SJÄLVSTÄNDIGA ARBETEN I MATEMATIK
MATEMATISKA INSTITUTIONEN, STOCKHOLMS UNIVERSITET

The Category of Iterative Sets in Cubical Agda

av
Fabian Lukas Grubmüller

2026 – No M1

The Category of Iterative Sets in Cubical Agda

Fabian Lukas Grubmüller

Självständigt arbete i matematik 30 högskolepoäng, avancerad nivå

Handledare: Anders Mörtberg

2026

Abstract

Iterative sets form a constructive Tarski-style universe V^0 of h-sets. This universe is closed under common type-theoretic constructions and is itself an h-set. It arises naturally from the study of iterative multisets, where V^0 is defined as a specific type-indexed W-type for which the indexing function is restricted to embeddings, effectively collapsing higher structure.

In previous work, Gratzer, Gylterud, Mörtberg, and Stenholm showed that V^0 is a model of dependent type theory, in particular a Category with Families (CwF), that admits both Π- and Σ-structures. While their proofs were rather straightforward on paper, often reducing to reflexivity, their formalization in standard Agda (using agda-unimath) faced significant obstacles. For the formalization of Π- and Σ-structures, they faced problems due to complex path algebra involving multiple layers of transport and function extensionality, leading them to abandon the formalization of the Σ-structure.

In this thesis, we explore whether a formalization in Cubical Agda, an extension of Agda for cubical type theory, is easier to accomplish. We implement the general properties of iterative sets, as well as CwF and Σ-structures. For the latter we use three distinct strategies: a naive translation of the prior work, a more cubical approach replacing equalities containing transport with heterogeneous path types, and a strategy that eliminates transport in favor of ad-hoc functions that can be later instantiated by the identity function. We find that while the cubical metatheory simplifies reasoning about extensionality, it also introduces new challenges. One of the main issues is the lack of a definitional J-rule in Cubical Agda, which means that certain terms do not compute definitionally, requiring manual handling of transport structures. Ultimately, we are also unable to finish the proof of the naturality condition for Σ-structures due to the inherent complexity of the goal types. However, our approach substantially simplifies the remaining proof goals, which makes us hopeful that the proof will be able to be completed in future work. We conclude that while Cubical Agda improves clarity in specific areas, we concede that the trade-off regarding the definitional behaviour of the J-rule makes the balance between benefits and downsides approximately equal.

Sammanfattning

Iterativa mängder utgör ett konstruktivt Tarski-universum V^0 av h-mängder. Detta universum är slutet under vanliga typteoretiska konstruktioner och är självt en h-mängd. Det uppstår naturligt från studiet av iterativa multimängder, där V^0 definieras som en specifik typ-indexerad W-typ där indexeringsfunktionen begränsas till inbäddningar, vilket effektivt kollapsar högre struktur.

I tidigare arbete visade Gratzer, Gylterud, Mörtberg och Stenholm att V^0 är en modell för beroende typteori, i synnerhet en kategori med familjer (CwF), som medger både Π- och Σ-strukturer. Även om deras bevis var ganska rättframma på papper, och ofta reducerades till reflexivitet, stötte deras formalisering i standard-Agda (med användning av agda-unimath) på betydande hinder. Vid formaliseringen av Π- och Σ-strukturer mötte de problem på grund av komplex vägalgebra som involverade flera lager av transport och funktionsextensionalitet, vilket ledde till att de övergav formaliseringen av Σ-strukturen.

I denna uppsats undersöker vi huruvida en formalisering i Cubical Agda, en utvidgning av Agda för kubisk typteori, är lättare att genomföra. Vi implementerar de allmänna egenskaperna hos iterativa mängder, såväl som CwF- och Σ-strukturer. För de senare använder vi tre olika strategier: en naiv översättning av det tidigare arbetet, en mer kubisk metod som ersätter likheter innehållande transport med heterogena vägtyper, samt en strategi som eliminerar transport till förmån för ad hoc-funktioner som senare kan instansieras av identitetsfunktionen.Vi finner att även om den kubiska metateorin förenklar resonemang kring extensionalitet, introducerar den också nya utmaningar. Ett av huvudproblemen är avsaknaden av en definitionell J-regel i Cubical Agda, vilket innebär att vissa termer inte beräknas definitionellt, vilket kräver manuell hantering av transportstrukturer. Slutligen lyckas vi inte heller slutföra beviset för naturalitetsvillkoret för Σ-strukturer på grund av den inneboende komplexiteten i måltyperna. Vårt tillvägagångssätt förenklar dock avsevärt de återstående bevismålen, vilket gör oss hoppfulla om att beviset kommer att kunna slutföras i framtida arbete. Vi drar slutsatsen att även om Cubical Agda förbättrar tydligheten inom specifika områden, medger vi att avvägningen gällande J-regelns definitionella beteende gör att balansen mellan fördelar och nackdelar är ungefär jämn.

On The Use Of Artificial Intelligence

I declare that the research, reasoning and writing of this thesis are solely the product of my own natural intelligence (NI). Artificial intelligence (AI) was only used to create the obligatory Swedish version of the abstract. For this, my original English version was translated using Gemini-3 and subsequently reviewed by a native Swedish speaker.

Acknowledgements

First and foremost, I would like to thank Anders for his advice and the much-appreciated help I received on the many occasions when Agda did not behave as I wanted it. I would also like to thank Peter and Gabriel for supporting me and offering great insights throughout the various logic courses at Stockholm University. In this same spirit, I want to thank Iosif for introducing me to the world of logic and constructive mathematics in the first place during my time at LMU Munich, as well as for his continued support.

Next, I want to thank everyone who supported me during my master's and made it such an enjoyable experience. Thank you, Num, for always being by my side, even when things were stressful. Thank you Anton, Giacomo, Alberto, Jorge, Vlad, Keno, Zhirui, Celia, Aleksei, Andrea, and Valeria for the many fun experiences, including all the board game sessions. I am particularly grateful to Anton and Giacomo for the numerous logic discussions and more or less productive study sessions together. I also want to thank Giacomo, Florian, Edvin, Victor, Jean-Baptiste, and Leonora for the effort we shared in organizing the master seminars.

Finally, I would like to thank my parents for supporting me since day one and for allowing me to follow my path, even though it means I live far away.

This document has been typeset using typst 0.14.2.

Contents

Introduction

As mathematical research progresses, new research fields emerge, old techniques are refined and many deep connections between fields are unveiled. As such, modern mathematics has been characterized by the emergence of ever more sophisticated theories and techniques in order to better understand new and old problems alike. At the same time, this development has made it harder and harder for individual mathematicians to grasp all the new advances in their respective fields and to provide effective feedback to peers.

Alongside the increasing complexification of mathematical research, the field of computer-assisted proof has been rapidly emerging. The idea is that if a dedicated computer program accepts a proof, then we can trust that the proof is correct. While even a computer program can never give a full guarantee, e.g. there could be programming or even hardware bugs present, by relying on an (in principle) unbiased program with a small, well-defined, and easily intelligible core logic, we can significantly decrease the surface for errors.

While proof verification can be seen as the first major goal for the field, modern proof assistants offer a variety of additional features that are meant to *assist* the user with writing such a proof in the first place. These features include among others a detailed description of the proof goal, gradual proof refinement, smaller subgoals as well as the automation of various simple but tedious tasks, like equational reasoning. With them, formalizing a proof with a proof assistant is an interactive process in which the user gradually implements the proof step by step with guidance and reassurance from the proof assistant at every step.

In the field of computer formalization, the major proof assistants are Lean [21], Rocq (formerly: Coq) [22] and Agda [23]. While Lean is very well suited for formalizing classical mathematics, the strengths of Rocq and Agda lie more in providing the ability to formalize proofs in a constructive or otherwise non-standard framework. These include, among others, univalent homotopy type theory and cubical type theory. The principal repository of formalizing univalent mathematics in Rocq is the rocq-unimath library [24], and a similar repository exists for Agda in the form of the agda-unimath library [18]. The main effort of formalizing cubical mathematics is the agda-cubical library [25] for the Cubical Agda proof assistant [27].

Type theory is a logical framework that is characterised by a strong computational notion of proof. In contrast to set theory, where certain sets are (non-constructively) postulated by axioms, type theory takes a different more constructive approach in the sense that it defines precisely how to build new types from other types and how to construct its elements. This makes type theory a viable foundation of constructive mathematics and more generally for use in proof assistants. In fact, most modern proof assistants are built on some type theory.

Iterative sets were originally introduced by P. Aczel [1] in order to develop a constructive account of set theory. It is based on type theory and uses the concept of W-types, i.e. well-founded trees, to mimic the cumulative hierarchy of sets, where every set at a certain level can be built from the sets of lower levels. The use of W-types has the advantage that it is much simpler, i.e. it can be carried out in a simpler base type theory, than other approaches using e.g. inductive-inductive or inductive-recursive types.

It turns out that in our modern homotopical interpretation of type theory, this theory of iterative sets is in fact a theory of multisets. H. R. Gylterud and E. Stenholm [10] then show that by restricting the indexing functions of the W-type, one can derive a hierarchy of truncated iterative multisets. At the bottom of this hierarchy sit the iterative sets, which have been extensively studied by D. Gratzer, H. R. Gylterud, A. Mörtberg, and E. Stenholm [7]. In their work, they elaborate on the fact that iterative sets define a universe of set that is itself a set. Moreover, they show that this iterative set universe is in fact a model of dependent type theory. In particular, they show that it can be equipped with the structure of a Category with Families (CwF), a kind of categorical model that promises better equational behaviour and thus easier computer formalization by utilizing an explicit notion of terms. In addition, they also show that iterative sets can also be equipped with structures that allow it to capture the notion of Π- and Σ-types.

As part of their work, D. Gratzer, H. R. Gylterud, A. Mörtberg, and E. Stenholm [7] also provide a computer formalization of the theory using Agda together with the agda-unimath library. The notable exception to this is the definition of Σ-structure as well as the corresponding implementation for iterative sets, which the authors deemed to be too complicated, even though the proofs seem conceptually simple on paper.

While the authors did not specify the exact reason, it is likely that it is due to the non-reducing nature of function extensionality in univalent homotopy type theory. The premise of this thesis is the conjecture that a formalization with a cubical metatheory might be simpler, since it allows, among other things, the constructive proof of function extensionality. The major part of this thesis is the formalization of the theory of iterative sets in the proof assistant Agda. While it has not proven to be true that the formalization becomes easier overall, in fact, many proofs become harder, using various methods, we managed to significantly improve the goal type of the remaining proof, albeit ultimately failing to complete the proof. We are, however, confident that one should be able to formalize the proof eventually.

Contribution

The main contribution of this thesis is the formalization [8] of the theory of iterative sets using Cubical Agda together with the agda-cubical library. For this, we first define the corresponding structure of iterative multisets and iterative sets. We then show that

iterative multisets encode a universe of sets that is itself a set. We then show that this universe is closed under some common type constructions and that it forms a category that is closed under certain categorical constructions. For this, we generally follow the prior formalization, but we sometimes need to alter the proofs significantly due to the fact that the J-rule (path induction) is not definitional in cubical type theory. Another point of divergence is the fact that the agda-cubical and agda-unimath libraries differ significantly, so direct translations would require extensive reimplementation work for all the definitions that are found in the agda-unimath library but not in the agda-cubical library.

Next, we define CwF and Σ-structures. We do this in three stages: First, we define CwF in a way that closely follows the prior formalization. Then, we define Σ-structure in a similar way, though in this case there is no prior formalization. Then, we define CwF in a slightly more cubical way by using heterogenous path types instead of equality types. Lastly, we get rid of transport completely in all definitions (though not in the later implementation) by replacing it with ad-hoc functions, that can later be instantiated by the identity function. For all three stages, we implement the CwF and Σ-structures, leaving open only the naturality condition of the equivalence that is part of the Σ-structure. Due to the difference between cubical and homotopy type theory, the implementation of the structures differs significantly from the one in the prior formalization.

Even though we are not successful in proving the naturality condition, we significantly improve its goal type and point out ways towards a potential solution.

Thesis Structure

In Chapter 1, we introduce the necessary prerequisites for our later treatment. We do this by first introducing the basics of dependent type theory (Section 1.1), and then highlighting its modern homotopical interpretations in the sense of homotopy type theory (Section 1.2) and cubical type theory (Section 1.3). Next, we will introduce models of dependent type theory in the form of Categories with Families as well as the corresponding Σ-structure (Section 1.4). Finally, we will introduce the Agda proof assistant and will demonstrate the usual workflow, particularly for Cubical Agda, by means of a practical example (Section 1.5).

In Chapter 2, we first introduce the notion of iterative sets and iterative multisets, including the proofs of some important basic properties (Section 2.1). We then show that iterative sets form a universe of sets that is closed under various standard type constructions (Section 2.2). Lastly, we show that iterative sets form a model of type theory by exhibiting a corresponding CwF structure as well as a Σ-structure (Section 2.3).

In Chapter 3, we explain the formalization of the proofs from Chapter 2. First, we explain the prior formalization by D. Gratzer, H. R. Gylterud, A. Mörtberg, and E.

Stenholm [7] and highlight potential reasons why the authors failed to formalize Σ-structures (Section 3.1). We then demonstrate our formalizations first in the naive version and then using two different sets of improvements (Section 3.2). We then highlight potential improvements that might make a potential future formalization easier (Section 3.3). Lastly, we will explain where exactly to find our formalization since it is spread out over multiple files within the agda-cubical repository.

1. Preliminaries

In this chapter we introduce the type theoretical foundations necessary for the later work. We first introduce the essential notions of dependent type theory due to P. Martin-Löf [13] and their homotopical interpretation under the name of homotopy type theory. For a comprehensive introduction, we suggest the reader have a look at either of The Univalent Foundations Program [20], E. Rijke [17] or C. Angiuli and D. Gratzer [3]. In this thesis we assume some (very) basic knowledge of category theory, for a good and modern introduction to category theory see e.g. E. Riehl [15].

Next, we give a short introduction to cubical type theory, a more recent approach that aims to solve some of the computational shortcomings of standard homotopy type theory. For a good overview, we recommend to read A. Mörtberg [14] or the corresponding chapter of C. Angiuli and D. Gratzer [3].

Lastly, we will give a high-level overview over the proof assistant Agda [23], including an explanation of its capabilities and the usual workflows. We introduce both standard Agda, as well as its extension Cubical Agda [27] and the corresponding agda-cubical library [25], that will be the basis of our formalization later.

1.1. Martin-Löf's Type Theory

Type theory is a family of logical frameworks that are characterised by an internal notion of computation. In mathematics a type theory can serve as a basis for carrying out constructive mathematics and in computer science it can be used to structure data and code as well as enforce certain correctness guarantees. The last aspect has also found its way to mathematics in the form of the emergent field of computer formalization. Naturally, many major automatic theorem provers are internally based on type theory.

Even though, type theory is a whole family of such logical frameworks, we usually also refer to some particular member of that family as "type theory", similarly for homotopy type theory and cubical type theory later. In the following we will introduce the concepts of Martin-Löf's type theory (MLTT) [13]. The type theory in question will be dependently typed, meaning that types are allowed to depend on a particular context (i.e. under some particular assumptions), and intensional, meaning that there will be an internal notion of equality that is (in general) coarser than the outside definitional equality.

While ordinary set theory is unstructured in the sense that every object is simply a set, type theory has a more structured approach where there is a distinction between types (i.e. containers) and elements. In set theory, it is necessary to add (non-constructive) axioms to derive the existence of certain sets, e.g. the de-facto standard set-theoretic

framework of most of classical mathematics is ZFC, Zermelo–Fraenkel set theory together with the axiom of choice. Type theory, on the other hand, is much more constructive. In order to specify a particular type theory, one generally needs to add certain type rules that enforce the existence of certain type and type constructions. The difference to set theory is that these rules specify how exactly, different types are built from other types, how canonical elements of a particular type are constructed, how to eliminate elements of a certain type and finally how the elimination computes. Mostly in computer science, elimination is often referred to as pattern matching, because it allows us to define functions by specifying what their image is for the canonical elements. An example are the natural numbers, where we can define recursive functions by specifying what the image of 0 is and, supposing that we know what the image of n is, specifying what the image of $n + 1$ is. In this case, specifying the computation ensures that e.g. a term $1 + 1$ actually reduces to 2, i.e. the two-fold successor of the canonical element 0.

This difference between set theory and type theory is also expressed in the different nature of proof. In set theory, proof is entirely a concept of the surrounding metatheory, while in type theory exact proof terms are constructed as part of the type theory and the metatheory is merely used to verify whether that proof-term is well-typed. This also explains the difference between the set-theoretic elementhood $x \in A$ and the type-theoretic typehood $x : A$. While the first is a proposition in the sense that it is a statement subject to proof, the typehood $x : A$ is a judgement of well-typedness, which is (usually trivially) decidable in the metatheory.

Simple Type Constructions

In the following, we will define the typing rules for our version of of MLTT. We start by adding the basic types, i.e. the empty type 0, the unit type 1, the type 2 of Booleans and the type $\mathbb{N}$ of natural numbers. For the empty type, we simply declare that there is a type 0, without adding any canonical elements. To recover the familiar properties, we specify that we can eliminate a term of 0 into any arbitrary type. This is analogous to the principle *ex falso* in intuitionistic and classical logic. Dually, we specify a type 1 and specify that it has the canonical element 0_1. We do not add any eliminators. The empty and unit types behave analogous to the empty set and an arbitrary singleton set, i.e. they also form the initial and terminal elements of a suitable category. For the type 2 of Booleans we define the canonical elements 0_2 and 1_2. To eliminate an element of 2, we specify that we only need to define what to do if that element is one of the canonical elements of 2.

Next, we define the type $\mathbb{N}$ of natural numbers. Its canonical elements are 0 and $S(n)$ for some natural number $n : \mathbb{N}$, and in order to eliminate an element of $\mathbb{N}$, we need to specify what happens if that element is 0 and if that element is $S(n)$, where we already

know what happens in the case of n. This will mean later in particular that all functions from $\mathbb{N}$ are recursively defined.

So far the types we added are all atomic in the sense that they are not built up from other types. In the following, we will add the following common compound types (type constructions): type sums (coproducts), Π-types (dependent functions), Σ-types (dependent pairs), W-types (wellfounded trees) and identity/equality types.

For all types A and B, we define the sum type $A + B$ that has two kinds of canonical elements: $\mathrm{inl}(a)$ for some $a : A$ (i.e. an element of A) and $\mathrm{inr}(b)$ for some $b : B$ (i.e. an element of B). In order to eliminate an element of $A + B$, we need to say what happens in the case that that element is an element of A and the case that that element is an element of B.

Dependent Functions and Dependent Pairs

Next we define Π-types. Elements of Π-types are dependent functions, i.e. functions where the type of the function value can depend on the function argument. For this, suppose that A is a type, and B is a type family over A (i.e. the type B can depend on some variable $x : A$). Then we define the Π-type $\prod_{x:A} B(x)$ and let its elements be λ-term $\lambda(x : A).t(x)$, where $t(x) : B(x)$. For the sake of convenience, we will often omit the type of x, i.e. we write $\lambda x.t(x)$, or, if t does not depend on x, we will also sometimes write $\lambda\_.t$. The elimination principle of Π-types is function application. That means, if we have a term $f : \prod_{x:A} B(x)$ and some $a : A$, then $f(a)$ is some element of $B(a)$. The computation rule makes sure that applying a λ-term to a term $a : A$ is the same as substituting x for y in its definition. Note that the availability of Π-types in our type theory now means in particular that type families are themselves types. Also, the notion of ordinary functions can be recovered by letting the type family B be constant (i.e. not depending on any $x : A$). We will then write $A \to B$ instead of $\prod_{x:A} B$.

While Π-types contain dependent functions, Σ-types contain the (dependent) graphs of such dependent functions. In other words, if A is a type and B a type family over A as above, then the canonical elements of a new type $\sum_{x:A} B(x)$ are pairs (a, b) where $a : A$ and $b : B(a)$. The elimination principle allows us to treat every element of $\sum_{x:A} B(x)$ as such a dependent pair (a, b). Similarly to Π-types, ordinary (Cartesian) product types can be recovered by letting the type family B be constant. In this case we will then simply write $A \times B$ instead of $\sum_{x:A} B$. Σ-types are often used to attach certain structure to a type.

There is a deep connection between dependent type theory (i.e. computation) and intuitionistic first-order logic (i.e. proof), often called the Curry-Howard Correspondence or the Propositions-as-Types Paradigm. It can be boiled down to the fact that proving all-quantification and implication is done similarly to finding a term of a Π-type or simple function type, and that proving a constructive existence or a conjunction is

carried out similarly to giving a type of a Σ-type or simple product type. In this sense, the constants ⊥ and ⊤ correspond to the types 0 and 1, and disjunction to sum types. This allows us to translate mathematical statements that we want to prove into types of which we want to find well-typed terms. For this reason, there is no real distinction between proving a statement and finding a term.

However, the above translation is not completely isomorphic: a term $t : \sum_{x:A} B(x)$ is stronger than a proof of $\exists_{x:A} B(x)$. This is demonstrated by the so-called Type-Theoretic Axiom of Choice:

$$\prod_{x:A} \sum_{y:B} C(x,y) \to \sum_{f:A\to B} C(x, f(x))$$

which is an almost trivially provable statement and does not involve any kind of choice, since the left-hand side of the implication is much stronger than its logic-counterpart. C. Angiuli and D. Gratzer [3] therefore also paraphrase it as the "illusion of choice".

Equality Types

So far, the only concept of equality in our type theory is the (outside) definitional/judgmental equality which can only equate terms that are trivially the same, i.e. in particular it does not allow us to judge an equality that follows from any of the elimination rules, e.g. recursion of the natural numbers ℕ. To make our type theory useful for developing and formalizing mathematics, we now add inductively defined internal, also called propositional, equality types. For every type A and elements $x : A$ we now add the type of equalities between x and y in A, denoted $x \equiv_A y$[1]. The canonical elements and elimination principle are quite different in comparison to the previous type constructions: They are of the form $\mathrm{refl}_x : x \equiv_A x$ and the elimination rule, called the J-rule, works as follows: Suppose $x, y : A$, and $p : x \equiv_A y$. Then we only need to look at the case where $x = y : A$ definitionally and p is the proof refl_x. While canonically, only $x \equiv_A y$ are inhabited, other equalities can easily be defined using the eliminators of other type constructions. It turns out that this notion of equality is enough to prove the usual properties of equality (see Section 1.5) and to make our type theory a suitable foundation of mathematics. In a proof assistant, we will only ever actively use the internal notion of equality in our proofs. The proof assistant itself will use the definitional equality in order to decidably verify our proof. An important example where we will use equality types together with Σ-types are fibers. If A and B are types and $f : A \to B$ is a function, then we define the fiber over $b :$

[1]Note that we use non-standard notation for our two notions of equality: We try to stay close to the Agda formalization and denote the internal equality by $x \equiv_A y$ and the (outside) definitional equality by $x = y : A$. Many authors prefer them the other way round, i.e. $x \underset{A}{=} y$ for internal equalities and $x \equiv y : A$ for the definitional equality

B as the set $\mathrm{fiber}_f(b) \coloneqq \sum_{a:A} f(a) \equiv_B b$, i.e. all the elements of a that map to b together with a proof of equality.

W-Types

Another important addition to our type theories are so-called W-types. W-types contain recursive well-founded trees where the shape of the branching is determined by a type. For this, let A be a type and B a type family over A. Then we define the W-type $W_{x:A}B(x)$ together with its canonical elements that are of the form $\mathrm{sup}(a, f)$ where $a : A$ and $f : B(a) \to W_{x:A}B(x)$. Intuitively, $\mathrm{sup}(a, f)$ is a tree, where a determines the branching type $B(a)$. The element $\mathrm{sup}(a, f)$ has "$B(a)$-many" subtrees of the form $f(y)$ for $y : B(a)$. The elimination rule for $W_{x:A}B(x)$ lets us treat every element of $W_{x:A}B(x)$ as a canonical element of the form $\mathrm{sup}(a, f)$, where the elimination of all the $f(y)$ is already defined. For the sake of convenience, we define $\overline{\mathrm{sup}(a, f)} \coloneqq a$ and $\widetilde{\mathrm{sup}(a, f)} \coloneqq f$ to recover the parts of a W-tree. We will use W-types later to define the type of iterative multisets and iterative sets. One important concept, that we make use of is the notion of subtrees: If u and v are W-trees in $W_{x:A}B(x)$, then we define the type of subtree-relations between u and v by

$$\left(u \in^W v\right) \coloneqq \sum_{x:\overline{v}} \tilde{v}(x) \equiv_{W_{y:A}B(y)} u$$

i.e. as the type of indices of x that the subtree v is attached to.

Universes

Lastly, an important concept for using dependent type theory as a foundation for mathematics are universes. In the simplest case, we would add a single new type $\mathcal{U}$ and define that all of its elements are types. We would then need to add more rules to specify that the universe is closed under all of the type constructions above. Similarly to Russel's Paradox, one cannot have $\mathcal{U} : \mathcal{U}$. To remedy this, one can add an entire hierarchy of universes $\mathcal{U}_0 : \mathcal{U}_1 : \ldots$ and provide mechanims to raise the universe level of types. Since many practically relevant proofs can be formulated in a universe agnostic way and manually dealing with universe levels can be tedious, some proof assistants elect to use a knowingly inconsistent meta-theory (at least in their prototype stages) or provide extensions that flatten the universe hierarchy into a single universe, sacrificing consistency for convenience. Standard Agda assumes one universe for every natural number and provides the type `Lift` to raise a type into a higher universe level. Since the discussion of iterative sets (apart from universe codes) can be formulated entirely without mention of universe levels, we follow D. Gratzer, H. R. Gylterud, A. Mörtberg, and E. Stenholm [7] and formulate Chapter 2

using only a single universe $\mathcal{U}$ for the sake of clarity. The formalization in Chapter 3, however, is written maximally universe polymorphic.

1.2. Homotopy Type Theory and Univalence

In traditional (intuitionistic) logic, any two proofs of the same proposition are identified. Treating propositions as types now suggests that multiple distinct proofs of the same statement (type) can exist and type theory even provides ways to identify and distinguish them. Similarly, in traditional logic there is usually just one type of equality, and two objects can only be equal in at most one way, i.e. if their extensions are the same. Our dependent type theory now provides a new perspective that allows to ask the question whether two objects $x, y : A$ can be (propositionally) equal in more than one way, i.e. if there are two distinct elements of the equality type $x \equiv_A y$, and indeed, it turns out that this is consistent with our type theory. This idea is one of the key step leading to the development of homotopy type theory (HoTT), a fruitful endeavour to equip mathematics with a new constructive foundation [20].

The central observation of HoTT is that types together with equality on them behave exactly like homotopy types and paths in homotopy theory: e.g. they connect two endpoints, are invertible and composable. Additionally, equalities between equalities, i.e. $p \equiv_{x \equiv y} q$ behave just like homotopies, revealing additional higher structure analogous to higher homotopies. The similarity to homotopy theory enables extensive applications of category theory in the form of categorical semantics, ranging from the early groupoid model [12] to modern ∞-categorical semantics (see e.g. E. Riehl [16]).

In this homotopical interpretation, Π-types can be viewed as fibrations and Σ-types as total spaces. This motivates the important concept of transport, which we will later make frequent use of. Suppose that A and B are types such that $p : A \equiv_{\mathcal{U}} B$. We would therefore expect that any element $x : A$ has some counterpart in B. And indeed, we can easily define such an element, which we will call the transport of x via p, by using the J-rule as $\mathsf{transport}_{\mathsf{refl}_A}(x) :\equiv x : A$. The J-rule guarantees that such an element $\mathsf{transport}_p(x) : B$ exists for all $p : A \equiv_{\mathcal{U}} B$ and the computation rule makes sure that transporting via refl_A is actually the same as the identity function. A special case of transport is (internal) substitution: If A is a type such that $x, y : A$ and $p : x \equiv_A y$ and B is a type family such that $w : B(x)$, then the equality p induces an equality $\mathsf{cong}_B(p) : B(x) \equiv_{\mathcal{U}} B(y)$ which we can transport over. For the sake of convenience we will call this kind of transport substitution[2] (not to be confused with the outside syntactical notion of substitution), and define $\mathsf{subst}^B_p(x) :\equiv \mathsf{transport}_{\mathsf{cong}_B(p)}(x)$.

[2] following the terminology of the agda-cubical library [25]

Depending on how much non-trivial higher structure a type has, one distinguishes different so-called homotopy levels (henceforth called h-levels). The most trivial types, contractible types, have h-level 0[3]. Contractible in this sense simply means that the type has precisely one inhabitant, i.e. $\sum_{x:A} \prod_{y:A} x \equiv_A y$. All higher h-levels are defined by saying that a type A has h-level $n+1$ if for all $x, y : A$, the type $x \equiv_A y$ has h-level n.

Notable examples are homotopy propositions (henceforth h-propositions), which have h-level 1. A type is an h-proposition if and only if all of its elements (potentially non-existent) are equal, i.e. $\prod_{x,y:A} x \equiv_A y$. Next are the homotopy sets (henceforth h-sets) with level 2 that have the property that any two equalities between elements are the same. The names "h-propositions" and "h-sets" highlight the fact that they behave like ordinary propositions (i.e. at most one proof) and sets (only one notion of equality) in ordinary set-based mathematics. In our later study of iterative sets, the notions of h-propositions and h-sets will be important because they let us ignore any higher structure that complicates proofs or makes them outright impossible.

In category theory, an isomorphism between two object is a strong property that states that the two objects cannot be distinguished by categorical means, which lets us effectively treat two object as the same to all intents and purposes. An isomorphism in category theory is simply an invertible arrow. In HoTT, where arrows correspond simply to functions, the direct translation leads to the notion of quasi-equivalence[4], which is defined for types A and B, as well as a function $f : A \to B$ as follows:

$$\text{isQEquiv}(f) :\equiv \sum_{g:B\to A} \left(\prod_{x:A} g(f(x)) \equiv_A x \right) \times \left(\prod_{y:B} f(g(y)) \equiv_B y \right)$$

A specific function should be an isomorphism in at most one way. Unfortunately, it turns out that the type $\text{isQEquiv}(f)$ can, in general, have higher structure and is thus in general not an h-proposition. The correct (desirable) notion of isomorphism between types is the one equivalence, which is for A, B and f as before defined as follows:

$$\text{isEquiv}(f) :\equiv \left(\sum_{g:B\to A} \prod_{x:A} g(f(x)) \equiv_A x \right) \times \left(\sum_{h:B\to A} \prod_{y:B} f(g(y)) \equiv_B y \right)$$

One can show that every quasi-equivalence gives rise to an equivalence and conversely. However, since $\text{isEquiv}(f)$ is provably always an h-proposition, the two notions are in general not equivalent. For types A and B we introduce the following type of (proofs of) equivalences between A and B as follows:

[3] h-levels correspond precisely to truncation levels, simply translated by two, i.e. whereas truncation levels start from -2, h-levels start from 0

[4] counter-intuitively, this property is called "isomorphism" in the agda-cubical library

$$A \simeq B \coloneqq \sum_{f:A\to B} \mathrm{isEquiv}(f)$$

Note that, if $e : A \simeq B$ and $x : A$, then we often write $e(x) : B$ to mean $\pi_1(e)(x) : B$ as a notational shortcut for the sake of clarity.

Having this notion of equivalence (categorically: isomorphism), one would expect that, similarly to category theory, any two equivalent types are also equal as types, i.e. with respect to the equality type of the universe. In ordinary HoTT, this is unfortunately not provable. However, since this is a highly desirable property with a deep categorical motivation, one often adds this as an additional rule to the type theory. This rule, usually stated as postulating an element

$$\mathrm{ua}_B^A : (A \simeq B) \simeq (A \equiv_{\mathcal{U}} B)$$

is called the axiom of univalence and was originally introduced by Voevodsky [28]. It can be summarised by the slogan that isomorphic obects can be identified, a practice that is prevalent in most (if not all) fields of mathematics, even though the set theoretic foundations that these fields are usually carried out in, do not, strictly speaking, support it. Univalent foundations (i.e. HoTT together with the Axiom of Univalence) has proven to be a fruitful field of study and a suitable framework for constructive mathematics.

1.3. Cubical Type Theory

The Axiom of Univalence, however has one major flaw: since the term ua_B^A does not have a general eliminator, it does not compute. In practice, this means that any term containing ua_B^A gets "stuck" in the sense that it does not further reduce. E.g., if $1_{\mathbb{N}} : \mathbb{N} \simeq \mathbb{N}$ is the identity equivalence on $\mathbb{N}$, then $\mathrm{transport}_{\mathrm{ua}_B^A(1_{\mathbb{N}})}(0) : \mathbb{N}$ should "morally" reduce to 0, but because of the univalence axiom, the term does not further reduce. This is also a counter-example to canonicity, the property requiring every term of type $\mathbb{N}$ to reduce to a canonical natural number, i.e. a finite successor of 0.

Cubical type theory [5] tries to remedy this problem with a new approach: So far, what we called HoTT was still essentially ordinary MLTT, we just chose to use the familiar language of homotopy theory to guide our intuition. Crucially, from a philosophical standpoint, the rules of type theory come first and are derived from the familiar notions of set theory. E.g. there are no rules stating that an equality is actually a path. In cubical type theory, this notion of path is actually included in the rules of the type theory.

To be precise, instead of the J-rule we add a new dedicated interval type I that has two connected canonical elements $i_0, i_1 : \mathbb{I}$, corresponding to the two endpoints of the interval. Moreover, for every type family $A : \mathbb{I} \to \mathcal{U}$ as well as $x : A(i_0)$ and $y : A(i_1)$, we define a type of heterogenous paths from x to y via the family A, denoted $\mathrm{PathP}_A(x, y)$.

Then ordinary equality types are simply heterogenous path types where the the type family A is constant, i.e.

$$x \equiv_A y \coloneqq \mathrm{PathP}_{\lambda\_.A(\_)}(x, y)$$

By having multiple interval variables in our context, we can build more complex homotopical structures such as homotopy squares. Depending on the exact rules in the type theory, the interval has additional structure, e.g. it might be a lattice or a De-Morgan algebra equipped with $\wedge$ and $\vee$, together with an involutive negation $\sim$. For an in-depth introduction, see A. Mörtberg [14].

In cubical type theory, equalities therefore become actual definable paths. E.g. the reflexivity path for $x : A$ can be defined as $\mathrm{refl}_x \coloneqq \lambda\_.x$ and if $p : x \equiv_A y$, then one might define a path $q : y \equiv_A x$ by setting $q \coloneqq \lambda i.p(\sim i)$. Similarly, the J-rule is also provable, the proof involving a simple transport over the type family. Using some more complex rules, one can also prove many other properties of paths including higher homotopies.

The important advantage of cubical type theory over standard HoTT is that functional extensionality and univalence are provable. While the proof of function extensionality is a simple switch of two input arguments,

$$\mathrm{funExt}(f, g) : \left(\prod_{x:A} f(x) \equiv_B g(x)\right) \to f \equiv g$$
$$\mathrm{funExt}(f, g) = \lambda P.\lambda i.\lambda x.P(x)(i),$$

the proof of the univalence axiom involves so-called glue types that serve to "glue" the two faces (i.e. the types A and B) together via the particular equivalence.

The fact that cubical type theory includes the homotopical way of thinking inside the type theory makes it particularly useful for reasoning about abstract mathematical fields like algebraic geometry and algebraic topology and considerable effort has been carried out to formalize important results in cubical proof assistants like Cubical Agda [25].

For this thesis, the hope was that the fact that function extensionality and univalence are provable theorems rather than axioms would improve the computational behaviour, which would ultimately simplify the statements enough to finalize the conjectured proof. However, even though cubical type theory provides some important advantages over standard HoTT, having explicit paths inside the type theory can make reasoning about equality more complex. One major issue is the fact that transport behaves unexpectedly, most notably it is the case that $\mathrm{transport}_{\mathrm{refl}_A}(x)$ does not reduce definitionally to x. While the equality $\mathrm{transport}_{\mathrm{refl}_A}(x) \equiv_A x$ is provable, having to carry around one extra layer of $\mathrm{transportRefl}_x$ every time a transport is used becomes tedious. Most notably, since the J-rule is essentially a transport in cubical type theory, this means that, in particular, $J^A_a(C(y, q), q, \mathrm{refl}_a)$ does not reduce to q, as opposed to MLTT and thus standard HoTT.

1.4. Models of Type Theory

An important part of type theory, as with other logical frameworks, is the question of what kind of models they admit. One major requirement for any kind of model of type theory is to respect the concept of variable bindings and substitution, which is much more prominent and delicate in type theory compared to ordinary set theory. For dependent type theories, in particular, where even types are allowed to depend on particular contexts, a sophisticated approach is necessary to capture the behaviour of substitution.

The important observation is that the type constructions behave functorially with respect to substitution and the language of category theory is particularly suitable to enforce the coherence conditions introduced by variable binders and substitution. In the past, many different models of MLTT and HoTT have been proposed, each with inherent advantages and disadvantages. In this thesis, we will work with so-called *Categories with Families* (CwF) [6], see M. Hofmann [11] for an overview. While many types of models work with terms as sections of certain arrows, terms $\Gamma \vdash x : A$ in CwF are actual elements of the set of terms of A in the context Γ. Working with proof assistants, explicit elements can often have better definitional behaviour, which is also the reason why they were chosen by D. Gratzer, H. R. Gylterud, A. Mörtberg, and E. Stenholm [7] for their formalization. We will follow their unrolled definition in general, but will make slight adaptations later to make it more suitable for use in a cubical proof assistant in Chapter 3:

Definition 1.1. *A Category with Families consists of a category $\mathcal{C}$ of contexts and substitutions with an initial object $\langle\rangle$, called the* empty context*, and the following structure:*

- *a presheaf of types:* $\mathrm{Ty} : \mathcal{C}^{\mathrm{op}} \to \mathrm{hSet}$
- *a presheaf of terms:* $(\int \mathrm{Ty})^{\mathrm{op}} \to \mathrm{hSet}$
- *a functor context extension functor* $- \,.\, - : \int \mathrm{Ty} \to \mathcal{C}$
- *for every* $\Delta, \Gamma : \mathcal{C}$ *and* $A : \mathrm{Ty}(\Gamma)$ *(i.e.* $(\Gamma, A) : \int \mathrm{Ty}$*) an equivalence*

$$\mathcal{C}(\Delta, \Gamma \,.\, A) \simeq \sum_{\sigma : \mathcal{C}(\Delta,\Gamma)} \mathrm{Tm}(\Delta, (A \cdot \sigma)))$$

natural in Δ.

Note that hSet here is the category of h-sets and $\int \mathrm{Ty}$ denotes the category of elements, whose elements are pairs (Γ, A) such that $\Gamma : \mathcal{C}$ and $A : \mathrm{Ty}(\Gamma)$ and whose arrows $(\Gamma, A) \to (\Delta, B)$ are arrows $f : \Gamma \leftarrow \Delta$ such that $\mathrm{Ty}(f)(A) \equiv_{\mathrm{hSet}} B$. We also denote the presheaf actions of a substitution $\sigma : \Gamma \to \Delta$ on a type $B : \mathrm{Ty}(\Delta)$ by $B \cdot \sigma : \mathrm{Ty}(\Gamma)$ and on a term $b : \mathrm{Tm}(\Delta, B)$ by $b[\sigma] : \mathrm{Tm}(\Gamma, (B \cdot \sigma))$, where we suppress the fact that the acting arrow is strictly speaking $(\sigma, \mathrm{refl}_{B \cdot \sigma})$. With the same variables, we also define

$$\langle \sigma, B \rangle : \mathcal{C}(\Gamma \,.\, (B \cdot \sigma), \Delta \,.\, B)$$

for the substitution that simultaneously substitutes Γ to Δ and $B \cdot \sigma$ to B via σ, and, for $a : \mathrm{Tm}(\gamma, (B \cdot \sigma))$ the substitution $\langle \sigma, a \rangle : \mathcal{C}(\Gamma, (\Delta \,.\, B))$ that extends σ by a.

Any Category with Families can interpret a basic dependent type theory (i.e. without any added typing rules). The idea is that the dependency of types and terms is precisely captured by the functoriality conditions of the presheaves Ty and Tm as well as the context extension functor $- \,.\, -$. While the initial object $\langle\rangle$ is required to model all the types and terms that exist in the empty context, the natural equivalence then guarantees that context extension behaves as expected. For the details, see e.g. C. Angiuli and D. Gratzer [3].

In order to be able to interpret the additional typing rules introduced in Section 1.2, the CwF has to be equipped with additional structure. Similarly to CwF themselves, this additional structure needs to respect substitution. In the following, we will introduce the notion of Σ-structures that allows CwF to interpret Σ-types. For this, we will follow the D. Gratzer, H. R. Gylterud, A. Mörtberg, and E. Stenholm [7], and change the definition slightly in Chapter 3 to accommodate our cubical metatheory. For a detailed discussion of how to define the structure for other type formers, see e.g. Chapter 6 of C. Angiuli and D. Gratzer [3].

Definition 1.2. *Let $\mathcal{C}$ be a Category with Families. A Σ-structure on Σ consists of*

- *an operation* $\mathrm{sig} : \prod_{\Gamma:\mathcal{C}} (\mathrm{Ty}\,\Gamma \to \mathrm{Ty}(\Gamma \,.\, A) \to \mathrm{Ty}\,\Gamma)$ *natural in* Γ
- *for all contexts* $\Gamma : \mathcal{C}$ *and all types* $A : \mathrm{Ty}\,\Gamma$, $B : \mathrm{Ty}(\Gamma \,.\, A)$ *an equivalence*

$$\mathrm{Tm}(\Gamma, \mathrm{sig}_\Gamma(A, B)) \simeq \sum_{a:\mathrm{Tm}(\Gamma,A)} \mathrm{Tm}(\Gamma, B \cdot \langle \mathrm{id}, a \rangle)$$

natural in Γ.

Note that the operation sig being natural in Γ means that for all $A : \mathrm{Ty}(\Gamma)$, $B : \mathrm{Ty}(\Gamma \,.\, A)$ and all substitutions $\sigma : \Delta \to \Gamma$ it holds that

$$\mathrm{sig}_\Gamma(A, B) \cdot \sigma \equiv_{\mathrm{Ty}(\Delta)} \mathrm{sig}_\Delta(A \cdot \sigma, B \cdot \langle \sigma, A \rangle)$$

1.5. The Proof Assistant Agda

Proof assistants come in various shapes and sizes, offering a range of capabilities and workflows. An important feature is the ability to verify the correctness of a given proof. Usually this proof, manually or otherwise generated, has to be written in a particular format that is often structured like a computer program and is (usually) particular to the proof assistant in use. What distinguishes proof assistants from proof checkers is the fact that proof assistants provide various interactive tools to help the user generate a proof in the first place. This includes, splitting up goals into smaller subgoals, stating the exact goals that are left to prove and providing a list of assumptions and (potentially useful) known theorems for each goal. Some proof assistants also offer proof tactics that can automate various simple but tedious tasks like repeatedly applying equational rules (e.g. associativity, distributivity) automatically. Some proof assistants also provide

the functionality to fill simple goals automatically, though this is usually limited to very simple proofs.

Due to the good computational properties, notably the Curry-Howard Correspondence, many proof assistants are based on type theory. As such, the proof languages of many proof assistants are in fact worthy programming languages in their own right. This is also true for Agda, which can be seen as a modern dependently typed functional programming language in the tradition of Haskell. The type theory of agda includes all the standard type rules of MLTT introduced in Section 1.2 as well as a countable hierarchy of universes `Set ℓ`. However, it offers many extensions that introduce new type rules, or even change the existing ones, like for coinductive types, guarded type theory, sized types as well as a mode for cubical type theory. While Agda only provides limited support for tactics, it enforces only a very liberal syntax, allowing, in addition to prefix operators, also infix, postfix and even arbitrary mixfix operators with custom binding strength. This greatly improves the readability of code.

Working with Agda

The main usage of Agda is through interaction points called "holes", written as `{!!}` in the source code. In the final proof, each of these holes need to be filled with a term of the appropriate type. During development, Agda can give the user helpful information about each hole, notably the exact type of the goal and all active variable bindings (i.e. the current type context). The types can be displayed in three levels of normalization, ranging from the first being only expanded as little as possible to the third being full normalization, striking a balance between readability and wealth of detail. The user will try to refine these holes more and more, with the aim of filling every hole eventually. At every step, Agda makes sure that refinements and terms are well-typed, however, sometimes Agda cannot infer that a term is well-typed is correct, e.g. involving some unknown implicit arguments. In this case, Agda will usually allow the user to proceed, but it will highlight the corresponding code and prompt the user to add some additional annotation.

Terms are usually defined by pattern matching, e.g. we can define addition for natural numbers by setting:

```
_+_ : ℕ → ℕ → ℕ
zero     + n = n
(succ m) + n = succ (m + n)
```

This defines a infix operator `+`, that takes two natural numbers (curried), and outputs another natural number. Addititional features of the syntax include telescopes, that simplify the notation for a sequence of dependent functions (we can write `(x : A) (y z : B) → C` instead of `(x : A) → (y : B) → (z : B) → C`) and implicit arguments if

we do not wish to have to specify the argument ourselves, but rather let Agda determine in (for an implicit argument $x : A$ we write `{x : A}`).

Standard Agda also supports equality types and the J-rule by allowing pattern matching on refl. E.g. the proofs of symmetry and transitivity (path composition) can be implemented as:

```
sym : x ≡ y → y ≡ x
sym refl = refl
```

and

```
_·_ : x ≡ y → y ≡ z → x ≡ z
refl · refl = refl
```

Agda provides two ways of adding structure to a type, either traditionally by use of Σ-type and by use of records (corresponding to structs and classes in other programming languages). While Σ-types have better behaviour for when two structures need to be compared, records can significantly increase the legibility of the corresponding proofs. The downside of records is that they can sometimes normalize in unexpected ways, which can actually make some goals harder to read.

Working with Cubical Agda

Cubical Agda is a special mode of agda that enables a cubical metatheory. For this, definitional J-rule gets replaced by, among others, an atomic interval `I` as well as a dependent path type `PathP`. The main effort to formalize mathematics in Cubical Agda is carried out in the agda-cubical library [25], though other libraries like the `1lab` [26] exist. The formalization in this thesis is built upon the agda-cubical library and the correponding source code is going to be added to the upstream library after some clean-up effort.

Due to the cubical metatheory, particular equalities like refl as well as the J-rule, function extensionality and univalence become provable. In the following, we will demonstrate how to define refl, how to invert paths as well as prove function extensionality. For the other desirable properties, see the agda-cubical library and path symmetry. The reflexivity path refl_x is simply defined as the path that is constantly x, i.e.:

```
refl : {A : Type} {x : A} → x ≡ x
refl i = x
```

The proof of symmetry is similarly easy and conceptually straightforward as it merely involves walking the path in the opposite direction, i.e.

```
sym : {A : Type} {x y : A} → x ≡ y → y ≡ x
sym p i = p (~ i)
```

For the proof of function extensionality, we simulate how the actual development process could look like. For the sake of simplicity, assume that the type A and the type family B over A have already been declared as variables. We first generate the signature and skeleton of the proof `funExt`:

```
funExt : (f g : (x : A) → B x) → ((x : A) → f x ≡ g x) → f ≡ g
funExt = {!!}
```

Then Agda will tell us the following information for the hole

```
Goal: (f g : (x : A) → B x) → ((x : A) → f x ≡ g x) → f ≡ g
————————————————————————————————————————————————————————
B  : A → Type ℓ'
A  : Type ℓ
```

We can make Agda introduce the two arguments into the context, which leaves us with:

```
funExt : (f g : (x : A) → B x) → ((x : A) → f x ≡ g x) → f ≡ g
funExt f g P = {!!}
```

and the new goal:

```
Goal: f ≡ g
————————————————————————————————————————————————————————
P  : (x : A) → f x ≡ g x
g  : (x : A) → B x
f  : (x : A) → B x
```

In Cubical Agda, we can introduce equalities similar to λ-expressions and Agda will internally verify that the endpoints are correct. Therefore, we can introduce an interval parameter i to the context:

```
funExt : (f g : (x : A) → B x) → ((x : A) → f x ≡ g x) → f ≡ g
funExt f g P i = {!!}
```

which gives the updated goal:

```
Goal: (x : A) → B x
——— Boundary (wanted) ——————————————————————————————
i = i0 ⊢ λ x → f x
i = i1 ⊢ λ x → g x
————————————————————————————————————————————————————————
i  : I
P  : (x : A) → f x ≡ g x
g  : (x : A) → B x
f  : (x : A) → B x
```

Agda therefore tells us that our goal has a dependent function type and that the term needs to be `f` on the one endpoint and `g` on the other. Since our goal is a function type, we can introduce another variable to our context:

```
funExt : (f g : (x : A) → B x) → ((x : A) → f x ≡ g x) → f ≡ g
funExt f g P i x = {!!}
```

which results in the following context:

```
Goal: B x
——— Boundary (wanted) ——————————————————————————————
i = i0 ⊢ f x
i = i1 ⊢ g x
————————————————————————————————————————————————————————
x  : A
```

```
i  : I
P  : (x₁ : A) → f x₁ ≡ g x₁
g  : (x₁ : A) → B x₁
f  : (x₁ : A) → B x₁
```

This tells us that we essentially need to provide a path from `f x` to `g x` inside `B x`. Luckily, `P x` provides just that and we can finally fill the hole as follows:

```
funExt : (f g : (x : A) → B x) → ((x : A) → f x ≡ g x) → f ≡ g
funExt f g P i x = P x i
```

Agda checks that the type and boundary constraints are satisfied and notifies us that everything is correct, `*All Done*`.

2. Iterative Sets as Models of Type Theory

In this chapter, we will first define the iterative multisets based on the previously introduced W-types and then show how to obtain a notion of iterative sets by restricting the possible indexing functions in a suitable way. Next, we will show that the type of iterative sets forms a type-theoretic universe of sets which is itself a set. We then end this chapter by showing that this universe is in fact a model of dependent type theory by exhibiting a corresponding Categories-with-Families structure. The proofs in this chapter are paper-based and meant to give a human-understandable treatment of the material while the computer formalization can be found in Chapter 3. We will generally follow the work of D. Gratzer, H. R. Gylterud, A. Mörtberg, and E. Stenholm [7], but make slight changes to improve the clarity in some places.

The notion of iterative sets was born from the endeavour to develop a constructive set theory. In contrast to other approaches working with a constructive metatheory and removing or replacing non-constructive axioms, P. Aczel [1] proposed to work inside type theory which he considers more fundamental than any set-theoretic approach and which provides constructivity out of the box. Aczel's method provides a bottom-up approach of building sets that is similar to the cumulative hierarchy of sets in set theory. The idea of the cumulative hierarchy is that sets at some level κ can only contain the sets that are "already defined" at lower levels. For iterative multisets he follows a similar approach by using W-types. This endeavour proved to be fruitful, however, in modern homotopy type theory where we allow higher equality structure it turns out that Aczel's constructive set theory is in fact a constructive multiset theory D. Gratzer, H. R. Gylterud, A. Mörtberg, and E. Stenholm [7].

2.1. From Iterative Multisets to Iterative Sets

In analogy to the cumulative hierarchy, we view an iterative set as a (small) type indexed family of some previously defined iterative sets. This means that in essence, an iterative multiset X consists of two pieces of data: a small type A and a function f choosing one (already defined) iterative multiset for every element of A. The type A can be seen as a collection of indices and the iterative sets in the image of f as the elements of X. This works since W-types are by construction well-founded. This construction gives iterative multisets rather than iterative sets, since we place no restrictions on the function f and in general there might be multiple elements of A that map to a given element. In contrast to ordinary set-theory based multiset theory, this approach is more expressive since the

fiber over an element Y, which corresponds to the classical notion of multiplicity, can have some potentially interesting higher structure.

Definition 2.1. *We define the type V^∞ of iterative multisets as $V^\infty \coloneqq W_{A:\mathcal{U}} A$, retaining the definitions of the operations $\overline{\cdot}$ and $\tilde{\cdot}$ of ordinary W-types as defined in Section 1.1. Moreover, we define the multiplicity of some iterative set X in Y as the type*

$$X \in^\infty Y \coloneqq \mathrm{fiber}_{\tilde{Y}}(X)$$

Some simple examples include:

- the empty multiset $\varnothing^\infty \coloneqq \sup(0, \lambda())$, where $\lambda()$ denotes the unique empty function out of the empty type,
- the singleton multiset $\{x\} \coloneqq \sup(1, \lambda y.x)$, i.e. with the constant function x,
- the multiset with two distinct elements x and y: $\{x, y\} \coloneqq \sup(2, f)$, where $f(0_2) \coloneqq x$ and $f(1_2) \coloneqq y$,
- the multiset with one element with multiplicity 2: $\{x, x\} \coloneqq \sup(2, \lambda y.x)$, i.e. with the constant function x.

Since our aim is to develop a theory of iterative sets rather than multisets, we need to restrict the construction in a suitable way. The approach of D. Gratzer, H. R. Gylterud, A. Mörtberg, and E. Stenholm [7] is to require the functions f to be embeddings. Since embeddings have h-propositional fibers, this restriction amounts to enforcing that every element only appears once in the image. To capture this, we define a type for hereditarily iterative sets:

$$\mathrm{isIterativeSet} : \prod_{X:V^\infty} \mathrm{isEmbedding}(\tilde{X}) \times \prod_{a:\overline{X}} \mathrm{isIterativeSet}(\tilde{X}(a))$$

Since being an embedding is an h-proposition, it is straightforward to show by induction that the same holds for being a hereditarily iterative set:

Lemma 2.2. *For every iterative multiset X the type* $\mathrm{isIterativeSet}(X)$ *is an h-proposition.*

Together, an iterative set is then an iterative multiset with the additional property that each of the element selection functions is an embedding, i.e. we have the following definition:

Definition 2.3. *The type V^0 of iterative sets is defined as the following Σ-type:*

$$V^0 \coloneqq \sum_{x:V^\infty} \mathrm{isIterativeSet}(x)$$

We then generalize the definitions of the operations $\overline{\cdot}$ and $\tilde{\cdot}$ as well as the element relation from V^∞ to V^0 by setting for all $x, y : V^0$:

$$\overline{x} \coloneqq \overline{\pi_1(x)}$$

$$\tilde{x} \coloneqq \lambda a.\left(\widetilde{\pi_1(x)}(a), \pi_2(\pi_2(x))(a)\right)$$

$$\left(x \in^0 y\right) \coloneqq \mathrm{fiber}_{\tilde{y}}(x)$$

Before we start to work on the main results of this section, we first note an immediate consequence of Lemma 2.2 that is due to the general fact that for any type A and any family $B : A \to \mathcal{U}$ of h-propositions (i.e. for every $x : A$, the type Bx is an h-proposition), the first projection $\pi_1 : \sum_{x:A} B(x) \to A$ is an embedding:

Corollary 2.3.1. *The first projection $\pi_1 : V^0 \to V^\infty$ is an embedding, i.e. V^0 is a subtype of V^∞.*

Even though we place a sharp restriction on the shapes of the functions $\tilde{x}$, it would seem that, a priori, the indexing type $\overline{x}$ could be arbitrarily complicated and not necessarily set-like. However, we will show in the following, that all the types $\overline{x}$ are in fact h-sets, which lets us claim in the next section that V^0 encodes a universe of sets. This is a direct consequence of another curious result that V^0 is actually itself an h-set. The latter stands in contrast to the type hSet of h-sets which encodes another universe of sets, but which is not itself a set. Our approach here deviates significantly from the one in D. Gratzer, H. R. Gylterud, A. Mörtberg, and E. Stenholm [7], since their proofs are not easily replicable in a cubical setting.

These two results follow more or less directly from the following more general results, which are due to H. R. Gylterud [9], but for which we utilize the names as defined in the agda-cubical library [25].

Theorem 2.4 (Fibration Identity Principle [25]). *Let B be a type. Then the type* $\mathrm{Fib}(B)$ *of fibrations over B is defined as $\sum_{E:\,\mathrm{Type}} E \to B$. For any $(E, f), (F, g) : \mathrm{Fib}(B)$, it holds that*

$$\left((E, f) \equiv_{\mathrm{Fib}(B)} (F, g)\right) \simeq \left(\prod_{b:B} \mathrm{fiber}_f(b) \simeq \mathrm{fiber}_g(b)\right)$$

In the context of embeddings, FIP can be made precise in the following way:

Corollary 2.4.1 (Embedding Identity Principle [25]). *Let B be a type. Then the type* $\mathrm{Emb}(B)$ *of embeddings into B is defined as $\sum_{E:\,\mathrm{Type}} E \hookrightarrow B$. Then we have that* $\mathrm{Emb}(B)$ *is an h-set and for any $(E, f), (F, g) : \mathrm{Emb}(B)$ it holds that*

$$\left((E, f) \equiv_{\mathrm{Emb}(B)} (F, g)\right) \simeq \left(\prod_{b:B} \mathrm{fiber}_f(b) \to \mathrm{fiber}_g(b)\right) \times \left(\prod_{b:B} \mathrm{fiber}_g(b) \to \mathrm{fiber}_f(b)\right)$$

Proof. The first part follows from Theorem 2.4 and the fact that, if f and g are embeddings, their fibers are h-propositions and both the equivalence type of two h-propositions

as well as dependent families of propositions are again propositions. The second part, i.e. the fact that all identity types are propositional, then follows from the fact that

$$(S \simeq T) \simeq (S \to T) \times (T \to S)$$

if one (and consequently both) of S and T are h-propositions as well as the fact that currying is an equivalence. ■

To apply Theorem 2.4 and Corollary 2.4.1, we need a way to convert between V^∞ and $\mathrm{Fib}(V^\infty)$ as well as between V^0 and $\mathrm{Emb}(V^0)$. Fortunately the correspondence is straightforward:

Definition 2.5. *We define the functions*[5]

$$V^\infty \underset{\text{fromFib}}{\overset{\text{toFib}}{\rightleftarrows}} \mathrm{Fib}(V^\infty) \quad \text{and} \quad V^0 \underset{\text{fromEmb}}{\overset{\text{toEmb}}{\rightleftarrows}} \mathrm{Emb}(V^0)$$

as follows:

$$\mathrm{toFib}(x) :\equiv (\overline{x}, \tilde{x}) \qquad \mathrm{fromFib}(A, f) :\equiv \sup^\infty(A, f)$$
$$\mathrm{toEmb}(x, p) :\equiv (\overline{x}, (\tilde{x}, \_)) \qquad \mathrm{fromEmb}(A, (f, p)) :\equiv (\sup^\infty(A, \pi_1 \circ f), \_)$$

where we omit the straightforward proofs of the embedding property of $\tilde{x}$ as well as the property that $\sup^\infty(A, \pi_1 \circ f)$ is an iterative set, which can be extracted from the corresponding properties in V^0 and $\mathrm{Emb}(V^0)$.

An important feature of the definition of fromEmb is that the indexing type of the resulting iterative set is definitionally equal to the supplied type, i.e. $\widetilde{\mathrm{fromEmb}(A, \_)} = A$. This will allow us later to easily see that some iterative set "definitionally encodes" a type.

It is also easy to see that these correspondences are in fact equivalences. For V^∞ the necessary equalities even hold definitionally (i.e. by reduction), while the ones for V^0 hold by reflexivity as well as the fact that isEmbedding and isIterativeSet are mere propositions. This shows the following:

Corollary 2.5.1. *It holds that $V^\infty \simeq \mathrm{Fib}(V^\infty)$ and $V^0 \simeq \mathrm{Emb}(V^\infty)$ and the functions are given by Definition 2.5.*

With Corollary 2.5.1, we can now apply Theorem 2.4 as well as Corollary 2.4.1 to V^∞ and V^0:

Corollary 2.5.2. *Let $x, y : V^\infty$, then it holds that*

$$(x \equiv_{V^\infty} y) \simeq \left(\prod_{z : V^\infty} (z \in^\infty x) \simeq (z \in^\infty y) \right)$$

Intuitively, Corollary 2.5.2 states that two iterative multisets are equal if they have the same elements and with the same multiplicities, including all the higher structure

[5] Note that D. Gratzer, H. R. Gylterud, A. Mörtberg, and E. Stenholm [7] call the functions fromEmb and toEmb as $\sup^0$ and desup^0 respectively, since $\sup^0$ essentially functions like a constructor, though it is not generated by a polynomial functor [19]. In fact, they leave the type $\mathrm{Emb}(B)$ unnamed. We use different names in order to highlight the fact that we are applying the general principles of Theorem 2.4 and Corollary 2.4.1.

contained in the corresponding notion of multiplicity. In the case of iterative sets, the multiplicities are propositional, so the statement of Corollary 2.5.2 reduces to the following.

Corollary 2.5.3. *The type V^0 is an h-set and for all $x, y : V^0$ it holds that*

$$(x \equiv_{V^0} y) \simeq \left(\prod_{z:V^0} (z \in^0 x) \to (z \in^0 y)\right) \times \left(\prod_{z:V^0} (z \in^0 y) \to (z \in^0 x)\right)$$

In other words, Corollary 2.5.3 states that two iterative sets are equal if and only if they have the same elements. That this is a provable property highlights the constructive nature of iterative sets, which sets it apart from classical ordinary set theories like ZF, where extensionality has to be guaranteed by an axiom.

A consequence of this is now the following corollary that follows from the fact that types that embed into an h-set are itself h-sets.

Corollary 2.5.4. *For every $x : V^0$, it holds that $\overline{x}$ is an h-set.*

Corollary 2.5.4 now completes our motivation for calling V^0 the type of iterative **sets**: An iterative set x is a family of pairwise distinct iterative sets x_a indexed by an h-set $\overline{x}$. In particular the embedding property implies that two elements x_a and x_b are equal if and only if already their indices a and b are equal in $\overline{x}$.

Even though the following work will solely focus on the case of iterative sets, it is a valid question to ask what happens if we allow some higher structure for the element-hood types. In fact, H. R. Gylterud and E. Stenholm [10] explore an entire hierarchy of iterative multisets, in which iterative set at the very bottom. Their discussion involves the notion of so-called n-truncated maps, i.e. maps whose fibers are n-truncated. In essence, embeddings are precisely the 1-truncated maps, but this view enables us to consider also higher generalizations of iterative sets where the type of V^n of $n+1$-truncated[6] multisets is simply defined by the Σ-type $V^n := \sum_{x:V^\infty} \mathrm{isTruncated}_{n+1}(x)$.

2.2. Iterative Sets Form A Universe Of Sets

Shifting our viewpoint slightly, we can imagine that the structure of an iterative set x characterizes $\overline{x}$ as an h-set with a special property that makes it suitable for the previously discussed iterative construction. In this viewpoint, V^0 defines a single-sorted (i.e. Tarski-style) universe of indexing types, that we will also call V^0 for the sake of convenience. To highlight this fact, we will adopt the appropriate terminology and define $\mathrm{El}^0(x) := \overline{x}$ and call $\mathrm{El}^0(x)$ the decoding of x. Note that this decoding is not unique as an index set can index many different iterative sets (see below). The question about what kind of h-sets can be encoded by V^0 is answered in the following:

[6] We will bear with this somewhat unfortunate misalignment between n and $n+1$ as it is the notation used by D. Gratzer, H. R. Gylterud, A. Mörtberg, and E. Stenholm [7]

Theorem 2.6. *The following equivalences hold for the image of* El^0*:*

$$\left(\sum_{E:\,\mathcal{U}}\sum_{x:V^0}\mathrm{El}^0(x) \equiv_{\mathcal{U}} E\right) \simeq V^0 \simeq \mathrm{Emb}(V^0)$$

and since the latter are h-sets, so is the image.

Proof. The statement follows from Lemma 4.8.2 of The Univalent Foundations Program [20] that states that for every function $f : A \to B$ it holds that

$$\sum_{b:B}\mathrm{fiber}_f(b) \simeq A$$

The lemma follows with $f = \mathrm{El}^0$, noting that

$$\mathrm{fiber}_{\mathrm{El}^0}(E) = \sum_{x:V^0}\mathrm{El}^0(x) \equiv_{\mathcal{U}} E$$

definitionally for all $E : \mathcal{U}$. ∎

In other words, types and their ways of being encoded by some $x : V^0$ correspond precisely to types and the way they embed into V^0. Since the equivalence leaves the first components constant, we also have the slightly stronger statement that for every concrete type $E : \mathcal{U}$, the ways E can be encoded by some iterative set are equivalent to the ways E embeds into V^0.

Corollary 2.6.1. *Let E be a type. Then the following equivalence holds:*

$$\left(\sum_{x:V^0}\mathrm{El}^0(x) \equiv_{\mathcal{U}} E\right) \simeq \left(E \hookrightarrow V^0\right)$$

Proof. The proof follows from following general result: If $B : A \to \mathcal{U}$ and $C : A \to \mathcal{U}$ are two type families and

$$e : \sum_{x:A} B(x) \simeq \sum_{x:A} C(x)$$

is an equivalence such that keeps the first component constant, i.e. for all $(a, b) : \sum_{x:A} B(x)$ it holds that $\pi_1(e(a, b)) \equiv_A a$, then there is a fiberwise equivalence

$$\prod_{x:A} B(x) \simeq C(x)$$

This is itself a corollary of Lemma 4.7.6 of The Univalent Foundations Program [20]. The present corollary is then proven by instantiating with $A = \mathcal{U}$, $B(E) = \mathrm{fiber}_{\mathrm{El}^0}(E)$ as well as $C(E) = E \hookrightarrow V^0$, noting that the equivalence of Theorem 2.6 does indeed leave the first component constant. ∎

In particular, Theorem 2.6 and Corollary 2.6.1 confirm that calling the elements of V^0 iterative **sets** makes sense. The type V^0 has the curious property of being a universe of sets that is itself again a set.

It turns out that the universe V^0 is closed under the most common type theoretic constructions, provided that they are already available in the universe $\mathcal{U}$. In the remaining part of this section, we will show how these constructions can be implemented such that

the decoding is definitional in the following sense: For simple types, the decoding of the encoding will be definitionally equal to to the type. E.g. the type $\mathbb{N}$ of natural numbers, we will have an encoding $\mathbb{N}^0 : V^0$ such that $\mathrm{El}^0(\mathbb{N}^0) = \mathbb{N}$. For complex types, given a number of definitional encodings $x_i : V^0$ (i.e. x_i encodes $\mathrm{El}^0(x_i)$), we will define an iterative set whose decoding is definitionally equal to the construction applied to the $\mathrm{El}^0(x_i)$. E.g. for the coproduct, given two iterative sets x and y (definitionally encoding $\mathrm{El}^0(x)$ and $\mathrm{El}^0(y)$), we will define an iterative set $x +^0 y$ such that $\mathrm{El}^0(x +^0 y) = \mathrm{El}^0(x) + \mathrm{El}^0(y)$. We will generally follow the outline of D. Gratzer, H. R. Gylterud, A. Mörtberg, and E. Stenholm [7], but provide occasionally slightly altered proofs that better reflect our later formalization.

It should be noted that, for our convenience's sake, we will define the respective encodings using the function fromEmb as it allows us to prove the embedding property more easily in the computer formalization. This is the case because it lets us prove the embedding property for a function into V^0 instead of V^∞. Since V^0 is an h-set, any function to V^0 is an embedding if and only if it is injective, and proving injectivity (without any indirection) is often (at least slightly) easier. Another advantage of this notation is that we use fromEmb like a constructor in a way that is quite similar to sup and sup^∞ for ordinary W-types as well as V^∞. In fact, this is the reason why D. Gratzer, H. R. Gylterud, A. Mörtberg, and E. Stenholm [7] call fromEmb sup^0 (cf.[5]). When using fromEmb, we will also make use of the notational convention[7] where we write $\mathrm{fromEmb}(A, f)$ for a function $f : A \to V^0$, i.e. omitting the proof of the embedding property in the formula, and just mention how to prove said property in the accompanying text.

Empty Type, Unit Type and Booleans

The encodings of the empty type 0 and the unit type 1 are straightforward:

$$0^0 \coloneqq \mathrm{fromEmb}(0, f)$$

$$1^0 \coloneqq \mathrm{fromEmb}(1, g)$$

where $f : 0 \to V^0$ is the unique empty function and $g : 1 \to V^0$ is the function that sends the single element $* : 1$ to 0^0. Both functions are clearly embeddings and by definition we have $\mathrm{El}^0(0^0) = 0$ and $\mathrm{El}^0(1^0) = 1$, i.e. the universe V^0 contains the empty type and the unit type.

Note that instead of 0^0, we can also define the singleton $\{x\}^0$ for any iterative set x by $\{x\}^0 \coloneqq \mathrm{fromEmb}(1, h)$ where $h : 1 \to V^0$ sends the single element $* : 1$ to x, which is easily recognised as an embedding. In particular we have that $1^0 = \{0^0\}^0$. Since also $\mathrm{El}^0(\{x\}) = 1$, this shows in particular that encodings need not be unique.

The type 2 of Booleans can then be encoded by the iterative set

[7] i.e. abuse of notation

$$2^0 :\equiv \mathrm{fromEmb}(2, i)$$

where $i : 2 \to V^0$ sends the canonical elements $0_2 : 2$ to 0^0 and $1_2 : 2$ to 1^0. The injectivity is given by the fact that $0^0 \neq 1^0$ which in turn follows from the fact[8] that $0 \not\equiv_{\mathcal{U}} 1$. Again, by definition we have that $\mathrm{El}^0(2^0) = 2$. Similarly to the Unit type, we can also define for iterative sets x and y with $x \not\equiv_{V^0} y$ the unordered pair $\{x, y\}^0 :\equiv \mathrm{fromEmb}(2, j)$ where $j : 2 \to V^0$ maps $0_2 : 2$ to x and $1_2 : 2$ to y, which is injective as $x \neq y$, and therefore an embedding. The pair

Note that, as expected, by the the Embedding Identity Principle, Corollary 2.5.3, equality of singletons is just equality of the elements, i.e.

$$\left(\{x\}^0 \equiv_{V^0} \{y\}^0\right) \simeq (x \equiv_{V^0} y)$$

More interestingly, equality of unordered pairs is a choice of pairwise equality of elements, i.e.

$$\left(\{x, y\}^0 \equiv_{V^0} \{a, b\}^0\right) \simeq ((x \equiv_{V^0} a) \times (y \equiv_{V^0} b)) + ((x \equiv_{V^0} b) \times (y \equiv_0 a))$$

In contrast to ordinary classical set theory, where the axiom of pairing does not require the two elements to be distinct, here an unordered pair already includes the property that it has two distinct elements.

Similarly to 0, 1 and 2, any finite type n can be encoded by an[9] iterative set and we can choose the encoding to be definitional in the above sense. Similarly to before, defining the iterative set n^0 encoding the type n already includes a proof that n^0 actually contains n distinct elements. The equality of such n-tuples is then similarly given by a choice of component-wise equalities.

Natural Numbers

In order to encode the type $\mathbb{N}$ of natural numbers, we first need to define its elements. For this, we define the individual natural numbers via a modified version of the von-Neumann encoding. In this encoding, we define the successor x^+ of an arbitrary iterative set x as

$$S^0(x) :\equiv \mathrm{fromEmb}(\overline{x} + 1, k)$$

where the function k is defined recursively as

$$k : \overline{x} + 1 \to V^0$$
$$k(\mathrm{inl}(a)) :\equiv \tilde{x}(a)$$
$$k(\mathrm{inr}(*)) :\equiv \{x\}^0$$

[8] i.e. the inhabitant of the type $(0 \equiv_{\mathcal{U}} 1) \to 0$

[9] in fact many different

The function k is injective since $\tilde{x}$ is injective and also clearly $\{x\}^0$ is not an element of x, so it too is distinct from any values of $\tilde{x}$. This allows us now to define all the (von-Neumann encoded) natural numbers as follows:

$$f : \mathbb{N} \to V^0$$
$$f(0) \coloneqq 0^0$$
$$f(S(n)) \coloneqq S^0(f(n))$$

That the function f is an embedding follows from the fact that $f(n) \simeq \mathrm{Fin}(n)$, where $\mathrm{Fin}(n)$ is the type of all natural numbers less than n, and $\mathrm{Fin}(n) \simeq \mathrm{Fin}(m)$ implies $n \equiv m$. Therefore, we can encode the type $\mathbb{N}$ of natural numbers as follows:

$$\mathbb{N}^0 \coloneqq \mathsf{fromEmb}(\mathbb{N}, f)$$

and as required we have that $\mathrm{El}^0(\mathbb{N}^0) = \mathbb{N}$.

Ordered Pairs

While we have already exhibited the fact that the Booleans can be encoded by unordered pairs, it will be important later to also have a notion of ordered pair. For this we use the Norbert-Wiener Encoding[10]:

$$\langle x, y\rangle^0 \coloneqq \left\{\left\{0^0, \{x\}^0\right\}^0, \left\{\{y\}^0\right\}^0\right\}^0$$

This is clearly well defined as $0^0 \not\equiv_{V^0} \{x\}^0$ and $\left\{0^0, \{x\}^0\right\}^0 \not\equiv_{V^0} \left\{\{y\}^0\right\}^0$, which hold simply due to the fact that $0 \not\equiv_{\mathcal{U}} 1$ and $2 \not\equiv_{\mathcal{U}} 1$. Note that this allows us to prove that equality of pairs is just pairwise equality of components, i.e. we have that:

$$\left(\langle x, y\rangle^0 \equiv_{V^0} \langle a, b\rangle^0\right) \simeq (x \equiv_{V^0} a) \times (y \equiv_{V^0} b)$$

In other words, in contrast to unordered pairs, which are unordered in the sense that equality contains a choice of permutation for the component-wise equalities, ordered pairs are actually ordered in the sense that their equality is the fixed choice of equalities between the first and second components respectively.

Dependent Pair Types

Ordered pairs now allow us to now encode Σ-types. The definition is rather straightforward as we encode a Σ-type as a Σ-type indexed families of ordered pairs. The definition is rather straightforward: Suppose that x is an iterative set (definitionally encoding $\mathrm{El}^0(x)$)

[10] This is the approach used by H. R. Gylterud and E. Stenholm [10] that is due to fact that the simpler Kuratowski pairing will not work for higher levels of their hierarchy V^n. In our case, defining $\langle x, y\rangle^0 \coloneqq \left\{\{x\}^0, \{x, y\}^0\right\}^0$ would also suffice

and $y : \mathrm{El}^0(x) \to V^0$ a family of iterative sets ($y(a)$ definitionally encoding $\mathrm{El}^0(y(a))$). Then we define the iterative set

$$\Sigma^0(x, y) \coloneqq \mathrm{fromEmb}\left(\sum_{a:\,\mathrm{El}^0(x)} \mathrm{El}^0(y(a)), \varphi\right)$$

where $\varphi : \sum_{a:\,\mathrm{El}^0(x)} \mathrm{El}^0(y(a)) \to V^0$ is defined as $\varphi(a, b) \coloneqq \langle \tilde{x}(a), \widetilde{y(a)}(b)\rangle^0$. The function φ is an embedding since $\tilde{x}$ and $\widetilde{y(a)}$ are both embeddings and due to the fact that two ordered pairs are equal if and only if the first and second components are equal respectively.

As required, the iterative set $\Sigma^0(x, y)$ encodes the desired Σ-type definitionally:

$$\mathrm{El}^0\big(\Sigma^0(x, y)\big) = \sum_{a:\,\mathrm{El}^0(x)} \mathrm{El}^0(y(a))$$

In particular this means that V^0 is closed under Σ-types. This includes ordinary (non-dependent) products which are special cases in which the family y is constant.

Dependent Function Types

Like Σ-types, V^0 is also closed under formation of Π-types. For this, let x be an iterative set and $y : \mathrm{El}^0(x) \to V^0$ be a family of iterative sets. We will give an encoding for the type $\prod_{a:\,\mathrm{El}^0(x)} \mathrm{El}^0(y(a))$. As before, our encoding will draw inspiration from ordinary set theory, where a function is (definitionally!) equal to its graph. Hence we define for every dependent function $f : \prod_{a:\,\mathrm{El}^0(x)} \mathrm{El}^0(y(a))$ the iterative set Γ^0_f by:

$$\Gamma^0_f \coloneqq \mathrm{fromEmb}\big(\mathrm{El}^0(x), l\big)$$

$$l : \mathrm{El}^0(x) \to V^0$$

$$l(a) \coloneqq \langle \tilde{x}(a), \widetilde{y(a)}(f(a))\rangle^0$$

where l is injective since already $\tilde{x}$ is an injective. This now lets us define the code for the Π-type as:

$$\Pi^0(x, y) \coloneqq \mathrm{fromEmb}\left(\prod_{a:\,\mathrm{El}^0(x)} \mathrm{El}^0(y(a)), \lambda f.\Gamma^0_f\right)$$

The injectivity of the function that sends a dependent function f to its graph Γ^0_f is due to the following observation: If $\Gamma^0_f \equiv_{V^0} \Gamma^0_f$, then by definition and by the Embedding Identity Principle for V^0 we have for every $a : \mathrm{El}^0(x)$ that

$$\langle \tilde{x}(a), \widetilde{y(a)}(f(a))\rangle^0 \in^0 \Gamma^0_f \simeq \langle \tilde{x}(a), \widetilde{y(a)}(f(a))\rangle^0 \in^0 \Gamma^0_g$$

and since the left hand side holds by definition, we have a proof (a', p) of the right hand side. The injectivity of $\tilde{x}$ and $\widetilde{y(a)}$ now gives us $q : a \equiv_{\mathrm{El}^0(x)} a'$ and the injectivity of $\widetilde{y(a)}$ gives us $f(a) \equiv_{V^0} g(a)$ via a transport over q, which shows (by function extensionality) that $f \equiv g$, i.e. injectivity.

By construction, this encoding is again definitional, i.e.

$$\mathrm{El}^0\big(\Pi^0(x, y)\big) = \prod_{a:\, \mathrm{El}^0(x)} \mathrm{El}^0(y(a))$$

as required, which shows that V^0 is closed under Π-types.

Coproducts

Our encoding of sum types will be similar to the encoding of disjoint unions in ordinary set theory, where two not-necessarily disjoint sets A and B are artificially made disjoint e.g. as $A \uplus B \coloneqq (A \times \{0\}) \cup (B \times \{1\})$. For this we will use the following lemma:

Lemma 2.7 ([7, Lemma 27]). *Let A, B and C be types and consider the embeddings $f : A \hookrightarrow C$ as well as $g : B \hookrightarrow C$. If for all $x : A$ and $y : B$ we have that $f(x) \not\equiv_C g(y)$, then the function $h : A + B \to C$, defined as*

$$h(\mathrm{inl}(x)) \coloneqq f(x)$$
$$h(\mathrm{inr}(y)) \coloneqq g(y)$$

is also an embedding.

Now let x and y be iterative sets (definitionally encoding the types $\mathrm{El}^0(x)$ and $\mathrm{El}^0(y)$ respectively). Then we define two functions $f : \mathrm{El}^0(x) \to V^0$ and $g : \mathrm{El}^0(y) \to V^0$ by letting

$$f(a) \coloneqq \langle 0^0, \tilde{x}(a)\rangle^0$$
$$g(b) \coloneqq \langle 1^0, \tilde{y}(b)\rangle^0$$

Both f and g are injective (hence embeddings) since $\tilde{x}$ and $\tilde{y}$ are, and clearly all their values are pairwise unequal since $0^0 \not\equiv_{V^0} 1^0$. Therefore the function $h : \mathrm{El}^0(x) + \mathrm{El}^0(y) \to V^0$ as defined in Lemma 2.7 is an embedding. We can now assemble this into the iterative set

$$x +^0 y \coloneqq \mathrm{fromEmb}\big(\mathrm{El}^0(x) + \mathrm{El}^0(y), h\big)$$

which decodes definitionally as $\mathrm{El}^0\big(x +^0 y\big) = \mathrm{El}^0(x) + \mathrm{El}^0(y)$.

Identity Types

To encode identity types we will use the fact that all types that are encoded by some iterative set are h-sets, i.e. their equality types are propositional. For this, let x be an iterative set consider the two elements $u, v : \mathrm{El}^0(x)$. Then we define the following iterative set:

$$u \equiv^0 v \coloneqq \mathrm{fromEmb}\big(u \equiv_{V^0} v, \lambda\_.0^0\big)$$

Note that the function is an embedding since its domain $u \equiv_{V^0} v$ is an h-proposition. For the decoding we have

$$\mathrm{El}^0\big(u \equiv^0 v\big) = (u' \equiv_{V^0} v')$$

definitionally, which shows that V^0 is also closed under identity types.

Other Notable Types

The universe V^0 is also closed under other type constructions, given that the surrounding type theory also supports them [7]. This includes, e.g., quotient types as well as universe codes, whereby for the latter we mean that if the type theory has (at least) two universes $\mathcal{U}$ and $\mathcal{U}'$ with $\mathcal{U} \hookrightarrow \mathcal{U}'$ (e.g. as part of a hierarchy of universes), then $V^0_{\mathcal{U}'}$ contains a code for $V^0_{\mathcal{U}}$. As with the encodings above, the encodings can be chosen such that the decodings are definitional.

Moreover, the closure properties are not limited to type constructions that that are introduced via certain typing rules. They also hold for all definable type constructions, e.g. fibers.

2.3. Iterative Sets Form A CwF

In this chapter we will show that the universe V^0 supports a Category-with-Families structure to model dependent type theory. The way that this structure is defined is very similar to the set universe hSet. The approach is rather straightforward and many equalities will hold definitionally. Even the ones that do not hold definitionally are (in theory) repeated applications of function extensionality and transports over the reflexivity path. This makes the proofs easy to write down on paper, where it is possible to gloss over the intricacies working in the background.

We start by showing how the universe V^0 forms a category in the obvious way:

Theorem 2.8 ([7, Definition 35]). *The following structure describes a category $\mathcal{V}$:*

- *The collection of objects is the type V^0.*
- *If x and y are iterative sets, then the collection $\mathcal{C}(x, y)$ is given by the h-set of functions $\mathrm{El}^0(x) \to \mathrm{El}^0(y)$.*
- *For every iterative set x, the identity arrow is the identity function $\mathrm{El}^0(x) \to \mathrm{El}^0(x)$*
- *If $x \xrightarrow{f} y$ and $y \xrightarrow{g} z$ are arrows, then the composition $x \xrightarrow{f*g} z$ is defined as the composition $g \circ f$.*

Associativity and the neutrality properties of the identities hold definitionally by function extensionality. The decoding function El^0 can be made into a fully faithful functor where all the required equalities hold definitionally:

Lemma 2.9 ([7, Lemma 36]). *There is a fully faithful functor* $\mathrm{El}^0 : \mathcal{V} \to \mathrm{hSet}$ *that sends* $x \xrightarrow{f} y$ *to* $\mathrm{El}^0(x) \xrightarrow{f} \mathrm{El}^0(y)$.

It is easy to see that this holds since functions between the index types correspond precisely to arrows in $\mathcal{V}$. Note that El^0 is not an embedding of $\mathcal{V}$ into hSet since the object map is not an embedding, in fact it is not even injective. A counterexample is the before mentioned fact that $\mathrm{El}^0\big(1^0\big) = 1 = \mathrm{El}^0\big(\{x\}^0\big)$.

Note that our (natural) definition of arrow between iterative sets as functions between element sets now implies that our universe V^0 is not univalent. This is due to the fact that for all $x, y : V^0$, the type $x \equiv_{V^0} y$ is an h-proposition while the type $x \simeq y = \mathrm{El}^0(x) \simeq \mathrm{El}^0(y)$ is in general an h-set, meaning that the two types cannot in general be equivalent.

Similar to hSet, the category $\mathcal{V}$ is closed under various categorical constructions that make it a worthwile object of study. The precise definitions of the respective structures follow the definitions inside hSet.

Theorem 2.10 ([7, Theorem 37]). *The category* $\mathcal{V}$ *contains initial and terminal objects and is closed under:*

1. *binary products,*
2. *binary coproducts,*
3. *pullbacks,*
4. *pushouts,*
5. *and exponentials.*

Moreover, the precise definitions can be chosen in a way such that the decodings are definitionally equal to the corresponding structure in hSet.

The above shows in particular that $\mathcal{V}$ is a Cartesian Closed Category, which enables $\mathcal{V}$ to model a simple non-dependent type theory. We will show now that $\mathcal{V}$ can even model dependent type theory by giving it a CwF structure.

Theorem 2.11 ([7, Proposition 45]). *There are functors:*

1. $\mathrm{Ty} : \mathcal{V}^{\mathrm{op}} \to \mathrm{hSet}$ *with* $\mathrm{Ty}(\Gamma) \coloneqq \mathrm{El}^0(\Gamma) \to V^0$,
2. $\mathrm{Tm} : (\textstyle\int \mathrm{Ty})^{\mathrm{op}} \to \mathrm{hSet}$ *with* $\mathrm{Tm}(\Gamma, A) \coloneqq \prod_{x:\, \mathrm{El}^0(\Gamma)} \mathrm{El}^0(A(x))$,
3. *and* $- \,.\, - : \textstyle\int \mathrm{Ty} \to \mathcal{V}$ *with* $\Gamma \,.\, A \coloneqq \Sigma^0(\Gamma, A)$

such that they equip $\mathcal{V}$ *with a CwF structure.*

Proof. First note that the initial object of $\mathcal{V}$ is given by the empty set 0^0. Next, the presheaf Ty maps arrows $x \xleftarrow{f} y$, i.e. functions $\mathrm{El}^0(y) \to \mathrm{El}^0(x)$ to the precomposition

$$(g \mapsto g \circ f) : \big(\mathrm{El}^0(x) \to V^0\big) \to \big(\mathrm{El}^0(y) \to V^0\big)$$

such that all the required equalities hold definitionally. The actions of the presheaf Tm and the functor $- \,.\, -$ on arrows are similarly conceptually clear, but they require substitution and will be examined later in Section 3.2. ∎

This means that $\mathcal{V}$ is expressive enough interpret dependent type theory. Moreover, $\mathcal{V}$ supports a Π- and a Σ-structure. These proofs are more or less immediate on paper, but stating them precisely in a proof assistant requires significantly more effort. For a proof that $\mathcal{V}$ supports Π- and Σ-structures, see D. Gratzer, H. R. Gylterud, A. Mörtberg, and E. Stenholm [7]. We will repeat the argument for Σ-structure in the following:

Theorem 2.12 ([7, Lemma 51]). *There is an operation* sig *that equips $\mathcal{V}$ with a Σ-structure.*

Proof. The operation sig can be defined as follows:

$$\mathrm{sig}_\Gamma(A, B) \coloneqq \lambda\big(x : \mathrm{El}^0(\Gamma)\big).\Sigma^0(A(x), \lambda a.B(x, a))$$

The naturality of sig follows by reflexivity. The equivalence also follows directly and the its naturality condition holds definitionally as well. ■

3. Formalization in Cubical Agda

In this chapter we will explore the computer formalizations of the results of the previous chapter. First, we will analyze the prior formalization of D. Gratzer, H. R. Gylterud, A. Mörtberg, and E. Stenholm [7] in standard Agda using the agda-unimath library [18], examining the difficulty that led to the inability to define Σ-structure as well as an implementation for iterative sets. We then describe our own formalization, with a focus on the latter parts that were problematic for the prior formalization. In addition to a first set of definitions that follow the prior formalization closely, we suggest two sets of changes that improve the situation and provide us with hope for a solution even though we were ultimately unable to complete the naturality condition for the implementation of Σ-types for iterative sets. Afterwards, we describe the difficulties that we faced and suggest possible ways forward.

3.1. The Prior Formalization

In their formalization, D. Gratzer, H. R. Gylterud, A. Mörtberg, and E. Stenholm [7] first implement the type V^∞ of iterative multisets as well as the hierarchy of iterative truncated universes V^n. This is based on the work of H. R. Gylterud and E. Stenholm [10] and contains the closure properties under the standard constructions that turn out to even hold in this generalised setting. Here, the universe V^0 of iterative sets is merely a special case. We will explain the differences to our present formalization in Section 3.2.

The second part is the formalization of Categories with Families and the associated Π-structure. In this part, the authors show that the universe V^0 in particular can be equipped with a CwF as well as a Π-structure. However, they stop short of defining Σ-structure and implementing it for V^0 due to "complex path algebra" [7].

The definition of Categories[11] with Families up until the definition of the naturality condition is a rather straightforward translation of the corresponding paper definition, note that it uses record to save all the required data. The naturality condition is expressed as:

```
ext-iso-nat :
  (Δ Δ' Γ : Ctx) (A : Ty Γ) (σ : Sub Δ' Δ) →
  (τ : Sub Δ (ext Γ A)) →
  map-equiv (ext-iso Δ' Γ A) (comp-sub γ δ) ≡
  (comp-sub (pr1 (map-equiv (ext-iso Δ Γ A) γ)) δ ,
  inv-tr (Tm Δ') (preserves-comp-Ty A _ δ)
    (pr2 (map-equiv (ext-iso Δ Γ A) γ) [ δ ]))
```

[11] In the agda-unimath library the term "precategory" is used as the word "category" is reserved for the case in which the identifications between the objects are precisely the isomorphisms [18].

where in the second component of the right-hand side we need a transport via the proof $p : \mathrm{Tm}(\Delta' . ((A \cdot \tau) \cdot \sigma)) \equiv \mathrm{Tm}(\Delta' . (A \cdot (\tau \circ \sigma)))$ since the application of $-[\sigma]$ gives an element of the former type while naturality requires the latter. This makes it so that the commutative diagram for the naturality condition can be broken down to the following

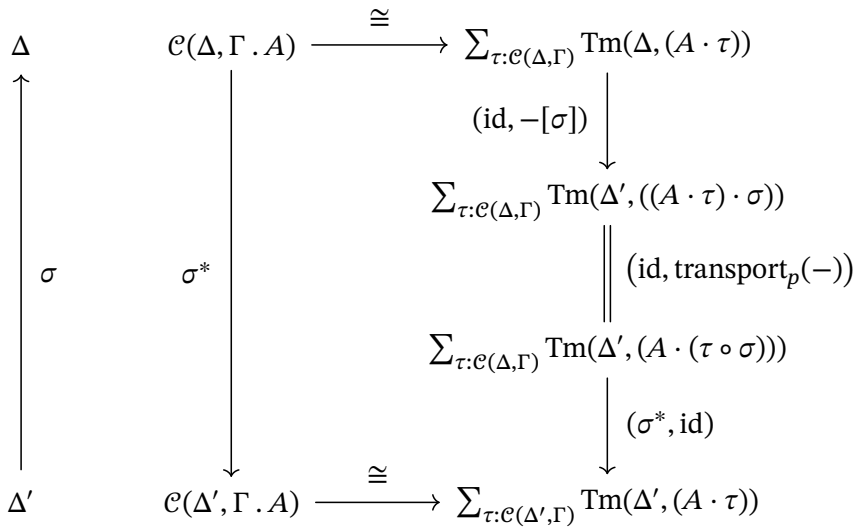

Figure 1: The naturality condition for CwF

In many cases, the right downward edge could simply be the function $(\sigma^*, -[\sigma]))$, but in general the proof p might not be (definitionally equal to) refl, so in order to cover the full generality, we require a transport. Working in standard Agda and book HoTT, this transport will simply become definitionally equal to the identity function in many cases (including the one of iterative sets), and therefore we can happily ignore its use here. However, working in Cubical Agda and cubical type theory, things get more complicated as transports over refl are not definitionally equal to the identity function, so introducing a transport here makes things more complicated, see Section 3.2.

The subsequent instantiation of CwF for iterative sets is rather straightforward, but the ease of pattern-matching refl hides some complexity. For the type presheaf and the context extension functor the definitions are direct and the necessary equalities hold definitionally. The case for the term presheaf requires two caveats

```
Tm-V⁰ : _
pr1 Tm-V⁰ (Γ , A) =
    raise-Set (lsuc i) (Π-Set' (El⁰ Γ) λ x → El⁰-Set i (A x))
pr1 (pr2 Tm-V⁰) (σ , refl) (map-raise a) = map-raise (a ∘ σ)
pr1 (pr2 (pr2 Tm-V⁰)) (γ , refl) (σ , refl) =
    eq-htpy λ { (map-raise a) → refl }
pr2 (pr2 (pr2 Tm-V⁰)) (Γ , A) =
    eq-htpy (λ { (map-raise a) → refl })
```

First, the terms require universe lifting such that the term is in the instance of hSet with the correct universe level, and second, to show that the term presheaf respects composition (the third case above), they use the J rule together with function extensionality (in the form of `eq-htpy`). The first point makes goals later slightly harder to read, but the second point can make things a lot more complicated later. This is particularly the case when using the presheaf action this means that paths do not reduce as expected. The reason is that function extensionality needs to be postulated and hence does not compute in standard Agda and book HoTT.

It remains to prove the natural equivalence. Since the equations for the context extension functor are definitional, showing the equivalence is rather easy and the naturality condition thereof holds definitionally. Similarly, the definition of Π-structure is straightforward (again using a transport) and its instantiation for iterative sets as well.

Σ-structures, however, seem to be particularly complicated. The reason is likely that in contrast to Π-structure, we need to deal with dependent pairs where the second component contains a transport over a complicated path over a proof of function extensionality. For this reason, D. Gratzer, H. R. Gylterud, A. Mörtberg, and E. Stenholm [7] abandoned their formalization of Σ-structures and the corresponding instantiation for iterative sets. In the next section, we demonstrate our approach of the formalization in Cubical Agda.

3.2. A New Formalization in Cubical Agda

In this section we describe our formalization of iterative sets as a Category with Families. We will first give a short outline of our formalization up to and including the implementation of $\mathcal{V}$. We then present our definitions and instantiations of CwF and Σ-structures for iterative sets. For this we start by making a definition that is very close to the prior formalization. We then suggest two sets of improvements, replacing equality types with transport by heterogenous path types and the introduction of ad-hoc functions as dedicated record fields. The second set of improvements makes the goals inside Agda easier to read. However, we still fail to completely formalize the naturality condition, even though we could simplify the final goal significantly and are more optimistic about the general feasibility of the formalization.

For the first part, we follow the general direction of D. Gratzer, H. R. Gylterud, A. Mörtberg, and E. Stenholm [7] and make changes as outlined in Chapter 2. A direct translation is impossible due to the fact that the J-rule is not definitional anymore, which forces us to take explicit care of the corresponding transports. This includes specifying the family that is used in the definition of the J-rule, which Agda often fails to infer. Another reason is that the agda-unimath [18] and agda-cubical [25] libraries are structured differently and many proofs of the agda-unimath library do not have a counterpart

in the agda-cubical library yet. This forces us to use a slightly different approach for some parts, though the overall proof ideas remain the same. E.g. we do not need many of the statements in the full generality since V^0 is an h-set and we can thus e.g. injectivity already implies the embedding property.

In the following, we explain our formalization of CwF and Σ-structure, i.e. parts that turned out to be problematic for the prior formalization.

The Naive Approach

In our first naive version, we follow D. Gratzer, H. R. Gylterud, A. Mörtberg, and E. Stenholm [7] for the definition of Categories with Families, including the substitution in the naturality condition. Even though the implementation of the type presheaf is as easy as in the prior formalization, the rest is generally more difficult due to the differences between standard HoTT and cubical type theory, in particular the lack of a definitional J-rule.

Here the use of Cubical Agda exposes the hidden challenges that working with the category of elements poses: Suppose that $P : \mathcal{C} \to \mathrm{hSet}$ is a presheaf. Then the composition of arrows in $\int P$ behaves somewhat poorly as already associativity as well as the identity rules do not hold definitionally. Similar problems also arise for functors out of $\int P$, for us in particular for the term presheaf and the context extension functor. In both cases, the definition of the image of an arrow already requires substitution to get the elements at the correct endpoints:

```
V-CwF .tmPresheaf .F-hom f t =
  lift (λ x → subst El⁰ (funExt⁻ (f .snd) x) (t .lower (f .fst x)))
...
V-CwF .ctxExtFunctor .F-hom {x} {y} f t .snd =
  subst⁻ El⁰ (funExt⁻ (f .snd) (t .fst)) (t .snd)
```

Similarly to before, however, we also need to raise the universe level as in our definition of CwF we want to, in principle, allow the target hSet to be the hSet of any arbitrary universe level. Due to the way substitutions work in a cubical setting, the proofs of the necessary functor equalities become more difficult.

For the natural equivalence, the definition of the actual equivalence is as easy as before, while the proof of the naturality condition is now not reflexivity anymore, but the composition of instances of transportRefl, i.e. the proof that $\mathrm{transport}_{\mathrm{refl}_A}(x) \underset{A}{=} x$. Note that the naturality condition is an equality of a Σ-type, and fortunately the first component of that equality is indeed refl. Even though the proof of the second component is not too difficult, these complexities accumulate for every additional layer of structure added.

Moving to the definition of Σ-structure, the difficulty here lies in the fact that for the naturality square in fact both the left-and-bottom as well as the top-and-right arrow

compositions require substitutions as the natural vertical arrows $-[\sigma]$ and $(-[\sigma], -[\sigma])$ do not produce terms of the correct type, as depicted in the following diagram:

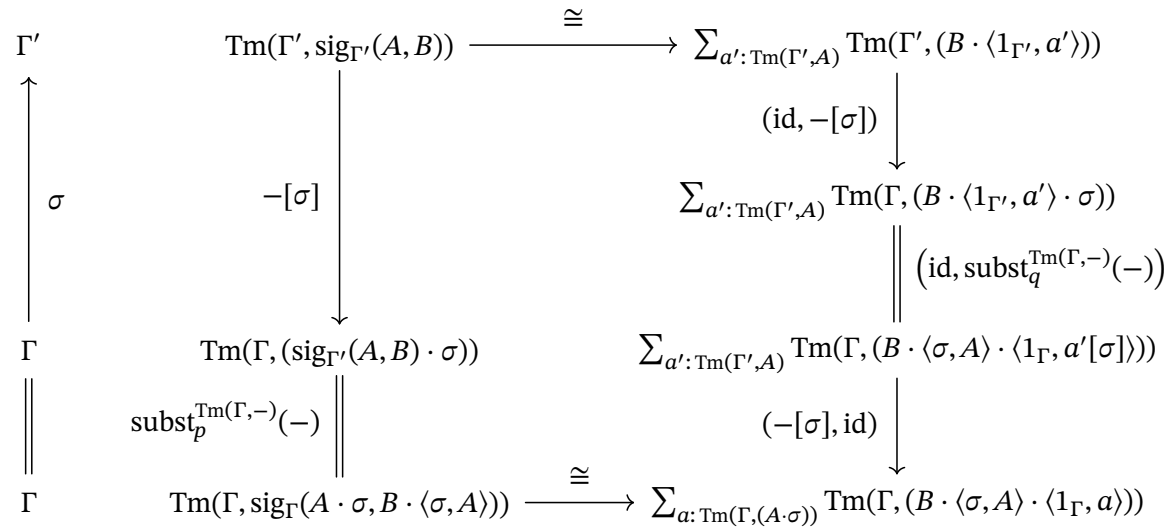

Figure 2: The naturality of the equivalence for Σ-structures

where

$$p : \mathrm{sig}_{\Gamma'}(A, B) \cdot \sigma \equiv_{\mathrm{Ty}(\Gamma)} \mathrm{sig}_{\Gamma}(A \cdot \sigma, B \cdot \langle \sigma, A \rangle)$$

and

$$q : (B \cdot \langle 1_{\Gamma'}, a' \rangle) \cdot \sigma \equiv_{\mathrm{Ty}(\Gamma)} (B \cdot \langle \sigma, A \rangle) \cdot \langle 1_{\Gamma}, a'[\sigma] \rangle$$

Here, p is precisely the the naturality property of sig and we introduce q as a new field. While q could likely be proven from the axioms of a CwF, this proof will generally not be optimal, as some particular instances might have ways to prove this equality more easily (it might even hold definitionally). Relying on ad-hoc instantiations instead of general proofs for substitution will make the proof goals reduce better and significantly simplify proofs in many cases. For the other record fields of the Σ-structure, the definition is straightforward, precisely following Theorem 2.12.

The instantiation of Σ-structure for iterative sets is more complicated: While the implementation of p is straightforward, the implementation of q requires complex path composition over multiple instances of substRefl. Since a transport over q is part of the type of the naturality condition, this very much complicates its implementation. What is more, the definition of the actual equivalence itself requires transport over substRefl as well.

Following a similar pattern, where the first component of an equality is significantly easier to implement[12] than the second, the first component of the naturality condition is a rather easy application of substRefl. For the second component, however, all the

[12] in some cases it is even refl

the amassed complexity combines: the second component is now a heterogenous path over substRefl in a type that itself contains multiple transports over compositions of substRefl. While this seems like a problem that should be, in principle, solvable, in reality the proof assistant goals become highly unreadable and efforts to simplify the main goal by creating intermediate subgoals will often not fully typecheck because Agda can not infer some hidden types. Due to the ballooning complexity, we similarly abandon this approach, but in the following sections we will discuss different strategies in order to simplify the implementation of the naturality condition.

We highly suspect that this ballooning stack of complexity was also the reason why D. Gratzer, H. R. Gylterud, A. Mörtberg, and E. Stenholm [7] did not formalize Σ-structure in the end. While we believe that defining Σ-structure would have been possible using the naturality condition above as well requiring an additional field for the equality q, the instantiation for iterative sets would also suffer from similar problems. In the case of standard Agda, the source of the problems is, however, not the fact that $\mathsf{transport}_{\mathrm{refl}(A)}(x)$ does not reduce to x, but due to the non-computation of function extensionality. In our definitions, function extensionality is used quite significantly, in particular also in the case where a similar definition by the J-rule would then require a transport via that proof of function extensionality, which will not reduce. We found it interesting to see that mitigating and improving upon one aspect of standard Agda would significantly complicate another aspect inside Cubical Agda, effectively leaving the difficulty level of the overall problem unchanged.

Replacing Equalities With Heterogenous Paths

Since transport exhibits this undesirable behaviour in our cubical setting, it is only natural to try to avoid it as much as possible. In this section we try to minimize the use of transports by employing heterogenous path types and adding additional fields to our structures that give us the functionality of transport with improved reductions in the instantiations. Unfortunately, we find that avoiding transport completely is impossible due to the category of elements construction that is necessary for term presheaf as well as the context extension functor. As noted before, arrow composition in the category of elements of a presheaf behaves poorly in the sense that associativity as well as the identity laws do not hold definitionally. This means, that at least for the implementation of the term presheaf and context extension functor, transport cannot be avoided.

The more feasible approach is to try to at least remove transport from the type signatures of the fields for CwF and Σ-structures. One way to remove transport from equality types, is by using heterogenous path types since they have better definitional behaviour. In fact, this is actually the preferred way in a cubical setting. To demonstrate how this works, let A and B be types, $r : A \equiv_{\mathcal{U}} B$, $x : A$ and $y : B$. Whereas in book HoTT

one would consider the homogeneous equality type $\mathrm{transport}_p(x) \equiv_B y$, in cubical type theory it is better to instead consider the heterogenous path type $\mathrm{PathP}_p(x, y)$. It is clear that one can convert from either of the statements to the other, but the latter has better reduction behaviour: e.g. we have that $\mathrm{refl}_x : \mathrm{PathP}_{\mathrm{refl}_A}(x, x)$, but refl_x is not a proof of $\mathrm{transport}_p(x) \equiv_A x$, which can be seen as the main culprit of our previously experienced problems.

We apply this technique now for the naturality condition of CwF, i.e. with the notation from Figure 1 and naming the equivalence components φ_Δ, the statement

$$\varphi_{\Delta'}(\tau \circ \sigma) \equiv (\pi_1(\varphi_\Delta(\tau)) \circ \sigma, \mathrm{transport}_q(\pi_2(\varphi_\Delta(\tau))[\sigma])$$

for all $\tau : \Delta \to \Gamma \,.\, A$. Ideally, we would want to directly use a heterogenous path type over the path q[13] to connect $\varphi_{\Delta'}(\tau \circ \sigma)$ and $(\pi_2(\varphi_\Delta(\tau)) \circ \sigma, \pi_2(\varphi_\Delta(\tau))[\sigma])$. Unfortunately, the second term cannot be well-typed since its second component depends on some substitution $\gamma : \mathcal{C}(\Delta', \Gamma)$ while its first component is a substitution $\mathcal{C}(\Delta, \Gamma)$.

In order to solve this, we can split the naturality condition, which is an equality of a Σ-type, into two components. The first component is a proof

$$r_1 : \pi_1(\varphi_{\Delta'}(\tau \circ \sigma)) \equiv_{\mathcal{C}(\Gamma,\Delta)} \pi_1(\varphi_\Delta(\tau)) \circ \sigma$$

The second component is a heterogenous path

$$r_2 : \mathrm{PathP}_l(\pi_2(\varphi_{\Delta'}(\tau \circ \sigma)), \pi_2(\varphi_\Delta(\tau))[\sigma])$$

where $l : \mathrm{Tm}(\Gamma, (A \cdot \pi_1(\varphi_{\Delta'}(\tau \circ \sigma)))) \equiv_\mathcal{U} \mathrm{Tm}(\Gamma, ((A \cdot \varphi_\Delta(\tau)) \cdot \sigma))$ given by

$$\mathrm{cong}_{\mathrm{Tm}(\Gamma,-)}(\mathrm{cong}_{A.-}(r_1) \cdot q)$$

In the instantiation for iterative sets, both paths r_1 and q are refl, so $l = \mathrm{cong}_{\mathrm{Tm}(\Gamma,-)}(\mathrm{refl} \cdot \mathrm{refl})$, but unfortunately $\mathrm{refl} \cdot \mathrm{refl}$ is not definitionally equal to refl, so we can use substRefl to prove

$$\pi_2(\varphi_{\Delta'}(\tau \circ \sigma)) \equiv_{\mathrm{Tm}(\Delta',(A\cdot\tau))} \pi_2(\varphi_\Delta(\tau))[\sigma])$$

and then transport the result via the proof of $\mathrm{refl} \cdot \mathrm{refl} \equiv \mathrm{refl}$.

One could now try to implement a similar strategy for the naturality condition for Σ-structure. However, due to the fact that the corresponding equality contains two transports, and already the "more cubical" approach for CwF where there is only a single transport, has only led to marginal improvements, it is unlikely to improve the overall situation. Moreover, using heterogenous paths instead of transport can only be applied to equality types, which makes it unsuitable for the other fields whose types all contain instances of transport as well.

[13] recall, this is simply the fact that the presheaf action respects arrow composition

Replacing Transport With Ad-Hoc Functions

Fortunately, there is another way of eliminating the need for transport in the definitions of Categories with Families as well as Σ-structures. Consider types A and B as well as $p : A \equiv_{\mathcal{U}} B$ and $x : A$. Then by definition $\mathrm{transport}_p(x) : B$, i.e. transporting along p has the type

$$\mathrm{transport}_p(-) : A \to B$$

If $\mathrm{transport}_p(-)$ occurs in the types of some of our fields, we can simply add another field $f_p : A \to B$ and replace all occurrences of $\mathrm{transport}_p(-)$ by f_p. For a specific instantiation, if the path p is particularly simple, we can then implement the function f_p ad-hoc with a method that reduces better than $\mathrm{transport}_p(-)$. This is especially relevant, if p is refl_A, since then, we can implement f_p simply as the identity function, i.e. $f_p(x) \coloneqq x$ and eliminating the need for a transport entirely. This approach is beneficial, because it can significantly improve the reduction behaviour and simplify proof assistant goals. Additionally, there is no real downside, since even if no simple ad-hoc implementation of f_p exists, one could always fall back to using $\mathrm{transport}_p(-)$.

For the types in the definition of a CwF, transport over the proof p that the type presheaf action respects composition of arrows is the only occurrence of transport. For this, we introduce a new record field $f_p : (A \cdot \gamma) \cdot \sigma \to A \cdot (\gamma \circ \sigma)$ which simplifies the naturality condition to

$$\varphi_{\Delta'}(\tau \circ \sigma) \equiv (\pi_1(\varphi_\Delta(\tau)) \circ \sigma, f_p(\pi_2(\varphi(\tau))[\sigma])$$

With this definition, the second component in the instantiation of the naturality condition is simply a single application of substRefl, which shows that this approach has real practical benefits.

A similar simplification can be made for Σ-structures: Here, there are several instances of transport via the proof $s : A \cdot \mathrm{id} \equiv_{\mathrm{Ty}(\Gamma)} A$. By adding a field $f_s : A \cdot \mathrm{id} \to A$, we can use it to replace every occurrence of $\mathrm{transport}_s(-)$, significantly improving the proofs. In this case, the equivalence becomes a strong equivalence, i.e. the retraction and section properties hold definitionally, and also the auxiliary equality becomes significantly easier, albeit still not refl.

The goals for the naturality condition of are also much simpler, though we were still unable to fully complete the proof. Currently, the only open goal is

```
subst (Tm V-CwF Γ)
(λ i x → B (σ x , subst El⁰ refl (p' x i)))
(lift (λ x → subst (Tm V-CwF Γ)
   (λ i x →
        Σ⁰ (A (σ x)) ((λ a₁ → B (σ x , substRefl a₁ (~ i)))))
  (lift (λ x → subst El⁰
    (refl {x = Σ⁰ (A (σ x)) (λ a₁ → B (σ x , a₁))})
```

```
      (a .lower (σ x)))) .lower x .snd))
≡
subst (Tm V-CwF Γ)
(V-Σ-Structure {ℓ} .ctxExtSubstSigmaSndEq A B
  (lift (λ x → a .lower x .fst)) σ)
  (lift (λ x → subst El⁰ refl (a .lower (σ x) .snd)))
```

where `ctxExtSubstSigmaSndEq` is our auxiliary equality. Continuing from here, one potentially promising approach could be to try to show that the two paths that are being substituted over can be continuously deformed into each other (though they cannot be equal because the endpoints do not match definitionally). This would amount to exhibiting a homotopy square that might not be that difficult to fill due to the fact that it is located in an h-set. If this holds, one could then try to show that the points that are being transported are equal, and combined with the previous, one might be able to show that the substitution results, i.e. equal points transported over equal paths, are also equal, proving the missing goal.

Even though a solution seems to be in close reach, it is unclear how to carry out the above steps in detail. For one, the goal constraints displayed by Agda are suboptimal in the sense that the normalization level is either not enough (record fields are not expanded at all), too high (complete normalization down to the level of Agda internals like `hcomp` and `transp`), or in fact both (definitions in `where` do not get expanded even in the highest normalization mode). This also makes Agda's automatic goal filling, that relies on these normalization levels practically useless. Another difficulty is that at every intermediate step, the proof assistant cannot infer various hidden constraints (e.g. successive universe level raising and elimination with undetermined level) and will highlight various terms without clearly communicating what precisely is missing. Moreover, Agda sometimes allows us to refine goals with terms that we can clearly deem to be wrong, if it can find some technically possible combination of constraints that allows the refining term to be of the correct type. Unfortunately, this means that it becomes difficult to find the exact reason why a particular term is highlighted.

In conclusion, our formalization in Cubical Agda so far is promising and could highlight potential advantages of formalization in a cubical setting as opposed to book HoTT. It should be stated though, that the path towards the current formalization was, in many places, comparatively more difficult than in the original formalization due to the poorer reduction properties of transport and the J-rule.

3.3. Potential Improvements to the New Formalization

In addition to unclear goal types, our formalization also encountered various performance issues. Notably, every module directly using `Cubical.Data.IterativeSets.UnorderedPair` will take a long time (> 5 minutes) to typecheck or will not typecheck at all within a reasonable amount of time (ca. 1 hour) using Agda's default unification algorithm. While we did manage to significantly improve the typechecking time by enabling the "lossy unification" mode for almost all modules (initial typechecking taking at most 30 seconds), one notable exceptions is the module `Cubical.Data.IterativeSets.Pi`, the module where we define the encoding of Π-types. There, even enabling the lossy unification mode will not make the module compile within a reasonable amount of time. While we concede that this means that this module has not been completely verified by Agda, we are still confident that the implementation is correct, since we managed to type check all parts individually, including some overlap.

In order to improve the performance, we can think of two possible solutions: Either we change some individual implementations to make them reduce better and thus more performant, or we can do the opposite by adding the `opaque` keyword which instructs Agda not to unfold some particular definitions, meaning that all the potentially complicated implementations of these definitions are ignored for outside use. Unfortunately, it requires some effort to find the exact performance bottlenecks, though we have identified one potential candidate for improvement, the Fibration and Embedding Identity Principles (see Theorem 2.4 and Corollary 2.4.1 respectively). Their implementation involves "some universe magic to achieve good universe polymorphism" [25], which is a level of generality that we do not in fact need. One could imagine that using only the specific parts that are necessary from that definition would improve the type checking time, but at this point it is unclear if it is the actual main culprit responsible for the poor performance.

The above performance improvements could also positively impact the legibility of goal types later on, though we still believe that the major problem pertains to the utilization of the category of elements as described above. Since one possible reason that hinders Agda to infer some hidden/implicit type constraints are universe levels, one could also try to use Agda's "type in type" mode. With this mode enabled, Agda will ignore universe level checks completely, obviating the need for universe level raising. One could try to enable this mode temporarily until a solution for the last remaining goal has been found, and then reintroduce universe level mechanics to the solution as necessary.

3.4. Navigating the Code Repository

The code for the formalization can be found in my fork of the agda-cubical library [8]. It is intended to be merged into the upstream library shortly after submission. The formalization is organized inside the agda-cubical library as follows:

- `Cubical.Data.W`: contribution of the disproof of $x \in^W x$
- `Cubical.Data.IterativeMultisets`: formalization of V^∞
- `Cubical.Data.IterativeSets`: formalization of V^0
- `Cubical.Categories.Instances.IterativeSets`: implementation of $\mathcal{V}$
- `Cubical.Categories.WithFamiliesNaive`: naive approach defining CwF and Σ-structure and instantiating them for iterative sets. This is mostly following along D. Gratzer, H. R. Gylterud, A. Mörtberg, and E. Stenholm [7] with some major adaptations in the implementation
- `Cubical.Categories.WithFamiliesCubical`: improvement defining the naturality condition for CwF as a heterogenous path
- `Cubical.Categories.WithFamiliesAdHoc`: improvement using ad-hoc functions instead of transport. This is the most complete formalization so far
- Various minor changes can be found throughout the library, notably in the `Cubical.Functions.Embedding` module.

Conclusion

In this thesis, we presented the notion of iterative sets as introduced by D. Gratzer, H. R. Gylterud, A. Mörtberg, and E. Stenholm [7]. We changed some of their proofs to make them more suitable for the use with cubical type theory, and ultimately formalized the treatment using the proof assistant Cubical Agda in conjunction with the agda-cubical library. Moreover, we formalized the definitions of Categories with Families as well as Σ-structure in three ways, a naive version following the prior formalization and two suggested improvements, the latter of which significantly improving goal legibility and computational behaviour. We then implemented the CwF and Σ-structure for the category $\mathcal{V}$ of iterative sets for all three versions, leaving open only the naturality condition of the equivalence that is part of the definition of Σ-structure. While we ultimately failed to complete the proof, we still managed to substantially improve the final goal to a degree that makes us confident that a proof will be possible.

The obvious next goal is to try to finish the formalization. Apart from trying to simplify some definitions, making them opaque and disabling Agda's universe level checker, one could also think about using a different notion of categorical model of dependent type theory altogether. While Categories with Families in theory offer good practical benefits, they rely on the category of elements in a way that requires us to use transport for terms and context extensions. One could therefore explore whether other notions of models might improve the implementation for iterative sets. E.g., one could look at displayed categories [2] and natural models [4], but it needs to be seen if this translates to actual improvements in our particular setting. While natural models are conceptually simpler and closer to the definition of Categories with Families, one would need to formalize all its definitions from scratch, whereas there already exists some pioneering work about displayed categories in the agda-cubical library.

Another interesting research problem beyond the present formalization of Σ-structure would be to explore other natural ways how the category $\mathcal{V}$ can be used to derive models of dependent type theory. This notably includes $\mathcal{V}$-valued presheaves and their expressivity compared to standard hSet-valued presheaves. In addition one can explore how the fact that $\mathcal{V}$ is itself a set (as opposed to hSet) manifests itself in other areas and applications.

Bibliography

[1] P. Aczel, "The Type Theoretic Interpretation of Constructive Set Theory," in *Studies in Logic and the Foundations of Mathematics*, vol. 96, 1978. doi: 10.1016/S0049-237X(08)71989-X.

[2] B. Ahrens and P. L. Lumsdaine, "Displayed Categories," *Logical Methods in Computer Science*, vol. 15, no. 1, Mar. 2019, doi: 10.23638/LMCS-15(1:20)2019.

[3] C. Angiuli and D. Gratzer, "Principles of Dependent Type Theory (Manuscript)," Jul. 19, 2025. [Online]. Available: https://www.danielgratzer.com/papers/type-theory-book.pdf

[4] S. Awodey, "Natural Models of Homotopy Type Theory," *Mathematical Structures in Computer Science*, vol. 28, no. 2, Feb. 2018, doi: 10.1017/S0960129516000268.

[5] C. Cohen, T. Coquand, S. Huber, and A. Mörtberg, "Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom," in *21st International Conference on Types for Proofs and Programs (TYPES 2015)*, Mar. 2018. doi: 10.4230/LIPIcs.TYPES.2015.5.

[6] P. Dybjer, "Internal Type Theory," in *Types for Proofs and Programs (TYPES 1995)*, 1996. doi: 10.1007/3-540-61780-9_66.

[7] D. Gratzer, H. R. Gylterud, A. Mörtberg, and E. Stenholm, "The Category of Iterative Sets in Homotopy Type Theory and Univalent Foundations," *Mathematical Structures in Computer Science*, vol. 34, no. 9, 2024, doi: 10.1017/S0960129524000288.

[8] F. L. Grubmüller and Various Upstream Contributors, *My agda-cubical Fork*. [Online]. Available: https://github.com/flgrubm/cubical/tree/mt-submission

[9] H. R. Gylterud, "Multisets in Type Theory," *Mathematical Proceedings of the Cambridge Philosophical Society*, vol. 169, no. 1, Jul. 2020, doi: 10.1017/S0305004119000045.

[10] H. R. Gylterud and E. Stenholm, "Univalent Material Set Theory," *Annals of Pure and Applied Logic*, vol. 177, no. 2, 2026, doi: 10.1016/j.apal.2025.103662.

[11] M. Hofmann, "Syntax and Semantics of Dependent Types," Jan. 30, 1997. doi: 10.1017/CBO9780511526619.004.

[12] M. Hofmann and T. Streicher, "The Groupoid Interpretation of Type Theory," in *Twenty Five Years of Constructive Type Theory*, 1998. doi: 10.1093/oso/9780198501275.003.0008.

[13] P. Martin-Löf, "Intuitionistic Type Theory," *Bibliopolis*.

[14] A. Mörtberg, "Cubical Methods in Homotopy Type Theory and Univalent Foundations," *Mathematical Structures in Computer Science*, vol. 31, no. 10, Nov. 2021, doi: 10.1017/S0960129521000311.

[15] E. Riehl, *Category Theory in Context*. 2017. [Online]. Available: https://math.jhu.edu/~eriehl/161/context.pdf
[16] E. Riehl, "On The ∞-Topos Semantics of Homotopy Type Theory," Feb. 02, 2024. doi: 10.1112/blms.12997.
[17] E. Rijke, *Introduction to Homotopy Type Theory*. 2025. doi: 10.1017/9781108933568.
[18] E. Rijke, E. Stenholm, J. Prieto-Cubides, F. Bakke, V. Štěpančík, and Various Contributors, *The agda-unimath Library*. [Online]. Available: https://github.com/UniMath/agda-unimath
[19] E. Stenholm, "Material Set Theory in Homotopy Type Theory," Doctoral dissertation, 2024.
[20] The Univalent Foundations Program, *Homotopy Type Theory: Univalent Foundations of Mathematics*. 2013. [Online]. Available: https://homotopytypetheory.org/book
[21] Various Contributors, *The Lean Proof Assistant*. [Online]. Available: https://leanlang.org/
[22] Various Contributors, *The Rocq Proof Assistant*. [Online]. Available: https://rocq-prover.org/
[23] Various Contributors, *The Agda Proof Assistant*. [Online]. Available: https://github.com/agda/agda
[24] Various Contributors, *The rocq-unimath Library*. [Online]. Available: https://github.com/UniMath/UniMath
[25] Various Contributors, *The agda-cubical Library*. [Online]. Available: https://github.com/agda/cubical
[26] Various Contributors, *The 1Lab Library*. [Online]. Available: https://1lab.dev/
[27] A. Vezzosi, A. Mörtberg, and A. Abel, "Cubical Agda: A Dependently Typed Programming Language With Univalence and Higher Inductive Types," *Journal of Functional Programming*, vol. 31, 2021, doi: 10.1017/S0956796821000034.
[28] V. Voevodsky, "The Equivalence Axiom and Univalent Models of Type Theory," 2024. [Online]. Available: https://doi.org/10.48550/arXiv.1402.5556