.

Display map category was defined as a collection of display maps together with a stability condition in Definition 4. Therefore, to formalise it in Univalent Foundations, we will need to formalise these two. Since the morphisms in a category form a set, and we pick our collection of display maps from them, this collection too will be a set. A way to formalise sets is to give a function that will select the elements, and this will indeed be the choice we will make.

For the second part, the condition stated that given a display map $d : C \to A$ and a morphism $f : B \to A$, the pullback $f^*\ d$ of d along f should also be a display map. In our case, this will mean that the selection function from the first part should also include the pullback $f^*\ d$. This condition also presumes that a pullback $f^*\ d$ exists in the base category $\mathcal{C}$, which we will also need to include in our definition.

Definition 33 (`display_map_class`)**.** Given a category $\mathcal{C}$, a *display map category* $\mathcal{D}$ will consist of

- a set of maps, given by the dependent function $\mathcal{D} : \prod_{d:\mathcal{C}[\![a,b]\!]} \mathsf{hProp}$;
- a proof of the fact that $\mathcal{D}$ includes pullbacks of display maps

$$\mathsf{has\_pullbacks}(\mathcal{D}) := \prod_{(f:\mathcal{C}[\![b,a]\!])} \prod_{(d:\mathcal{C}[\![c,a]\!])} \mathcal{D}(d) \to \sum_{f^*\ d} \mathcal{D}(f^*\ d). \tag{4.1}$$

Here, hProp is the types of propositions, as presented in Equation (3.4). This means that d will be a display map, if $\mathcal{D}(d)$ is an inhabited type, thus an element $H : \mathcal{D}(d)$ can be understood as a proof of d being a display map. The second part states that given a morphism f and a display map d, they must have a pullback f^*d, and that this must also be a display map $\mathcal{D}(f^*\ d)$.

In the definition above, we chose to use $\sum$ in Equation (4.1) that requires the existence of a specific pullback, alternatively, we could have used the truncated version $\exists$, which requires a mere existence of a pullback. We chose this version as it is simpler to work with and, as it turns out, this choice does not have consequences for the rest of the formalisation, for reasons similar to why we have not included the *repleteness* condition. This, as mentioned in Remark 5, is due to the fact that taking $\mathcal{C}$ to be *univalent*, as in Definition 31, gives the same isomorphism invariance guarantees. Further, since univalent categories are considered to be the 'correct' version of categories by Ahrens, Kapulkin and Shulman [13], this is a natural thing to require.

Remark 34. *display map category*, as defined in Definition 33, satisfies the repleteness condition, if the base $\mathcal{C}$ is univalent (as in Definition 31).

Proof. The repleteness condition states that given a display map $d : x \to y$, and two isomorphisms $i : x \to x'$ and $j : y \to y'$, any other arrow $d' : x' \to y'$, such that the following diagram commutes,

$$\begin{tikzcd} x \arrow[r, leftrightarrow, "i"] \arrow[d, "d"', -latex] & x' \arrow[d, "d'"] \\ y \arrow[r, leftrightarrow, "j"'] & y' \end{tikzcd}$$

must also be a display map. Since $\mathcal{C}$ is univalent, we can replace the isomorphisms with equalities.

$$\begin{tikzcd} x \arrow[r, equal] \arrow[d, "d"', -latex] & x' \arrow[d, "d'"] \\ y \arrow[r, equal] & y' \end{tikzcd}$$

From this it is easy to see that we must have

$$d =_* d',$$

a *dependent* equality, as the two sides differ in types. From this, it is simple to conclude that d' must be a display map too. ◻

Further, since all pullbacks (of the same two morphisms) must be isomorphic (see Remark 3), the base category $\mathcal{C}$ being univalent will also make them equal. This means that all pullbacks will be equal, therefore it does not matter whether we choose $\exists$ or Σ in the Definition 33. As a consequence of this, we can show that the stability condition is in fact just a predicate on the type of sets of display maps. For this, we must show that has_pullbacks maps to a proposition type. We know, by definition, that the second element of the dependent pair $\mathcal{D}(f^*\ d)$ is a proposition, so the only thing to show is that the pullback is unique, which follows directly from above.

Remark 35 (`isPredicate_has_map_pullbacks`)**.** The function has_pullbacks, as defined in Equation (4.1), is a predicate in the sense of Equation (3.5). This is useful, as it means that *display map category* are a subtype of the type 'subset of morphisms in $\mathcal{C}$', and two of them are equal if and only if they include the same display maps.

As defined in Definition 6, we have an analogous concept for display map categories to functors. We know that a map is just a functor with an added property, so we will define the function that will tell us whether this property holds for a given functor. Using this function, if we show that it is indeed a predicate, we can then define the subtype of functors that will be the type of maps.

The property to be a map consists of two parts. First, the functor is required to preserve display maps, i. e. it has to map display maps in the first DMC to display maps in the second. Second, the functor has to also preserve pullbacks. In the following definition we will make an exception, and briefly explain the implementation of this second property in UniMath.

Definition 36 (`is_display_map_class_functor`)**.** Given two display map categories $\mathcal{D}$ and $\mathcal{D}'$, and a functor between their bases $F : \mathcal{C} \to \mathcal{C}'$, this functor will be a map $F : \mathcal{D} \to \mathcal{D}'$, if the following two types are inhabited.

First, the type corresponding to the preservation of display maps

$$\mathsf{preserves\_maps}(F : \mathcal{C} \to \mathcal{C}') := \prod_{(d:\mathcal{C}[\![a,b]\!])} \mathcal{D}(d) \to \mathcal{D}'(F_m\ d) \tag{4.2}$$

This is logically equivalent to saying 'd being a display map in D implies that its image $F\ d$ is a display map in $\mathcal{D}'$.'

For the second condition, we need to express that the image of the pullback of a display map will be a pullback.

$$\begin{tikzcd}
\bullet \arrow[r, dashed] \arrow[dd, "f^*\ d"'] \arrow[ddr, phantom, "\lrcorner", very near start] & \bullet \arrow[dd, "d"] & & \bullet \arrow[r, dashed] \arrow[dd, "F(f^*\ d)"'] \arrow[ddr, phantom, "\lrcorner", very near start] & \bullet \arrow[dd, "F(d)"] \\
 & {} \arrow[r, "F"] & {} & & \\
\bullet \arrow[r, "f"'] & \bullet & & \bullet \arrow[r, "f"'] & \bullet
\end{tikzcd}$$

For this, we can define the following

$$\mathsf{preserves\_pullbacks}(F : \mathcal{C} \to \mathcal{C}') := \prod_{(f:b\to a)} \prod_{(d:c\to a)} \mathcal{D}(d) \to \mathsf{isPullback}(F(f^*\ d)) \tag{4.3}$$

Here, we will just use a simplified version of isPullback, as the details will depend on how isPullback is defined. In the UniMath library, it takes the commutativity of the square, so the line looks like

```
isPullback (!functor_comp F _ _ @ maponpaths (#F) (PullbackSqrCommutes pb) @ functor_comp F _ _)
```

This maps the commutativity of the first square (`PullbackSqrCommutes`) to the second squares commutativity.

For completeness, we should also give proof of this definition being a predicate on functors, as a functor is either a map or not, and it can only be a map in one way. The proof for both properties being a predicate rely on that $\mathcal{D}$ selects display maps using a proposition, and that due to the univalence of the base category $\mathcal{C}$ all pullbacks of d along f are equal. Then, using these it follows that their pair would also be a predicate (`isPredicate_is_display_map_class_functor`). This will mean that maps between two display map categories will be a subtype of the functors between their bases.

Using the definitions in this section, we can define the arrow versions as well, as had been described in Section 2.1.3. Due to the similarity with the arrow category, their implementation follows the same structure. This similarity could be made even more explicit by defining $\mathcal{D}^{\twoheadrightarrow}$ as a subcategory of $\mathcal{C}^{\rightarrow}$, but we will just use the direct version, as the two should be equivalent. The corresponding code segments are as follows

- Definition 10 ($\mathcal{D}^{\twoheadrightarrow}$) — `display_map_cat`;
- Definition 12 ($F^{\twoheadrightarrow}$) — `display_map_functor`;
- Definition 13 ($\alpha^{\twoheadrightarrow}$) — `display_map_nat_trans`.

These are implemented as displayed versions, since we gave special significance to the codomain functor. The specifics of these implementations will be discussed further in Section 5.2.

While the formalisation in this section could be used to construct functions directly to the corresponding definitions for comprehension category, and I have indeed written those functions (see for example: `display_map_to_comprehension_category`), in the next sections we will instead follow the comparison paper in constructing a bicategory and defining a pseudofunctor. This will give extra guarantees about the functions defined, as to be a pseudofunctor, they will have to satisfy certain preservation laws. However, the effort of implementing these was not entirely useless, as parts of these functions were useful when constructing the pseudofunctor, since these are similar to the underlying functions of the pseudofunctor.

4.2 (Displayed) Bicategory of Display Map Categories

As stated in Section 2.3, we can organise DMCs into a bicategory. This can be done directly (see `bicat_display_map_cat`), but to allow for a bit more modularity, as well as to better utilise existing proofs in the UniMath library, we can also define this as a displayed bicategory over the bicategory of univalent categories. One of the benefits of this approach will be in Section 4.3, where we will be able to use the fact that the bicategory of univalent categories is univalent, when showing that the bicategory of DMCs is also univalent. Another example will be in Section 5.1, where it will allow us to add terminal objects into the base category, without having to re-implement the bicategory.

To create the displayed bicategory over the bicategory of univalent categories, we will simply define the types required by Definition 26. Then, we will need to also need to define the displayed identity 1-cell and the composition of displayed 1-cells. Both of these will be a term witnessing that a theorem, which we already proved, holds.

Definition 37 (`disp_prebicat_1_id_comp_cells_display_map_cat`)**.** The displayed bicategory of DMCs over the bicategory of univalent categories consists of

- for each univalent category $\mathcal{C}$, the type of displayed 0-cells over it is the type of display map classes $\mathcal{D}$ over $\mathcal{C}$ (Definition 33);
- for each functor $F : \mathcal{C} \to \mathcal{C}'$, the type of displayed 1-cells $\bar{F}$ is the type witnessing the fact that F is a map of DMCs, that is, it preserves display maps, as well as, pullbacks of display maps (Definition 36);
- the identity 1-cell $\mathsf{id}_1\ \mathcal{D}$ is witnessing the fact that the identity functor $\mathbb{1}_{\mathcal{C}} : \mathcal{C} \to \mathcal{C}$ preserves both display maps and pullbacks (Lemma 7);
- composition of the 1-cells $\bar{F} \cdot \bar{G}$ is witnessing that composition of functors that preserve display maps and their pullbacks, also preserves display maps and their pullbacks (Lemma 8);
- for each natural transformation $\theta : F \Rightarrow G$, the type of displayed 2-cells $\bar{\theta}$ is simply the unit type, as we do not have any conditions for the transformations.

All other components (e. g. identity displayed 2-cell, composition of displayed 2-cells, etc.) will be trivial to define, since the type of displayed 2-cells is also the unit type. After formalising these proofs, we finally get `disp_bicat_display_map_cat` and the corresponding total category `bicat_display_map_cat`. This total bicategory will be equivalent to the bicategory of display map categories. We will forgo giving a formal proof of this, and give only a notion of why it holds. For this, check the data of both bicategories, presented in Table 4.1. All of the types are the exact same, except the transformations, but even

	displayed version	direct version
0-cells	$(\mathcal{C}, \mathcal{D}) : \sum_{(\mathcal{C}:\mathsf{category})} \mathsf{DMC}_{\mathcal{C}}$	$(\mathcal{C}, \mathcal{D}) : \sum_{(\mathcal{C}:\mathsf{category})} \mathsf{DMC}_{\mathcal{C}}$
1-cells	$(F, H) : \sum_{(F:\mathcal{C}\to\mathcal{C}')} \mathsf{is\_map}\ F$	$(F, H) : \sum_{(F:\mathcal{C}\to\mathcal{C}')} \mathsf{is\_map}\ F$
2-cells	$(\alpha, \star) : \sum_{(\alpha:F\Rightarrow G)} \mathbf{1}$	$\alpha : F \Rightarrow G$

Table 4.1: Comparison of the displayed and direct version

those are equivalent, thus equal by univalence. This equivalence simply follows, from that for any type A, the function $a \mapsto (a, \star) : A \to \sum_{a:A} \mathbf{1}$ is the inverse of the projection function $(a, \star) \mapsto a : \sum_{a:A} \mathbf{1} \to A$. The laws are propositions, and thus do not matter for this comparison, as they all hold in both constructions. The only other potential difference would be the identities and compositions, but in both bicategories these solely rely on the identities and compositions of functors and transformations, and thus should be equivalent.

Ideally, we could take advantage of the displayed construction when constructing the pseudofunctor into the bicategory of comprehension categories, by making it a displayed pseudofunctor above the identity pseudofunctor to and from the bicategory of univalent categories. We are not able to do this in the UniMath library, as the bicategory of comprehension categories is defined as a series of displayed bicategories, and illustration of this can be seen in Figure 5.2. Theoretically, it should still be possible to use other definitions from the UniMath library to construct the pseudofunctor, but as we will see in Section 4.4, constructing the pseudofunctor directly is quite doable, and it also illustrates the correspondence of the two structures better. Further, since the comprehension categories are defined to include a terminal object in the base category, we will also need to add a terminal object to the display map category. For more information about this, see Section 5.1. Here it should suffice to mention that this will not influence the construction of the pseudofunctor.

4.3 Univalence of the Bicategory

As stated in Section 3.3, when working in UF, it is quite natural to take (bi)categories to be *univalent*. To this end, we will in this section prove that the displayed bicategory we have in the previous section, is indeed univalent. From there, the UniMath library already has proofs that the total version will be univalent too, as it has already been implemented that the total bicategory is univalent if the displayed bicategory and the base are both univalent.

As presented in Definition 32, the univalence of a bicategory needs it to be univalent both *locally* and *globally*, the same is true for displayed bicategories. Both types of univalence involve showing that some function is an equivalence. The local univalence turns out to be trivial for our displayed bicategory.

Theorem 38 (`disp_univalent_2_1_disp_bicat_display_map_cat`)**.** *The displayed bicategory of display map categories is locally univalent.*

Proof. By [11, Proposition 7.5], we need to show that the following function is an equivalence

$$\mathsf{disp\_idtoiso}^{2,1}_{F,\bar{F},\bar{F}'} : \bar{F} =_F \bar{F}' \to \bar{F} \cong_{\mathsf{id}_2\ F} \bar{F}'.$$

To construct this equivalence, we can make use of the fact that we have shown $\bar{F}$ (see Definition 36) to be a *proposition*. That will in turn mean that the type $\bar{F} =_F \bar{F}'$ will just be the unit type.

For the other side, we have a displayed adjoint over the identity natural transformation. However, we know that this can only be the unit type as well, as that is how we defined transformations in Definition 37.

Trivially, the unit type is equivalent to the unit type. □

Global univalence, on the other hand, is unfortunately not so simple, and requires a more involved proof. The proof we will present will first give an intuition on why the function should be an equivalence. Then we will give a more detailed proof where we construct two smaller equivalences, and show that their composition is equal to the function we need to be an equivalence. This pattern of showing equivalences using composition is a common one in the UniMath library, which helped during the implementation, as I could find other similar examples.

Theorem 39 (`disp_univalent_2_0_disp_bicat_display_map_cat`)**.** *The displayed bicategory of display map categories is globally univalent.*

Proof. By [11, Proposition 7.5], we need to show that the following function is an equivalence

$$\mathsf{disp\_idtoiso}^{2,0}_{\mathcal{C},\mathcal{D},\mathcal{D}'} : \mathcal{D} =_{\mathcal{C}} \mathcal{D}' \to \mathcal{D} \simeq_{\mathsf{id}_1\ \mathcal{C}} \mathcal{D}'.$$

To get an intuition, let's first examine the two sides. For the left side, we know that $\mathcal{D}$ is comprised of a subset $\mathcal{D}$ of the set of morphisms in $\mathcal{C}$, and a proof that it contains pullbacks. Following Remark 35, this will mean that two of them are equal if their underlying subsets are equal. By univalence, those will be equal if they include the same elements.

The other side will be the type of adjoints between two display map categories, which is just two proofs of the identity being a map between $\mathcal{D}$ and $\mathcal{D}'$. This will include proofs that the identity keeps all display maps in both, necessarily meaning that the two contain the same elements. Thus both sides will depend on the subsets being equal, hence, the equality holds.

Now that we have an intuition, we can proceed with the more formal version. For this, we first construct another equivalence between $\mathcal{D} =_{\mathcal{C}} \mathcal{D}'$ and $\mathcal{D} \simeq_{\mathsf{id}_1\ \mathcal{C}} \mathcal{D}'$, then we show that it is functionally equivalent to $\mathsf{disp\_idtoiso}^{2,0}_{\mathcal{C},\mathcal{D},\mathcal{D}'}$.

The equivalence we construct is a composition of two equivalences

$$\mathsf{adj\_weq\_disp\_adj}_{\mathcal{D},\mathcal{D}'} : (\mathcal{D} \xrightarrow[\mathsf{id}_1\mathcal{C}]{} \mathcal{D}') \times (\mathcal{D}' \xrightarrow[\mathsf{id}_1\mathcal{C}]{} \mathcal{D}) \simeq (\mathcal{D} \simeq_{\mathsf{id}_1\mathcal{C}} \mathcal{D}');$$
$$\mathsf{DMC\_equiv\_weq\_adj}_{\mathcal{D},\mathcal{D}'} : \mathcal{D} =_{\mathcal{C}} \mathcal{D}' \simeq (\mathcal{D} \xrightarrow[\mathsf{id}_1\mathcal{C}]{} \mathcal{D}') \times (\mathcal{D}' \xrightarrow[\mathsf{id}_1\mathcal{C}]{} \mathcal{D}).$$

The former is just an expansion of what it means to have an adjoint between two display map categories, as

$$\mathsf{id}_1\mathcal{C} \cdot \mathsf{id}_1\mathcal{C} = \mathsf{id}_1\mathcal{C}$$

by the fact that it is the identity, and so if it is a map in both directions, then it is an adjoint between $\mathcal{D}$ and $\mathcal{D}'$.

The latter is an equivalence, since if and only if $\mathsf{id}_1\ \mathcal{C}$ is a map in both directions, then, due to the preservation of maps, $\mathcal{D}$ and $\mathcal{D}'$ contain the same display maps, meaning that they are equal, using univalence.

Finally, we need to show that the underlying function of this equivalence and $\mathsf{disp\_idtoiso}$ are point-wise equal. This can be done using *path-induction*, and the fact that being a map of display maps is a predicate (`isPredicate_is_display_map_class_functor`). □

Remark 40. In the formalised proof, there is also an added helper equivalence between the display map category $(\mathcal{C},\mathcal{D})$ and the display map class $\mathcal{D}$

$$\mathsf{data\_weq} : (\mathcal{C},\mathcal{D}) = (\mathcal{C}',\mathcal{D}') \simeq \mathcal{D} = \mathcal{D}'.$$

This is trivial to create as a display map class $\mathcal{D}$ implicitly presumes some category C from which it selects its display maps.

We have shown that the displayed bicategory of display map categories is both locally univalent, as well as globally univalent. This finally allows us to put the two together, and prove that this displayed bicategory is univalent.

Theorem 41 (`disp_univalent_2_disp_bicat_display_map_cat`)**.** The displayed bicategory of display map categories is *univalent.*

Proof. A displayed bicategory is univalent, if it is both *locally* and *globally* univalent. For the first, we use Theorem 38. For the second, we use Theorem 39. □

Using the above proof, as well as the univalence of the bicategory of univalent categories, we can show that the total version is also univalent (`univalent_2_bicat_display_map_ _cat`), using already formalised proofs for displayed categories. As mentioned at the end of the previous section, in the implementation we also require the base category to have a terminal object. This luckily does not affect the univalence of the bicategory, and since we used displayed bicategories as we will present in Section 5.1, we can use proofs from the UniMath library to easily show this fact. This will be also presented in Theorem 55.

4.4 Constructing the Pseudofunctor

At the end of Section 4.1, we have stated that we can directly map a display map category to a comprehension category, and we can also map their related concepts of maps and transformations. But the correspondence between DMCs and CCs goes even further, for example, the functions mentioned above preserve identities and composition, among other things. So, following the paper by Ahrens, Lumsdaine and North [1], since we have the bicategory of both display map categories and comprehension categories, we can organise the above functions and their properties into a pseudofunctor.

To construct the pseudofunctor, first we will need to give the underlying functions, mapping 0-, 1- and 2-cells, these are essentially the same functions as the ones mentioned above. We will construct the function from the *0-cells* of the former to the *0-cells* of the latter (Section 4.4.1); then construct the function from the *1-cells* of the former to the *1-cells* of the latter (Section 4.4.2); and finally construct the function from the *2-cells* of the former to the *2-cells* of the latter (Section 4.4.3). The fact that I have already defined the direct functions from display map categories to comprehension categories, and their related concepts, helped tremendously in implementing these three functions, even though I couldn't use them as-is.

Finally, to complete the pseudofunctor, we will need to show that the aforementioned properties, required by the definition, do hold for the three functions we will define. This is split into two parts, as the preservation of identity 1-cells and preservation of composition of 1-cells need only to be preserved up to an invertible 2-cell. Thus, in Section 4.4.4, we show that both of the desired invertible 2-cells will be the identity 2-cell. Lastly, Section 4.4.5 will give the proofs of all the laws involving 2-cells. Both of these sections are here for completeness, and have largely trivial proofs, this is in an indirect way due to univalence, as it allows equivalences (such as invertible 2-cells) to be turned into equalities.

4.4.1 Function on 0-cells

First thing to define is the function mapping display map categories to comprehension categories. The input to this function will be a pair $(\mathcal{C}, \mathcal{D})$, where $\mathcal{C}$ is a univalent category, and $\mathcal{D}$ is the displayed version of the display map category above $\mathcal{C}$. The output of the function will need to be a comprehension category $(\mathcal{C}, p, \chi)$, consisting of a univalent category $\mathcal{C}$, a category of types $\mathcal{T}$ over $\mathcal{C}$ with a fibration p, and finally a comprehension χ from $\mathcal{T}$ into the arrow category $\mathcal{C}^{\to}$. This whole section corresponds to `bicat_terminal_display_map_cat_to_bicat_full_comp_cat` in the implementation. A reader might notice that the implementation also includes a terminal object in the base $\mathcal{C}$, this is due to how the bicategory of full comprehension categories is implemented in UniMath, but this addition has no effect on the correspondence, and it didn't require any extra effort to add these. For this reason, we present the non-terminal version in this section, for readability.

To construct the comprehension category given a display map category, first, we can take the base of the display map category $\mathcal{C}$ as the base of the comprehension category. Then, we need to give a fibration above this category. Since we know the connection between cartesian arrows and pullbacks in the case of the codomain functor (see Section 2.2.1), and we know that the display map category requires pullbacks of display maps, the functor mapping display maps to their codomain will be a fibration. This will mean that the category of types $\mathcal{T}$ will be the arrow version of the display map category $\mathcal{D}^{\rightarrowtriangle}$.

Theorem 42. Given a display map category $(\mathcal{C}, \mathcal{D})$, the triple $(\mathcal{C}, \iota \cdot \mathsf{cod} : \mathcal{D}^{\rightarrowtriangle} \to \mathcal{C}, \iota)$ is a comprehension category.

Proof. First, let us make a diagram

$$\begin{tikzcd}
\mathcal{D}^{\rightarrowtriangle} \arrow[rr, hook, "\iota"] \arrow[dr, "\iota\cdot\mathsf{cod}"'] & & \mathcal{C}^{\to} \arrow[dl, "\mathsf{cod}"] \\
& \mathcal{C} &
\end{tikzcd}$$

For this to be a comprehension category, we need to show that

1. $\mathcal{D}^{\rightarrowtriangle}$ is a univalent displayed category over $\mathcal{C}$;

2. $\iota \cdot \mathsf{cod} : \mathcal{D}^{\rightarrowtriangle} \to \mathcal{C}$ is a cleaving;

3. ι preserves cartesian arrows.

All three of these facts follow from the fact that $\mathcal{D}^{\rightarrowtriangle}$ is a subcategory of $\mathcal{C}^{\rightarrow}$ (see Definition 10), and that ι is just the inclusion functor into $\mathcal{C}^{\rightarrow}$ (see Definition 11). The fact that $\iota \cdot \mathsf{cod}$ is a cleaving can be shown using the fact that $\mathcal{D}$ has pullbacks, and that pullbacks correspond to cartesian arrows in the case of the cod functor. $\square$

This mapping does indeed reflect how the two sctructures are used as a semantic framework. For both, the *contexts* are objects in the base $\mathcal{C}$, which we keep directly. In the case of DMC, *types* are the display maps in $\mathcal{D}$, while in the case of CC, they must be objects in category $\mathcal{T}$. In this function, we set this category $\mathcal{T}$ to be the arrow version of $\mathcal{D}$, whose objects are indeed the display maps. The comprehension being the inclusion functor means that the projection of the types (display maps) will be themselves, meaning that they will retain all of their secions, and therefore retain their terms. Finally, context morphisms remain morphisms in $\mathcal{C}$, with their pullbacks becoming cartesian arrows. Therefore, we can be confident that we have managed to formalise this correspondence.

We can even go a step further, and show that this is no ordinary comprehension category, but it is also a *full* one. Being *full* means that the comprehension is *fully faithful*, which follows from the fact that we used an inclusion functor as the comprehension.

Theorem 43. The comprehension category presented in Theorem 42 is *full*.

Proof. A comprehension category being *full*, means that its comprehension functor (in this case ι) is *fully faithful*. Being *fully faithful* means that for any pair of objects $d, d' : \mathcal{D}^{\rightarrowtriangle}$, the following function is a bijection

$$\iota_m : \mathcal{D}^{\rightarrowtriangle}[\![d, d']\!] \to \mathcal{C}^{\rightarrow}[\![\iota\ d, \iota\ d']\!].$$

This follows from the fact that, since ι is an inclusion functor (see Definition 11), all its underlying functions are just the identity function (with a restricted domain). $\square$

As this thesis only presents the correspondence in one direction, this fact is not too useful. However, it is a step on the way to constructing the backwards direction. That would also require further refinement on the kind of comprehension category that a display map category corresponds to, before the inverse direction could be defined and the two could be put together into an equivalence. This will also be discussed as potential future work in Section 7.1.

4.4.2 Function on 1-cells

The second function that makes up the pseudofunctor needs to map 1-cells. In the case of display map category, the 1-cells are the maps of display map categories (see Definition 6), which consist of a functor and a proof that the functor satisfies certain properties. For comprehension categories, the 1-cells are pseudo maps (see Definition 21), which consist of two functors and a natural isomorphism. This section shows the proofs that correspond to `terminal_display_map_functor_to_full_comp_cat_functor` in the implementation.

Theorem 44. Given a map F between two DMCs $(\mathcal{C}, \mathcal{D}), (\mathcal{C}', \mathcal{D}')$, there is a corresponding pseudo map $(F, F^{\rightarrowtriangle})$ between the corresponding comprehension categories. Here, $F^{\rightarrowtriangle} : \mathcal{D}^{\rightarrowtriangle} \to \mathcal{D}'^{\rightarrowtriangle}$ is defined as in Definition 12.

Proof. For clarity, let us first make a diagram of the proposed pseudo map

$$\begin{tikzcd}
\mathcal{D}^{\rightarrowtriangle} \arrow[rr, "\iota"] \arrow[drrr, "F^{\rightarrowtriangle}"'] \arrow[ddr, "\rhd"] & & \mathcal{C}^{\rightarrow} \arrow[drrr, "F^{\rightarrow}"] \arrow[dd, phantom, "\overset{\varphi}{\Longrightarrow}"] \arrow[ddl] & & & \\
& & & \mathcal{D}'^{\rightarrowtriangle} \arrow[rr, "\iota'"] \arrow[ddr, "\rhd"] & & \mathcal{C}'^{\rightarrow} \arrow[ddl] \\
& \mathcal{C} \arrow[drrr, "F"] & & & & \\
& & & & \mathcal{C}' &
\end{tikzcd}$$

To make this a pseudo map, we need $F^{\rightarrowtriangle}$ to be cartesian; and we need to construct an isomorphism

$$\varphi : \iota \cdot F^{\rightarrow} \Rightarrow F^{\rightarrowtriangle} \cdot \iota'.$$

The former statement follows from the fact that F preserves the pullbacks of display maps, as both in $\mathcal{D}^{\rightarrowtriangle}$ and $\mathcal{D}'^{\rightarrowtriangle}$ the cartesian arrows correspond to pullbacks.

The isomorphism will be the identity isomorphism, as both ι and ι' are just inclusion functors, and $F^{\rightarrowtriangle} : \mathcal{D}^{\rightarrowtriangle} \to \mathcal{D}'^{\rightarrowtriangle}$ and $F^{\rightarrow} : \mathcal{C}^{\rightarrow} \to \mathcal{C}'^{\rightarrow}$ are constructed similarly over F. □

While it has not been used in the implementation, it is worth to mention that since we not only had a natural isomorphism, but also an equality

$$\iota \cdot F^{\rightarrow} = F^{\rightarrowtriangle} \cdot \iota'.$$

This makes the pseudo map also a *strict map* (see Remark 22). This, similarly to Theorem 43, is a step toward specifying the correspondence better, in order to eventually construct the equivalence.

4.4.3 Function on 2-cells

The pseudofunctor will also need to map *2-cells*, that is, it needs to map natural transformations to transformations between pseudo maps. The result presented here corresponds to `terminal_display_map_transformation_to_full_comp_cat_transformation` in the implementation. This is the least interesting of the three functions of the pseudofunctor, as display map categories have no conditions on transformations, and the needed equality holds trivially. We follow the same pattern of making use of the arrow versions of the concepts related to DMC, and then using the similarity of these definitions to the arrow category and its related concepts to show that they indeed form a transformation between the pseudo maps.

Theorem 45. Given a natural transformation θ between maps F, F' of DMCs, we have a corresponding transformation $(\theta, \theta^{\rightarrowtriangle})$ between the corresponding pseudo maps $(F, F^{\rightarrowtriangle}, \varphi)$ and $(F', F'^{\rightarrowtriangle}, \varphi')$. In this definition $\theta^{\rightarrowtriangle}$ is using Definition 13.

Proof. First, let us make a diagram.

$$\begin{tikzcd}[column sep=huge, row sep=huge]
\mathcal{D}^{\rightarrowtriangle} \arrow[r, "\iota"] \arrow[d, bend left=40, "F'^{\rightarrowtriangle}"] \arrow[d, bend right=40, "F^{\rightarrowtriangle}"'] \arrow[d, phantom, "\overset{\theta^{\rightarrowtriangle}}{\Uparrow}"] & \mathcal{C}^{\rightarrow} \arrow[d, bend left=40, "F'^{\rightarrow}"] \arrow[d, bend right=40, "F^{\rightarrow}"'] \arrow[d, phantom, "\overset{\theta^{\rightarrow}}{\Uparrow}"] \\
\mathcal{D}'^{\rightarrowtriangle} \arrow[r, "\iota'"'] & \mathcal{C}'^{\rightarrow}
\arrow[from=1-1, to=2-2, phantom, "=\varphi' \Longrightarrow", near start] \arrow[from=1-1, to=2-2, phantom, "=\varphi \Longrightarrow", near end]
\end{tikzcd}$$

To show that our candidate is indeed a transformation between the pseudo maps, we need to show that

$$\varphi \cdot \theta^{\rightarrow} = \theta^{\rightarrowtriangle} \cdot \varphi'.$$

Since φ and φ' are both identity isomorphisms, and $\theta^{\rightarrow}$ and $\theta^{\rightarrowtriangle}$ are defined using the same structure just with different (co)domains, this equation will hold. □

4.4.4 Preservation of Identity 1-cells and Composition

For the three functions defined in the prior sections to be a pseudofunctor, they must preserve operators. The following theorems are proven in two parts in the implementation, one gives the *2-cell*, and the other shows that it is *invertible*, these are

- `bicat_terminal_display_map_to_bicat_full_comp_cat_id_1cell`;
- `psfunctor_id_is_invertible_bicat_terminal_display_map_cat_to_bicat_full_comp_cat`;
- `bicat_terminal_display_map_to_bicat_full_comp_cat_comp_1cell`;
- `psfunctor_comp_is_invertible_bicat_terminal_display_map_cat_to_bicat_full_comp_cat`.

In the following, we will need to refer to the current pseudofunctor that we are defining. For this, we will use the name DMC_to_CC. We will use subscripts to refer to the functions: 0 for the 0-cell function; 1 for the 1-cell function and 2 for the 2-cell function. Both of the invertible 2-cells, which we need to show exist, will turn out to be the identity 2-cell.

The first invertible 2-cell we need to give, corresponds to the preservation of identity 1-cells, up to the 2-cell isomorphism, we will give. However, since the pseudofunctor which we are constructing preserves identity 1-cells on the nose, giving this invertible 2-cell is trivial.

Lemma 46. We need to show that the following *2-cell* exists, and is *invertible*.

$$\mathsf{DMC\_to\_CC}_{\mathsf{id}}(\mathcal{D}) : \mathsf{DMC\_to\_CC}_1(\mathsf{id}_1\ \mathcal{D}) \Rightarrow \mathsf{id}_1\ \mathsf{DMC\_to\_CC}_0(\mathcal{D})$$

Proof. We know that $\mathsf{id}_1\ \mathcal{D}$ is just the identity functor $\mathbb{1}_{\mathcal{C}}$. The corresponding pseudo map therefore is $(\mathbb{1}_{\mathcal{C}}, \mathbb{1}_{\mathcal{C}}^{\rightarrowtriangle})$. Similarly, $\mathsf{id}_1\ \mathsf{DMC\_to\_CC}_0(\mathcal{D})$ is also $(\mathbb{1}_{\mathcal{C}}, \mathbb{1}_{\mathcal{C}'}^{\rightarrowtriangle})$. Therefore, the *invertible 2-cell* we are looking for will just be the *identity 2-cell* in the bicategory of full CCs. ◻

The second invertible 2-cell corresponds to the preservation of the composition of 1-cells. Since our pseudofunctor preserves composition not only up to isomorphism, but directly, giving the invertible 2-cell is once more trivial.

Lemma 47. We need to show that the following *2-cell* exists, and is *invertible*.

$$\mathsf{DMC\_to\_CC}_{\mathsf{comp}}(F, G) : \mathsf{DMC\_to\_CC}_1(F) \cdot \mathsf{DMC\_to\_CC}_1(G) \Rightarrow \mathsf{DMC\_to\_CC}_1(F \cdot G)$$

Proof. The left hand side of the 2-cell will be

$$\begin{aligned}
\mathsf{DMC\_to\_CC}_1(F) \cdot \mathsf{DMC\_to\_CC}_1(G) &= (F, F^{\rightarrowtriangle}, \mathbb{1}) \cdot (G, G^{\rightarrowtriangle}, \mathbb{1}) && \text{(Theorem 44)} \\
&= (F \cdot G, F^{\rightarrowtriangle} \cdot G^{\rightarrowtriangle}, \mathbb{1}) && \textit{composition of pseudomaps}
\end{aligned}$$

Meanwhile the right side will be

$$\mathsf{DMC\_to\_CC}_1(F \cdot G) = (F \cdot G, (F \cdot G)^{\rightarrowtriangle}, \mathbb{1}).$$

But, from Definition 12, we can easily show that $F^{\rightarrowtriangle} \cdot G^{\rightarrowtriangle} = (F \cdot G)^{\rightarrowtriangle}$. Thus, the two sides are equal, therefore the invertible 2-cell we need will be the *identity 2-cell* in the bicategory of full CCs. ◻

The fact that both of these 2-cells are just the identity also simplifies the proofs in the following section. This is because the laws for 2-cells rely on the usage of these two 2-cells, and in our case this usage can be simplified away using identity laws for the 2-cells.

4.4.5 Preservation Laws

Finally, in order to fully be a pseudofunctor, a set of laws must hold. In this section, we show that all of those laws do indeed hold for our pseudofunctor. All of these laws state that the pseudofunctor must preserve certain structures. The corresponding definitions in the implementation are

- Lemma 48 — `psfunctor_id2_law_bicat_terminal_display_map_cat_to_bicat_full_comp_cat`;
- Lemma 49 — `psfunctor_vcomp2_law_bicat_terminal_display_map_cat_to_bicat_full_comp_cat`;
- Lemma 50 — `psfunctor_lunitor_law_bicat_terminal_display_map_cat_to_bicat_full_comp_cat`;
- Lemma 51 — `psfunctor_runitor_law_bicat_terminal_display_map_cat_to_bicat_full_comp_cat`;
- Lemma 52 — `psfunctor_lassociator_law_bicat_terminal_display_map_cat_to_bicat_full_comp_cat`;
- Lemma 53 — `psfunctor_lwhisker_law_bicat_terminal_display_map_cat_to_bicat_full_comp_cat`;
- Lemma 54 — `psfunctor_rwhisker_law_bicat_terminal_display_map_cat_to_bicat_full_comp_cat`.

All of these proofs boil down to a trivial equality, and are included only for completeness. What we must show in each case, that some composition of 2-cells equals some other 2-cell composition, as we will see, most of the proofs will show that both sides are an identity 2-cell, and thus are equal.

Lemma 48 (*preservation of identity 2-cells*)**.** Given two DMCs $\mathcal{D}, \mathcal{D}$, and a map F between them, the following holds

$$\mathsf{DMC\_to\_CC}_2(\mathsf{id}_2\ F) = \mathsf{id}_2(\mathsf{DMC\_to\_CC}_1(F)).$$

Proof. The left hand side, by definition, is just the identity natural transformation, and its lifted version $(\mathsf{id}_F^{\rightarrowtriangle} : F^{\rightarrowtriangle} \Rightarrow F^{\rightarrowtriangle})$. Meanwhile, the right hand side is just $(\mathsf{id}_F, \mathsf{id}_{F^{\rightarrowtriangle}})$. Thus, we only need to show that $\mathsf{id}_F^{\rightarrowtriangle} = \mathsf{id}_{F^{\rightarrowtriangle}}$. This follows from Definitions 12 and 13. □

Lemma 49 (*preservation of vertical composition*)**.** Given two DMCs $\mathcal{D}, \mathcal{D}'$, three maps between them F, G, H, and natural transformations between those $\varphi : F \Rightarrow G$ and $\gamma : G \Rightarrow H$, the following holds

$$\mathsf{DMC\_to\_CC}_2(\varphi \bullet \gamma) = \mathsf{DMC\_to\_CC}_2(\varphi) \bullet \mathsf{DMC\_to\_CC}_2(\gamma).$$

Proof. The left hand side, by definition, is $(\varphi \bullet \gamma, (\varphi \bullet \gamma)^{\rightarrowtriangle})$. Meanwhile, the right hand side is

$$\mathsf{DMC\_to\_CC}_2(\varphi) \bullet \mathsf{DMC\_to\_CC}_2(\gamma) = (\varphi, \varphi^{\rightarrowtriangle}) \bullet (\gamma, \gamma^{\rightarrowtriangle}) = (\varphi \bullet \gamma, \varphi^{\rightarrowtriangle} \bullet \gamma^{\rightarrowtriangle}).$$

This means we only need to show that $(\varphi \bullet \gamma)^{\rightarrowtriangle} = \varphi^{\rightarrowtriangle} \bullet \gamma^{\rightarrowtriangle}$. This can be easily done using the definition of the lifting of natural transformations between maps of DMCs (as given in Definition 13). □

Lemma 50 (*preservation of the left-unitor*)**.** Given two DMCs $\mathcal{D}, \mathcal{D}'$, and a map F between the, the following holds

$$\begin{aligned}\lambda(\mathsf{DMC\_to\_CC}_1(F)) = (&\mathsf{DMC\_to\_CC}_{\mathsf{id}}\ (\mathcal{D}) \triangleright \mathsf{DMC\_to\_CC}_1(F)) \bullet \\ &\bullet \mathsf{DMC\_to\_CC}_{\mathsf{comp}}(\mathsf{id}_1\ \mathcal{D}, F) \bullet \\ &\bullet \mathsf{DMC\_to\_CC}_2\ (\lambda(F)).\end{aligned}$$

Proof. The left side $\lambda(\mathsf{DMC\_to\_CC}_1(F))$ is just the *identity 2-cell*, as for any *pseudo map* P, it holds that $P = \mathbb{1} \cdot P$. Similarly, on the right side $\lambda(F)$ is also an *identity 2-cell*, which, by Lemma 48, will be mapped to the *identity 2-cell* $\mathsf{id}_2(\mathsf{DMC\_to\_CC}_1(F))$.

From Lemmas 46 and 47 respectively, we know that $\mathsf{DMC\_to\_CC}_{\mathsf{comp}}$ and $\mathsf{DMC\_to\_CC}_{\mathsf{id}}$ are both also an *identity 2-cell*, the latter of which will stay an identity when whiskered.

Thus, both sides are the *identity 2-cell* $\mathsf{id}_2(\mathsf{DMC\_to\_CC}_1(F))$, and are therefore equal. □

Lemma 51 (*preservation of the right-unitor*)**.** Given two DMCs $\mathcal{D}, \mathcal{D}'$, and a map F between the, the following holds

$$\begin{aligned}\rho(\mathsf{DMC\_to\_CC}_1\ F) = {} & (\mathsf{DMC\_to\_CC}_1 F \lhd \mathsf{DMC\_to\_CC}_{\mathsf{id}}\ \mathcal{D}') \bullet \\ & \bullet \mathsf{DMC\_to\_CC}_{\mathsf{comp}}(f, \mathsf{id}_1 \mathcal{D}') \bullet \\ & \bullet \mathsf{DMC\_to\_CC}_2\ (\rho\ F).\end{aligned}$$

Proof. The left side $\rho(\mathsf{DMC\_to\_CC}_1(F))$ is just the *identity 2-cell*, as for any *pseudo map* P, it holds that $P = P \cdot \mathbb{1}$. Similarly, on the right side $\rho(F)$ is also an *identity 2-cell*, which, by Lemma 48, will be mapped to the *identity 2-cell* $\mathsf{id}_2(\mathsf{DMC\_to\_CC}_1(F))$.

From Lemmas 46 and 47 respectively, we know that $\mathsf{DMC\_to\_CC}_{\mathsf{comp}}$ and $\mathsf{DMC\_to\_CC}_{\mathsf{id}}$ are both also an *identity 2-cell*, the latter of which will stay an identity when whiskered.

Thus, both sides are the *identity 2-cell* $\mathsf{id}_2(\mathsf{DMC\_to\_CC}_1(F))$, and are therefore equal. □

Lemma 52 (*preservation of the associator*)**.** Given four DMCs $\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_3, \mathcal{D}_4$, and maps $F : \mathcal{D}_1 \to \mathcal{D}_2, G : \mathcal{D}_2 \to \mathcal{D}_3$ and $H : \mathcal{D}_3 \to \mathcal{D}_4$ between them, the following holds

$$\begin{gathered}(\mathsf{DMC\_to\_CC}_1 F \lhd \mathsf{DMC\_to\_CC}_{\mathsf{comp}}(G, H)) \bullet \mathsf{DMC\_to\_CC}_{\mathsf{comp}}(F, G \cdot H) \bullet \\ \bullet \mathsf{DMC\_to\_CC}_2(\alpha(F, G, H)) \\ = \\ \alpha(\mathsf{DMC\_to\_CC}_1(F), \mathsf{DMC\_to\_CC}_1(G), \mathsf{DMC\_to\_CC}_1(H)) \bullet \\ \bullet (\mathsf{DMC\_to\_CC}_{\mathsf{comp}}(F, G) \rhd \mathsf{DMC\_to\_CC}_1(H)) \bullet \mathsf{DMC\_to\_CC}_{\mathsf{comp}}(F \cdot G, H).\end{gathered}$$

Proof. We can simplify the left side by doing

$$\begin{aligned}\mathsf{DMC\_to\_CC}_1 F \lhd \mathsf{DMC\_to\_CC}_{\mathsf{comp}}(F, G) &= \mathsf{DMC\_to\_CC}_1 F \lhd \mathsf{id}_2 && \text{(Lemma 47)}\\ &= \mathsf{id}_2 \\ \mathsf{DMC\_to\_CC}_{\mathsf{comp}}(F, G \cdot H) &= \mathsf{id}_2 && \text{(Lemma 47)}\\ \mathsf{DMC\_to\_CC}_2(\alpha(F, G, H)) &= \mathsf{DMC\_to\_CC}_2 \mathsf{id}_2 && \text{(functor associativity)}\\ &= \mathsf{id}_2 && \text{(Lemma 48)}\\ \hline lhs. &= \mathsf{id}_2 \bullet \mathsf{id}_2 \bullet \mathsf{id}_2 \\ &= \mathsf{id}_2. && \text{(identity laws)}\end{aligned}$$

Similarly, we can simplify the right side by doing

$$\begin{aligned}\alpha(\mathsf{DMC\_to\_CC}_1(F), \mathsf{DMC\_to\_CC}_1(G), \mathsf{DMC\_to\_CC}_1(H)) &= \mathsf{id}_2 && \text{(pseudo map associativity)}\\ \mathsf{DMC\_to\_CC}_{\mathsf{comp}}(F, G) \rhd \mathsf{DMC\_to\_CC}_1(H) &= \mathsf{id}_2 \rhd \mathsf{DMC\_to\_CC}_1(H) && \text{(Lemma 47)}\\ &= \mathsf{id}_2 \\ \mathsf{DMC\_to\_CC}_{\mathsf{comp}}(F \cdot G, H) &= \mathsf{id}_2 && \text{(Lemma 47)}\\ \hline rhs. &= \mathsf{id}_2 \bullet \mathsf{id}_2 \bullet \mathsf{id}_2 \\ &= \mathsf{id}_2. && \text{(identity laws)}\end{aligned}$$

From this, it is trivial to show that $\mathsf{id}_2 = \mathsf{id}_2$. □

Lemma 53 (*preservation of the left-whiskering*)**.** Given three DMCs $\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_3$, and maps $F : \mathcal{D}_1 \to \mathcal{D}_2, G, H : \mathcal{D}_2 \to \mathcal{D}_3$ between them, the following holds

$$\mathsf{DMC\_to\_CC}_{\mathsf{comp}}(F, G) \bullet \mathsf{DMC\_to\_CC}_2(F \lhd \varphi) \\ = \\ (\mathsf{DMC\_to\_CC}_1(F) \lhd \mathsf{DMC\_to\_CC}_2(\varphi)) \bullet \mathsf{DMC\_to\_CC}_{\mathsf{comp}}(F, H).$$

Proof. On both sides we do have $\mathsf{DMC\_to\_CC}_{\mathsf{comp}}$, which simplifies to an identity by Lemma 47, leaving us with

$$\mathsf{DMC\_to\_CC}_2(F \lhd \varphi) = \mathsf{DMC\_to\_CC}_1(F) \lhd \mathsf{DMC\_to\_CC}_2(\varphi).$$

This will hold since both bicategories have $\lhd$ defined using the whiskering of natural transformations and functors. Further, DMC_to_CC maps natural transformations to the transformation consisting of the base, and the lifted version of them, the latter of which defines its whiskering in terms of the prior. □

Lemma 54 (*preservation of the right-whiskering*)**.** Given three DMCs $\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_3$, and maps $F, G : \mathcal{D}_1 \to \mathcal{D}_2, H : \mathcal{D}_2 \to \mathcal{D}_3$ between them, the following holds

$$\mathsf{DMC\_to\_CC}_{\mathsf{comp}}(F, H) \bullet \mathsf{DMC\_to\_CC}_2(\varphi \rhd H) \\ = \\ (\mathsf{DMC\_to\_CC}_2(\varphi) \rhd \mathsf{DMC\_to\_CC}_1(H)) \bullet \mathsf{DMC\_to\_CC}_{\mathsf{comp}}(G, H).$$

Proof. On both sides we do have $\mathsf{DMC\_to\_CC}_{\mathsf{comp}}$, which simplifies to an identity by Lemma 47, leaving us with

$$\mathsf{DMC\_to\_CC}_2(\varphi \rhd F) = \mathsf{DMC\_to\_CC}_2(\varphi) \rhd \mathsf{DMC\_to\_CC}_1(F).$$

This will hold since both bicategories have $\rhd$ defined using the whiskering of natural transformations and functors. Further, DMC_to_CC maps natural transformations to the transformation consisting of the base, and the lifted version of them, the latter of which defines its whiskering in terms of the prior. □

With that done, our pseudofunctor is complete. In the implementation the final pseudofunctor can be found as `psfunctor_bicat_terminal_display_map_cat_to_bicat_full_comp_cat`.

Chapter 5

Implementation in the UniMath Library

As mentioned previously, results presented in this thesis have also been verified using a *proof assistant*, namely Rocq[1]. A *proof assistant* is a computer program that can verify that a specification given to it is sound. My work was done using the UniMath library[2], which is a library implementing various parts of mathematics formalised using Univalent Foundations. My contributions resulted in three pull requests adding to the library, these pull requests (in order) contained the following

- the first pull request contained a definition of display map categories and a direct conversion to comprehension categories
- the second pull request contained a direct definition of the bicategory of DMCs, and a pseudofunctor into the bicategory of full CCs
- the third pull request contained a displayed definition of the bicategory; definition of its univalence and a new pseudofunctor from it to the bicategory of full CCs

The main contributions from these pull requests are concentrated in three files, containing a combined 722 lines of specification (definitions, theorems, lemmas, etc.); and 919 lines of proof script, as reported by `coqwc` (a tool included in Rocq). The distribution of these lines can be seen in Table 5.1. While I also made some small contributions to other files as well, these files are where the majority of my contributions are.

spec	proof	file name
414	345	`DisplayMapCat.v`
111	268	`BicatOfDisplayMapCat.v`
197	306	`DispBicatOfDisplayMapCat.v`

Table 5.1: Contributions per file

A number of the definitions that I implemented, ended up being (dependent) pairs. This is also the case for many other definitions implemented in the UniMath library, for example, *univalent categories* are formalised as

$$\mathsf{univalent\_category} := \sum_{\mathcal{C}:\mathsf{category}} \mathsf{is\_univalent}\ \mathcal{C},$$

[1]Rocq home page
[2]UniMath repository

and in fact category is also a dependent pair

$$\mathsf{category} := \sum_{\mathcal{C}:\mathsf{precategory}} \mathsf{has\_homsets}\ \mathcal{C}.$$

To help deal with such types, and not have to litter the codebase with `pr1` and `pr2` (the projection functions out of a pair), the UniMath library often defines `Coercion`-s for these kinds of types. A `Coercion` allows Rocq to do the conversion between types automatically, allowing the code to be more readable. So, when we need something of type precategory, we can give it an element of univalent_category instead.

One possible downside of using these is that it can hide away details in Rocq's proof tab. I did run into this a few times, when trying to copy a type from the goal-tab that would not type check. Usually this could be resolved by asking Rocq to display all coercions, and copying this instead. I am not fully sure of why this happens, but usually adding only one of the coercions as an explicit function fixed this problem.

This chapter will discuss some of the implementation details that went into writing this code. First, as mentioned at the end of Section 4.2, the implementation also adds terminal objects to the base category. I will discuss the reasons for this, together with the architecture of the displayed categories in Section 5.1. Second, the thesis has used both the arrow category $C^{\rightarrow}$ and the arrow version of display map categories $\mathcal{D}^{\twoheadrightarrow}$, therefore, Section 5.2 will briefly discuss the version of these that is used in the implementation. Finally, I will end this chapter with the topic of optimisation, which is unique to computer proofs. Section 5.3 will give the rationale for optimising the proofs, as well as, some of the steps I took to optimise my code.

5.1 Construction of Displayed Bicategories

This thesis has been discussing all DMCs and all CCs, which has one crucial difference to the implementation, namely that the implementation of the pseudofunctor instead relies on display map categories *with a terminal object in the base category*. The reason for this was that the bicategory of CCs was already implemented using a base category that has a terminal object. As this addition was done at a low level (see Figure 5.2), removing it would have required an almost full copy of multiple files. For this reason, I have opted to just add this condition to the implementation. This change should not affect any other part of the implementation in a substantial way. It is worth to mention that adding this terminal object is included in some definitions of DMC and CC, when presenting them for the use of modelling type theory. This is due to the fact that the terminal object will correspond to the empty context.

The implementation of this change was done in a modular manner, thanks to *displayed bicategories*. The displayed bicategory of univalent categories with terminal objects was already defined as `disp_bicat_terminal_obj`. To begin with, I have defined the displayed bicategory of DMCs following the formalisation in Section 4.2. Then, as given by Definition 28, we can create the product of them, corresponding to the displayed bicategory of DMCs with a terminal object in the base. An illustration of this process can be seen in Figure 5.3. This way of construction should allow future contributors to be able to freely use either version, it is also not too dissimilar to certain notions of inheritance in other programming languages.

Since this is a new bicategory that we constructed, we should show that it is univalent. As mentioned in Section 3.3, the UniMath library already has proofs that the product of two displayed bicategories, as defined in Definition 28, will be univalent if three things hold. These three are: that the base is univalent, and that both displayed bicategories are univalent. We have shown that the displayed bicategory of display map categories

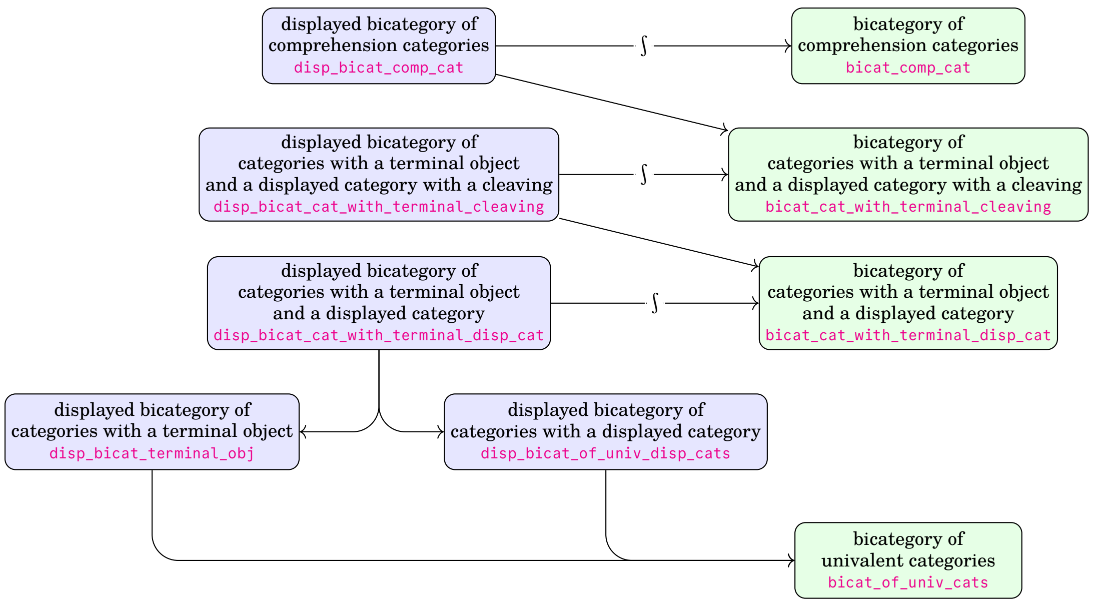
displayed bicategory of comprehension categories
disp_bicat_comp_cat
bicategory of comprehension categories
bicat_comp_cat
displayed bicategory of categories with a terminal object and a displayed category with a cleaving
disp_bicat_cat_with_terminal_cleaving
bicategory of categories with a terminal object and a displayed category with a cleaving
bicat_cat_with_terminal_cleaving
displayed bicategory of categories with a terminal object and a displayed category
disp_bicat_cat_with_terminal_disp_cat
bicategory of categories with a terminal object and a displayed category
bicat_cat_with_terminal_disp_cat
displayed bicategory of categories with a terminal object
disp_bicat_terminal_obj
displayed bicategory of categories with a displayed category
disp_bicat_of_univ_disp_cats
bicategory of univalent categories
bicat_of_univ_cats

Figure 5.2: Construction of the bicategory of CCs

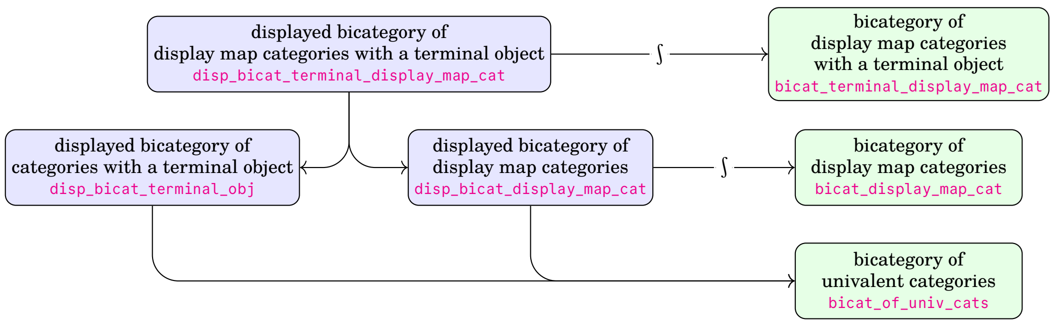
displayed bicategory of display map categories with a terminal object
disp_bicat_terminal_display_map_cat
bicategory of display map categories with a terminal object
bicat_terminal_display_map_cat
displayed bicategory of categories with a terminal object
disp_bicat_terminal_obj
displayed bicategory of display map categories
disp_bicat_display_map_cat
bicategory of display map categories
bicat_display_map_cat
bicategory of univalent categories
bicat_of_univ_cats

Figure 5.3: Construction of the displayed bicategory of terminal DMCs

is univalent in Section 4.3. The other two facts have been already shown in the library, thus we can show that the bicategory of display map categories with a terminal object in the base will be univalent.

Theorem 55 (`is_univalent_2_bicat_terminal_display_map_cat`)**.** The bicategory of terminal DMCs, obtained as the total bicategory of the displayed version, is univalent.

Proof. The univalence of total bicategories of displayed product bicategories is already shown in `is_univalent_2_total_dirprod`, and requires the following to hold

- the base bicategory must be univalent (`univalent_cat_is_univalent_2`);
- the displayed bicategory of categories with a terminal object must be univalent (`disp_ _univalent_2_disp_bicat_terminal_obj`);
- the displayed bicategory of DMCs must be univalent (Theorem 41).

All of these conditions have been shown prior, so we are done. □

5.2 Arrow Category as a Displayed Category

Another difference with the formalisation is the use of a displayed category instead of the arrow category $\mathcal{C}^{\rightarrow}$. This displayed category is called the codomain category Cod, and it can be described as a displayed version of the arrow category, with a focus on the codomain functor $\mathsf{cod} : \mathcal{C}^{\rightarrow} \to \mathcal{C}$.

Definition 56 (`disp_codomain`)**.** Given a category $\mathcal{C}$, the codomain displayed category Cod contains

- for every object $y : \mathcal{C}$, the type above Cod_x it is the type of pairs $(x, f) : \sum_{(x:\mathcal{C})} \mathcal{C}[\![x, y]\!]$ made from another object and a morphism from it pointing to y;
- for every morphism $g : \mathcal{C}[\![x, y]\!]$, the type above it $(x', f) \underset{g}{\to} (y', f')$ is the type of pairs

  $$(g', H) : \sum_{(g' : x' \to y')} g' \cdot f' = f \cdot g$$

  consisting of a morphism, and a witness to the fact that the following diagram commutes in C

  $$\begin{tikzcd} x' \arrow[r, "g'"] \arrow[d, "f"'] & y' \arrow[d, "f'"] \\ x \arrow[r, "g"'] & y. \end{tikzcd}$$

All operators and laws are given by the corresponding operator or law in $\mathcal{C}$.

The above definition will have the same objects as the arrow category, but each of them is displayed over its own codomain. This is why we use this form, as its *forgetful* projection functor will correspond to the codomain functor:

$$\mathsf{cod} := \pi_1^{\mathsf{Cod}} : \int \mathsf{Cod} \to \mathcal{C},$$

mapping elements of the total category $\int \mathsf{Cod}$ to their first element, i. e. it will map objects $(y, (x, f))$ to their base y, which is the codomain of f.

For this thesis, I have used the implementation of this displayed category as a guide when defining the arrow version of display map category. This was very useful, as the definition of comprehension categories in UniMath requires $\mathcal{T}$ to be a displayed category over $\mathcal{C}$, and this way I could use my definition for $\mathcal{D}^{\rightarrow}$ directly. The definition of $\mathcal{D}^{\rightarrow}$ is essentially the same as Definition 56, with an added third element in the objects

$$(x, f, H) : \sum_{(y:\mathcal{C})} \sum_{(f:\mathcal{C}[\![x,y]\!])} Df,$$

this third element will be a proof that f is a display map. The proofs for this being a displayed category follow a similar structure to Cod.

This switch to the displayed versions also extends to the arrow version of functors and natural transformations, both of which also become displayed. Further, the same is true for the arrow versions of the DMC related concepts that were implemented by me.

The reason I implemented the arrow version of display map category so similar to the arrow category is because the former is a subcategory of the latter. The arrow version of DMC contains a subset of the elements of the arrow category, but is otherwise identical in construction. This fact could be made more obvious, by constructing it as a subcategory of the arrow category. This would likely mean constructing it as yet another displayed category, now above the total version of the arrow category. This would then require the refactoring of the pseudofunctor, to use this new version instead.

5.3 Optimisation

When converting the proofs into computer code, we suddenly also need to worry about the speed of the proof. This is because while our proof may be correct, but takes a minute to verify, then it will not be too useful, especially when considering the size of the UniMath library, as it has a large number of proofs that all need to be checked. To this end, this section will present some of the steps I took to ensure that all proofs I write will be reasonably quick. I will walk through three examples, mentioning how and why I achieved the speedup.

A very useful tool for this type of analysis is Rocq's built-in timer tool `Time`, which can simply be added before any command to time it. The times included were achieved using my laptop (HP ZBook Studio G5), and are only meant as a reference.

5.3.1 Removing `rewrite` and `try`

This may be one of the greatest time saves I have managed to make during this thesis. The combination of the `try` and `rewrite` really made the type-checking of this fairly simple proof take a while. For context, this proof states that all bicategory laws hold in the displayed bicategory of DMCs, which is true since the type of *displayed 2-cells* is just the unit type, therefore any two displayed 2-cell terms over a natural transformation will be equal. The original version in Listing 5.1 took a total of 164 seconds to execute.

Listing 5.1: Original Proof

```
Definition disp_prebicat_laws_display_map_cat
: disp_prebicat_laws disp_prebicat_data_display_map_cat.
Proof.
  Time repeat split; intro; intros; try (destruct ηη);
  simpl; rewrite transportb_const; apply idpath.
Time Qed.
```

Time taken in total $164.349s$.

```
Finished transaction in 23.902 secs (9.784u,0.121s) (successful)
Finished transaction in 140.447 secs (52.798u,0.674s) (successful)
```

The improved version in Listing 5.2 took only 0.14 seconds to execute, which is a substantial speedup. The speedup is twofold, first, by removing the `try`, Rocq no longer has to try and recover for every sub-goal, thereby reducing the time of the proof. Second, and probably more importantly, by removing the `rewrite`-s, the type checking done at the end of the proof has also become quicker. While `rewrite` more closely resembles a paper proof step, it can complicate the definition or proof internally, thereby leading to a longer type checking step at the end of proof. Further, `rewrite` can only be used if there is a one-to-one match between the type of the term and goal, often requiring a simplification step (`simpl`) to be executed beforehand.

Listing 5.2: Improved version

```
Definition disp_prebicat_laws_display_map_cat
  : disp_prebicat_laws disp_prebicat_data_display_map_cat.
Proof.
  Time repeat split; intro; intros; apply isProofIrrelevantUnit.
Time Qed.
```

Time taken in total $0.136s$.

```
Finished transaction in 0.042 secs (0.041u,0.s) (successful)
Finished transaction in 0.094 secs (0.059u,0.s) (successful)
```

5.3.2 Using `etrans`

This example will demonstrate another way of removing `rewrite` from a proof. The method relies on the `etrans` tactic, an equivalent of the `transitivity` tactic for the equality types of the UniMath library. This can split a goal like `a = b` into two sub-goals `a = ?` and `? = b`, where `?` is a 'hole' that can be filled by the programmer by applying other tactics. Using this, I could replace the `rewrite`-s with `apply`.

Also important to note that here I introduce `try` to remove the need to deal with each sub-goal individually, and to not have to repeat lines. This does not ultimately matter, as in the final version (see `display_map_class_adj_to_disp_adjoint_equiv`) we have a slightly faster version without `try`, sadly that also removes the `etrans`, so we do not show that here. Further, the final version adds `abstract` for better code practices, and its performance impact will be discussed in the next section. Ultimately, while `etrans` can be useful in certain situations, its uses are limited, mainly due to the fact that it needs a goal of the form `a = b`.

The original proof, seen in Listing 5.3, is a proof that if the identity is a map in both directions between two DMCs, then there is an adjoint equivalence between the two. The complicated part is to show that we have an equivalence, as this involves many equalities. This original takes 10 seconds.

Listing 5.3: Original Proof

```
Definition display_map_class_adj_to_disp_adjoint_equiv
  {C : bicat_of_univ_cats} (D D’ : disp_bicat_display_map_cat C)
  : is_display_map_class_functor D D’ (functor_identity C)
    × is_display_map_class_functor D’ D (functor_identity C)
    -> disp_adjoint_equivalence (idtoiso_2_0 C C (idpath C)) D D’.
Proof.
  intros [HF HG].
  exists HF. use tpair.
  - exists HG. cbn. exact (tt ,, tt).
  - Time split; use tpair.
    + Time cbn; rewrite transportb_const; apply idpath.
    + Time cbn; rewrite transportb_const; apply idpath.
    + Time exists tt; split; cbn.
      * Time rewrite transportb_const; apply idpath.
      * Time rewrite transportb_const; apply idpath.
    + Time exists tt; split; cbn.
      * Time rewrite transportb_const; apply idpath.
      * Time rewrite transportb_const; apply idpath.
Time Defined.
```

Time taken in total: $10.239s$.

```
Finished transaction in 0.145 secs (0.145u,0.s) (successful)
Finished transaction in 0.854 secs (0.384u,0.026s) (successful)
Finished transaction in 0.727 secs (0.301u,0.008s) (successful)
Finished transaction in 0.145 secs (0.145u,0.s) (successful)
Finished transaction in 0.005 secs (0.001u,0.s) (successful)
Finished transaction in 0.012 secs (0.002u,0.s) (successful)
Finished transaction in 0.135 secs (0.135u,0.s) (successful)
Finished transaction in 0.012 secs (0.003u,0.s) (successful)
Finished transaction in 0.005 secs (0.001u,0.s) (successful)
Finished transaction in 8.199 secs (3.658u,0.093s) (successful)
```

The improved version combines all the different cases into a single line that uses `etrans`. This version in Listing 5.4 takes less than 2 seconds, meaning it is about 5 times faster than the original.

Listing 5.4: Improved Version

```
 Definition display_map_class_adj_to_disp_adjoint_equiv
   {C : bicat_of_univ_cats} (D D’ : disp_bicat_display_map_cat C)
   : is_display_map_class_functor D D’ (functor_identity C)
     × is_display_map_class_functor D’ D (functor_identity C)
     -> disp_adjoint_equivalence (idtoiso_2_0 C C (idpath C)) D D’.
 Proof.
   intros [HF HG].
   exists HF. use tpair.
   - exists HG. cbn. exact (tt ,, tt).
  - Time split; use tpair; try (exists tt; split);
    (etrans;
      [apply (idpath (idfun _ tt))
      | apply (eqtohomot (!(transportb_const _ _)) tt)]).
Time Defined.
```

Time taken in total: $1.962s$.

```
Finished transaction in 0.323 secs (0.322u,0.s) (successful)
Finished transaction in 1.639 secs (1.034u,0.s) (successful)
```

5.3.3 Impact of `abstract`

In this example we can see how I optimised one of the proofs that had a part with `abstract`. The command `abstract` hides away the implementation details of whatever is put inside it. It is good practice, and is recommended in the UniMath guide, to use these for parts of a definition that are just a proof of a property, as these will never need to be unfolded, and making them opaque can help speed up proofs that rely on these. One downside of `abstract` is the fact that it needs all commands inside it to be executed together, which makes editing proofs more difficult. The easiest way to deal with it is to simply remove the `abstract`, optimise the proof and put back the `abstract`.

When we do so, we can see the impact of `abstract` on the proof. Since it makes part of a definition opaque, it has to type check that part separately. This separate type checking increases the time the opaque part of the definition needs, but is somewhat compensated for by the final type checking of the definition. Overall, it still reduces performance slightly, though this may be compensated by the speed-up in the other proofs (by the opaqueness), though that is hard to measure.

The example is part of the definition of the function on 1-cells, when defining the pseudofunctor. More specifically, we are interested in the last part, where we need to show that we have a natural isomorphism. The original version in Listing 5.5 takes 9 seconds, with most of that time taken up by the `abstract` part.

Listing 5.5: Original Code

```
Definition terminal_display_map_functor_to_full_comp_cat_functor
  : ∏ D₁ D₂ : bicat_terminal_display_map_cat,
    bicat_terminal_display_map_cat ⟦ D₁ , D₂ ⟧
    -> bicat_full_comp_cat ⟦ ι D₁ , ι D₂ ⟧.
Proof.
  intros D₁ D₂ F; use make_full_comp_cat_functor.
  - use make_comp_cat_functor.
    + use make_functor_with_terminal_cleaving.
      * use make_functor_with_terminal_disp_cat.
        -- exact F.
        -- exact F.
        -- exact (display_map_functor F).
      * exact (is_cartesian_display_map_functor (_ ,, _)).
    + exact (pr1 (map_ι_is_ι_map F)).
  - Time abstract (exact (
      λ x dx, pr1 ((pr11 (map_ι_is_ι_map F)) x dx)
      ,, id_left _
      ,, id_right _
  )).
Time Defined.
```

Time taken in total: $9.095s$.

```
Finished transaction in 9.018 secs (3.322u,0.062s) (successful)
Finished transaction in 0.077 secs (0.03u,0.s) (successful)
```

By removing the `abstract` in Listing 5.6, we save 0.2 seconds, and now the time is almost evenly split between the proof and `Defined`.

Listing 5.6: Without Abstract

```
Definition terminal_display_map_functor_to_full_comp_cat_functor
  : ∏ D₁ D₂ : bicat_terminal_display_map_cat,
    bicat_terminal_display_map_cat ⟦ D₁ , D₂ ⟧
    -> bicat_full_comp_cat ⟦ ι D₁ , ι D₂ ⟧.
Proof.
  intros D₁ D₂ F; use make_full_comp_cat_functor.
  - use make_comp_cat_functor.
    + use make_functor_with_terminal_cleaving.
      * use make_functor_with_terminal_disp_cat.
        -- exact F.
        -- exact F.
        -- exact (display_map_functor F).
      * exact (is_cartesian_display_map_functor (_ ,, _)).
    + exact (pr1 (map_ι_is_ι_map F)).
  - Time exact (
      λ x dx, pr1 ((pr11 (map_ι_is_ι_map F)) x dx)
      ,, id_left _
      ,, id_right _
    ).
Time Defined.
```

Time taken in total: $8.838s$.

```
Finished transaction in 4.07 secs (1.511u,0.029s) (successful)
Finished transaction in 4.768 secs (1.854u,0.035s) (successful)
```

Following this, we can optimise the proof, by removing holes, and overall better guiding Rocq, to get Listing 5.7. This version drops the time needed by a further 3 seconds, cutting the time almost in half. Finally, we add back the `abstract` in Listing 5.8, which increases the time by 0.4 seconds, and moves back the time needed to the proof step from the end of definition.

Listing 5.7: Optimised

```
Definition terminal_display_map_functor_to_full_comp_cat_functor
  : ∏ D₁ D₂ : bicat_terminal_display_map_cat,
      bicat_terminal_display_map_cat ⟦ D₁, D₂ ⟧
      -> bicat_full_comp_cat ⟦ ι D₁, ι D₂ ⟧.
Proof.
  intros D₁ D₂ F; use make_full_comp_cat_functor.
  - use make_comp_cat_functor.
    + use make_functor_with_terminal_cleaving.
      * use make_functor_with_terminal_disp_cat.
        -- exact F.
        -- exact F.
        -- exact (display_map_functor F).
      * exact (is_cartesian_display_map_functor (_ ,, _)).
    + exact (pr1 (map_ι_is_ι_map F)).
    - Time intros x dx;
      exists (pr1 ((pr11 (map_ι_is_ι_map F)) x dx));
      exists (id_left _); apply id_right.
Time Defined.
```

Time taken in total: $5.454s$.

```
Finished transaction in 0.055 secs (0.033u,0.s) (successful)
Finished transaction in 5.399 secs (1.933u,0.033s) (successful)
```

Listing 5.8: Abstracted

```
Definition terminal_display_map_functor_to_full_comp_cat_functor
  : ∏ D₁ D₂ : bicat_terminal_display_map_cat,
      bicat_terminal_display_map_cat ⟦ D₁, D₂ ⟧
      -> bicat_full_comp_cat ⟦ ι D₁, ι D₂ ⟧.
Proof.
  intros D₁ D₂ F; use make_full_comp_cat_functor.
  - use make_comp_cat_functor.
    + use make_functor_with_terminal_cleaving.
      * use make_functor_with_terminal_disp_cat.
        -- exact F.
        -- exact F.
        -- exact (display_map_functor F).
      * exact (is_cartesian_display_map_functor (_ ,, _)).
    + exact (pr1 (map_ι_is_ι_map F)).
  - Time abstract (intros x dx;
      exists (pr1 ((pr11 (map_ι_is_ι_map F)) x dx));
      exists (id_left _); apply id_right
    ).
Time Defined.
```

Time taken in total: $5.863s$.

```
Finished transaction in 5.787 secs (2.031u,0.028s) (successful)
Finished transaction in 0.076 secs (0.031u,0.s) (successful)
```

5.3.4 Overview of optimisations

Use less rewrite. I have found that in general it was better to replace `rewrite` operations. The impact of this is twofold. First, to be able to use a `rewrite`, the goal must have an exact match to the expression in the `rewrite`, with minimal type unification, this often means using one of the simplifying commands (`simpl` or `cbn`) beforehand, which sometimes takes a while, as the computer goes through all the simplification steps. Second, at least in my experience, using `rewrite` also made the final type-checking (when calling `Defined` or `Qed`) take longer. Solutions to this issue were to either find or define an applicable proof, or to use the tactic `etrans`, which splits equality goals into two transitive equalities with a hole in the middle. This latter tactic makes it so instead of `rewrite`, one can `apply` the same theorem.

Avoid try. While this command can be useful to shorten and condense proofs, it should be removed if possible. In one of the examples (in Section 5.3.1), it was possible to remove it, improving the speed of the proof drastically (together with the removal of `rewrite`). The reason for this is simple, `try` will try to execute the tactic its given, and revert on a failure, which can take long if there are many goals it needs to try.

Note about abstract. The UniMath library's contribution guidelines advise to keep proofs for theorems opaque (meaning that its definition cannot be unfolded), since their proofs should be irrelevant. Similarly, one can `abstract` away parts of proofs that should be similarly opaque. The one downside of this, in terms of performance, is that `abstract` has to perform type-checking similar to those at the end of proof. Though some of this time is compensated by a faster end of proof, overall I still found it increasing the time taken by a proof. For an example of this, see Section 5.3.3, where we can see that the proof got quicker when removing `abstract`, and got slower by a bit again when adding the `abstract` back after optimisation.

Chapter 6

Related Work

Category Theory in Univalent Foundations. There are a good number of articles using UniMath[1], here, I will highlight the ones that are most relevant to this thesis, and its implementations. Categories and their univalence have been implemented by Ahrens, Kapulkin and Shulman [13]. This serves as the core of the category theory in UniMath, this implementation also contains the definitions for functors and natural transformations. The implementation of displayed categories was given by Ahrens and Lumsdaine[8]. In the same paper, they also gave the implementation of `disp_codomain`, which we used as reference, when implementing the arrow version $\mathcal{D}^{\twoheadrightarrow}$ of display map categories. They also gave the definition of fibrations for displayed categories, which we used in Section 2.2.1.

Finally, bicategories have been implemented by Ahrens et al. [11]. They have also given implementations of displayed bicategories, together with univalence for both. The product of displayed bicategories (see Definition 28) was also implemented by them, together with a set of useful proofs, such as the univalence of this product displayed bicategory.

Further, the bicategory of comprehension categories has been implemented in UniMath, in this pull request. It is responsible for our need to include the terminal object in the base of displayed categories, as discussed in Section 5.1. As discussed in that section, the construction of the bicategory is done through a series of displayed bicategories, each displayed over the total version of the previous.

Van der Weide [15] has also published their work with comparing univalent categories with a finite limits to univalent full democratic finite limit comprehension categories, and implementing this comparison in UniMath. They also use bicategories and pseudofunctors between them to make the comparison, but they manages to show that these pseudofunctors form biequivalences. Van der Weide also makes use of displayed bicategories, when constructing the bicategories for the comparison, similar to how we have done.

Semantic Frameworks. Over the years, there have been many different models presented for various type theories. Here, I will mention some of the more relevant ones to the topics discussed in this thesis.

An early example for the use of display map categories as a framework for dependent type theory was given by Hyland and Pitts [5]. In their paper they did not use the term display map category, referring to them as a class of display maps with a stability condition instead. North [16, Chapter 2] and Von Glehn [17, Chapter 2] both give descriptions of typeformers in a display map category model of type theory.

Comprehension categories on the other hand, were introduced by Jacobs [7]. In the same paper, he discusses the use of comprehension categories as a model for depen-

[1]Articles using UniMath

dent type theory. Jacobs later also published a book titled “Categorical Logic and Type Theory” [2], in which Jacobs presents the use of both display map categories and comprehension categories to model dependent type theory in Chapter 10. The book also presents display map categories as a special case of a comprehension category, consistent with the map presented in this thesis. Lumsdaine and Warren [14] have built on this model with comprehension categories, to add a notion for local universes, adapted from the same universes that underline Univalent Foundations.

Speaking of Univalent Foundations, Kapulkin and Lumsdaine [18] have given a categorical model for it. For this, they rely on a different categorical structure, called *contextual category*, which is another structure studied in the paper that inspired this thesis. As a model, *contextual categories* fall take types as primitives, and are thus closer to comprehension categories rather than display map categories. In a similar vein, Lumsdaine and Shulman [19] give a model for higher inductive types, which are present in Univalent Foundations as homotopy types.

Another model for dependent type theory is *categories with families*, as presented by Dybjer [20]. This model has also been shown to be equivalent to some others, and to show this, Clairambault and Dybjer [21] have also used bicategories and a biequivalence between them.

For the comparison of such structures, we have been following the method given by Ahrens, Lumsdaine and North [1], that being, using bicategories and biequivalences. This is not the only way such comparisons can be made, for example Ahrens, Lumsdaine and Voevodsky [22] make their comparisons via *relative universes* (a concept separate from that of universes in Univalent Foundations).

Chapter 7

Conclusion

In this thesis, I have completed the formalisation of a number of concepts from category theory used to model type theory in Univalent Foundations. We started by stating what it means to be a display map category, continuing with the predicate of when a functor is a map between two display map categories. Our formal version definition of display map category (see Definition 33) included chosen pullbacks, and did not ask for repleteness, as we have shown that this can be guaranteed by taking the base category to be univalent. Using these two types, we could construct a displayed category over the bicategory of univalent categories in Section 4.2. As discussed in the same section, the total version of this displayed category is equivalent to the bicategory of display map categories. Finally, we described the construction of a pseudofunctor from the bicategory of display map categories into the bicategory of full comprehension categories.

All of this work has also been implemented and verified by the Rocq proof assistant, using the UniMath library. Following the implementation of the display map category, we have further implemented its interpretation as a displayed category, similar to the displayed category `disp_codomain`, which is a displayed category similar to the arrow category, but with an added focus on the functor $\mathsf{cod} : \mathcal{C}^{\to} \to \mathcal{C}$, as discussed in Section 5.2. This implementation could be improved, by showing the subcategorical nature of $\mathcal{D}^{\to}$, or by directly constructing it as such. Further, I have implemented a direct function from display map category to comprehension category, and related functions for their maps and transformations, all of which has been good practice and could be partially reused in the following.

The construction of the bicategory was done both directly and as a displayed bicategory over the bicategory of univalent categories. The prior requiring a lot more effort, and serving as good practice for the construction of the latter. In both cases I had to add a terminal object into the base categories, as this was required by the formalisation of the bicategory of comprehension categories. I have not proven that these two implementations are equivalent, but as I have argued at the end of Section 4.2, I am reasonably confident in that they are equivalent, as they both rely on the same definitions for the display map category. Further, we have constructed a pseudofunctor into the bicategory of full CCs from both versions. The construction of these has further convinced me of the similarity of the two implementations, as most proofs and definitions could be reused with minimal modifications to account for the slightly different structure.

While some of this work has been a little redundant, constructing multiple versions of the same concept, the direct versions contributed greatly to my understanding of the concepts, while the displayed versions should serve as a good basis for future work. As I have shown in Section 5.1, the construction of these displayed bicategories should allow them to be used with or without the terminal object in the base. Further, it allowed me to show the univalence of the bicategory far more easily than it would have been for the direct version. However, there is still more work to be done, especially when it comes to the bicategory of CCs, to also allow for a version without requiring the terminal object, and without having to duplicate all the code for it.

Throughout this project, I have learned a great deal about Univalent Foundations, type theories and their categorical models. I have also got a taste of developing a mathematical library in Rocq. While the results given in this paper are not novel, I do still think it is valuable to have these results be validated by a computer. Further, I hope that my contributions to the UniMath library may be useful for formalising future comparisons, or for any other purpose.

7.1 Future Work

What we managed to formalise in this thesis is just a part of the comparisons presented in [1]. The paper also includes three more frameworks with display maps as types, and two more with primitives as types, not counting variants. For reference, check the first two figures in their paper, which gives an overview of this. All of those correspondences could be formalised in a manner similar to the one presented in this thesis.

Equivalence. In this thesis, we have formalised that there is a corresponding comprehension category to every display map category. It would be interesting to finish this correspondence by giving a 'backwards' direction. This would need to involve stating any other relevant properties of the comprehension category that corresponds to a display map category. Then, using those properties, we could define the pseudofunctor backwards. Finally, in would remain to show that the two pseudofunctors form an equivalence.

UniMath. Throughout the thesis, I have mentioned a number of possible improvements to the UniMath library. These may not be too relevant academically, but I feel that for completeness, they are worth summarising here. First, the arrow version of display map category could be given as a subcategory of the arrow category, see the end of Section 5.2. Second, I have not proven that my implementations of the direct and displayed versions of the bicategory of display map categories are equivalent, this should not be difficult to show as they both use the same implementations for the display map category. I have given a comparison of the two in Section 4.2, which gives me confidence in that the two versions are indeed equivalent. As discussed in Section 5.1, the current version of the bicategory of comprehension categories is defined with base categories that contain a terminal object. For completeness, a version without the terminal could also be defined, though this would either be just a lot of repeated code, or require reorganisation of the current construction. Finally, as mentioned at the end of Section 4.2, ideally some parts of the pseudofunctor construction could be avoided by creating the pseudofunctor as a displayed one, but this too would require at least some reorganisation or rewriting of the construction of the bicategory of CCs. While none of these might be large enough for a thesis project, they may still be important for the health of the UniMath library.

Bibliography

[1] B. Ahrens, P. L. Lumsdaine, and P. R. North, “Comparing semantic frameworks for dependently-sorted algebraic theories,” in *Programming Languages and Systems - 22nd Asian Symposium, APLAS 2024, Kyoto, Japan, October 22-24, 2024, Proceedings*, O. Kiselyov, Ed., ser. Lecture Notes in Computer Science, vol. 15194, Springer, 2024, pp. 3–22, ISBN: 978-981-97-8943-6. DOI: 10.1007/978-981-97-8943-6_1. [Online]. Available: https://doi.org/10.1007/978-981-97-8943-6_1.

[2] B. P. Jacobs, *Categorical Logic and Type Theory* (Studies in logic and the foundations of mathematics). North-Holland, 2001, vol. 141, ISBN: 978-0-444-50853-9. [Online]. Available: https://shop.elsevier.com/books/categorical-logic-and-type-theory/jacobs/978-0-444-50170-7.

[3] B. Ahrens and K. Wullaert, “Category Theory for Programming,” Sep. 2022. DOI: 10.48550/arXiv.2209.01259. [Online]. Available: https://doi.org/10.48550/arXiv.2209.01259.

[4] B. Milewski. “Category Theory for Programmers. ”[Online]. Available: https://bartoszmilewski.com/2014/10/28/category-theory-for-programmers-the-preface/.

[5] J. M. E. Hyland and A. M. Pitts, “The theory of constructions: Categorical semantics and topos-theoretic models,” in *Categories in computer science and logic (Boulder, CO, 1987)*, ser. Contemp. Math. Vol. 92, Amer. Math. Soc., Providence, RI, 1989, pp. 137–199, ISBN: 0-8218-5100-4. DOI: 10.1090/conm/092/1003199. [Online]. Available: https://doi.org/10.1090/conm/092/1003199.

[6] P. Taylor, *Practical Foundations of Mathematics* (Cambridge studies in advanced mathematics). Cambridge University Press, 1999, vol. 59, Transferred to digital printing 2003, ISBN: 978-0-521-63107-5.

[7] B. Jacobs, “Comprehension categories and the semantics of type dependency,” *Theoretical Computer Science*, vol. 107, no. 2, pp. 169–207, 1993.

[8] B. Ahrens and P. L. Lumsdaine, “Displayed categories,” *Logical Methods in Computer Science*, vol. 15, no. 1, 2019. [Online]. Available: https://lmcs.episciences.org/5252.

[9] J. Bénabou, “Introduction to bicategories,” in *Reports of the Midwest Category Seminar*, Berlin, Heidelberg: Springer Berlin Heidelberg, 1967, pp. 1–77, ISBN: 978-3-540-35545-8.

[10] nLab authors. “Bicategory. ”[Online]. Available: https://ncatlab.org/nlab/revision/bicategory/43.

[11] B. Ahrens, D. Frumin, M. Maggesi, N. Veltri, and N. van der Weide, “Bicategories in univalent foundations,” *Mathematical Structures in Computer Science*, vol. 31, no. 10, pp. 1232–1269, 2021. doi: 10.1017/S0960129522000032. [Online]. Available: https://doi.org/10.1017/S0960129522000032.

[12] *Homotopy Type Theory - Univalent Foundations of Mathematics: The Univalent Foundations Program*. Institute for Advanced Study, 2013. [Online]. Available: https://homotopytypetheory.org/book/.

[13] B. Ahrens, K. Kapulkin, and M. Shulman, “Univalent categories and the rezk completion,” *Mathematical Structures in Computer Science*, vol. 25, no. 5, pp. 1010–1039, 2015. doi: 10.1017/S0960129514000486. [Online]. Available: http://dx.doi.org/10.1017/S0960129514000486.

[14] P. L. Lumsdaine and M. A. Warren, “The local universes model: An overlooked coherence construction for dependent type theories,” *ACM Trans. Comput. Log.*, vol. 16, no. 3, p. 23, 2015. doi: 10.1145/2754931. [Online]. Available: http://doi.acm.org/10.1145/2754931.

[15] N. van der Weide, “The internal languages of univalent categories,” *CoRR*, vol. abs/2411.06636, 2024. doi: 10.48550/ARXIV.2411.06636. arXiv: 2411.06636. [Online]. Available: https://doi.org/10.48550/arXiv.2411.06636.

[16] P. R. North, “Type theoretic weak factorization systems,” Ph.D. dissertation, Apollo - University of Cambridge Repository, 2017. doi: 10.17863/CAM.11207. [Online]. Available: https://www.repository.cam.ac.uk/handle/1810/265152.

[17] T. Von Glehn, “Polynomials and models of type theory,” Ph.D. dissertation, Apollo - University of Cambridge Repository, 2015. doi: 10.17863/CAM.16245. [Online]. Available: https://www.repository.cam.ac.uk/handle/1810/254394.

[18] K. Kapulkin and P. L. F. Lumsdaine, “The simplicial model of univalent foundations (after voevodsky),” *Journal of the European Mathematical Society*, vol. 23, pp. 2071–2126, 6 Mar. 2021, issn: 1435-9855. doi: 10.4171/JEMS/1050. [Online]. Available: https://ems.press/journals/jems/articles/274693.

[19] P. L. LUMSDAINE and M. SHULMAN, “Semantics of higher inductive types,” *Mathematical Proceedings of the Cambridge Philosophical Society*, vol. 169, no. 1, pp. 159–208, Jun. 2019, issn: 1469-8064. doi: 10.1017/s030500411900015x. [Online]. Available: http://dx.doi.org/10.1017/S030500411900015X.

[20] P. Dybjer, “Internal type theory,” in *Types for Proofs and Programs, International Workshop TYPES 95, Torino, Italy, June 5-8, 1995, Selected Papers*, S. Berardi and M. Coppo, Eds., ser. Lecture Notes in Computer Science, vol. 1158, Springer, 1995, pp. 120–134, isbn: 3-540-61780-9. doi: 10.1007/3-540-61780-9_66. [Online]. Available: http://dx.doi.org/10.1007/3-540-61780-9_66.

[21] P. Clairambault and P. Dybjer, “The biequivalence of locally cartesian closed categories and martin-löf type theories,” in *Typed Lambda Calculi and Applications - 10th International Conference, TLCA 2011, Novi Sad, Serbia, June 1-3, 2011. Proceedings*, C.-H. L. Ong, Ed., ser. Lecture Notes in Computer Science, vol. 6690, Springer, 2011, pp. 91–106, isbn: 978-3-642-21690-9. doi: 10.1007/978-3-642-21691-6_10. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-21691-6_10.

[22] B. Ahrens, P. L. Lumsdaine, and V. Voevodsky, “Categorical structures for type theory in univalent foundations,” *Logical Methods in Computer Science*, vol. 14, no. 3, 2018. doi: 10.23638/LMCS-14(3:18)2018. [Online]. Available: https://doi.org/10.23638/LMCS-14(3:18)2018.

Acronyms

CC	comprehension category
DMC	display map category
UF	Univalent Foundations