
The Presheaf Model of Type Theory in Type Theory

Élise Souche
Stockholm University
2026

M1 Internship report
Supervised by Anders Mörtberg
with collaboration with Fabian L. Grubmüller

Abstract

Iterative sets form a universe of sets that is itself a set in homotopy type theory. The category of presheaves valued in iterative sets can be made into a category with families and thus a model of type theory. This model furthermore supports type formers. In this work, we formalize this construction in Cubical Agda. We therefore obtain this classic model of type theory, inside of homotopy type theory.

Contents

Introduction	1
1 Preliminaries	2
1.1 Dependent type theory	2
1.1.1 Core theory	2
1.1.2 Dependent functions	2
1.1.3 Universes	2
1.1.4 Simple types	3
1.1.5 Dependent pairs	3
1.1.6 Equality	3
1.2 Univalence	4
1.2.1 The univalence axiom	4
1.2.2 The h-level hierarchy	4
1.2.3 Cubical type theory	5
1.3 Modeling type theory	5
1.4 Agda	6
1.4.1 Cubical Agda	6
1.4.2 Example of interactive proving in Agda	7
2 Internal models of type theory	8
2.1 Categories with Families	8
2.2 Type structures	11
2.2.1 Σ -structure	11
2.2.2 Π -structure	12
2.2.3 Other structures	12
3 The set and presheaf models	12
3.1 Sets in homotopy type theory	12
3.1.1 Iterative sets	12
3.1.2 Tarski universes	13
3.2 The set model	14
3.3 The presheaf model	15
3.3.1 Core CwF	15
3.3.2 The Π structure of the presheaf model	16
3.3.2.1 The Tarski universe case	17
4 Future work	17
Conclusion	17
Bibliography	18
Appendix A – A category C such that $\hat{C}$ doesn't have Π -types	19

Introduction

Dependent *type theory* is a syntactic, logical system, that is by nature constructive. Whereas set theory, built on first-order logic, is a material theory, in the sense that sets all are the same kind of object and can belong to many different sets, type theory is structural: each term has one type, and the rules of the theory describe how to build terms of certain types from other terms in other types. Type theory enjoys good computational properties that make it suitable for computer implementations, unlike set theory.

Homotopy type theory [1] is a new type theory that can be used as alternative foundations for mathematics. It provides a logically constructive framework that supports *univalence*. Homotopy type theory is classically built on Martin-Löf intensional type theory with additional axioms. A new theory called *cubical type theory* [2] promises to make homotopy type theory computational again.

Iterative sets were introduced by P. Aczel [3] as a way of encoding constructive set theory in type theory. They mimic the notion of a cumulative hierarchy of sets and are defined as a W-type, i.e. as well-founded trees. This result was later extended to homotopy type theory by H. R. Gylterud [4]. In homotopy type theory, the usual notion of a set is the homotopy set. But iterative sets have the benefit of forming a kind of universe of sets, that is itself a (homotopy) set.

The semantics of type theory are often described in terms of *categories with families*. One common model is the set model, in which types are sets. Another one has types be presheaves [5]. Different indexing categories have allowed sophisticated and useful models to be devised, notably ones supporting univalence [6]. For instance, models using simplicial sets have helped give geometrical intuition to type theory [7].

Computer *proof assistants* are languages in which mathematics can be written. Proofs can then be mechanically checked by the system. Developing mathematics inside such assistants promises to guarantee the correctness of proofs, as well as creating libraries of organized and searchable formalized mathematics. Most of these proof assistants are based on type theory.

Proof assistants in use today include Rocq [8], Lean [9], and Agda [10], which indeed all implement a type theory. They include features beyond simply proof checking: they permit interactive development of proofs, gradually making progress and refining the holes. They also provide proof automation techniques that allow automatically proving simple goals. Agda implements cubical type theory via the “Cubical Agda” mode. Additionally, libraries of mathematics formalized in this cubical framework have been developed [11], [12].

D. Gratzer, H. Gylterud, A. Mörtberg, and E. Stenholm [13] showed that iterative sets form a category with families akin to the set model of type theory, and formalized this result. They however failed to formalize some type formers. In this work, we formalize in Cubical Agda the notion of presheaves valued in iterative sets, and show that they form a category with families supporting the usual type constructors. This gives rise to a powerful model of type theory internal to type theory. However, we fail, for now, to support Π -types.

Note on formalization The Agda code can be found on GitHub¹. Throughout this text, we include clickable links to it. These manifest as little Agda logos that can be found near definitions or theorems, they look like .

¹<https://github.com/flgrubm/fun-with-cwf>

1 Preliminaries

We begin by presenting the calculus of dependent type theory, before studying its semantics. This is a bit long but is a necessary preliminary to be able to understand both what we study and formalize, and the metatheory in which we do it.

1.1 Dependent type theory

Dependent type theory [14], [15] is a logical system that can be seen as a generalization of the simply-typed λ -calculus. Like how the Curry-Howard correspondence says simple types are like propositional logic, it says that dependent types are like predicate logic. I will only give a brief and informal description.

1.1.1 Core theory

The main ingredients are contexts $(\Gamma, \Delta, \dots)$, terms $(a, b, \dots)$, and types $(A, B, \dots)$, which are all defined in terms of each other. We also have variables $(x, y, \dots)$. In a fixed context, types can be introduced with the judgement $\Gamma \vdash A$ type. Then each term is introduced with exactly one type, a relationship expressed by the judgement $\Gamma \vdash a : A$. Contexts are lists $(x_0 : A_0, \dots, x_n : A_n)$ such that $(x_0 : A_0, \dots, x_i : A_i) \vdash A_{i+1}$ type. There is also an empty context $\langle \rangle$.

There is a relation denoted $=$ called definitional equality. Two terms are equal, if you can compute one from the other, using rules like the β -reduction familiar in λ -calculus. For instance we will have $\emptyset \vdash 1 + 1 = 2 : \mathbb{N}$, once we have defined these natural numbers. All rules will always be compatible with definitional equality, as expressed by conversion rules like

$$\frac{\Gamma \vdash A = B \text{ type} \quad \Gamma \vdash a : A}{\Gamma \vdash a : B}$$

From now this kind of conversion rules will be omitted.

This theory can be used both as a programming language and as a logical system. As a programming language, it will be similar to the λ -calculus. As a logical system, the Curry-Howard correspondence allows us to see the types as propositions, and terms as proofs of them. It is in this sense that the logic is constructive: the rules show how to build proofs.

1.1.2 Dependent functions

There are dependent functions, or Π -types. Unlike functions in the simply typed calculus, here we allow the return type to depend on the argument. This is denoted by $B(x)$ to indicate that B is some expression where x appears and can be substituted.

$$\frac{\Gamma, x : A \vdash b : B(x)}{\Gamma \vdash \lambda x. b : \Pi_{x:A} B(x)} \quad \frac{\Gamma \vdash f : \Pi_{x:A} B(x) \quad \Gamma \vdash a : A}{\Gamma \vdash f a : B(a)}$$

We have the familiar β -reduction rule

$$\frac{\Gamma, x : A \vdash b : B(x) \quad \Gamma \vdash a : A}{\Gamma \vdash (\lambda x. b) a = b[a/x] : B(a)}$$

We can recover simple functions: we can define $A \rightarrow B := \Pi_{x:A} B$. We will use the Agda notation $(x : A) \rightarrow B \ x := \Pi_{x:A} B(x)$.

1.1.3 Universes

There is a type of types

$$\frac{\Gamma \vdash A \text{ type}}{\Gamma \vdash A : \mathcal{U}} \quad \frac{\Gamma \vdash A : \mathcal{U}}{\Gamma \vdash A \text{ type}}$$

One central point here is the status of $\mathcal{U}$. Is it an actual type? Should we have $\mathcal{U} : \mathcal{U}$?

This “type-in-type” hypothesis turns out to cause a paradox known as Girard’s paradox [14]. The solution is to say that $\mathcal{U}$ contains “small” types, but is itself “large”. This leads to introducing a full hierarchy of universes $\mathcal{U}_0 : \mathcal{U}_1 : \mathcal{U}_2 : \dots$, with all the basic types in $\mathcal{U}_0$.

With $\mathcal{U}$ in hand, we can now see dependent types as dependent functions into $\mathcal{U}$. For instance we can imagine a family of types indexed by natural numbers $B : \mathbb{N} \rightarrow \mathcal{U}$, and a dependent function of type $(n : \mathbb{N}) \rightarrow B\ n$.

1.1.4 Simple types

We illustrate the case of booleans:

$$\frac{\overline{\Gamma \vdash 2 \text{ type}} \quad \overline{\Gamma \vdash \mathbf{true} : 2} \quad \overline{\Gamma \vdash \mathbf{false} : 2} \quad \overline{\Gamma \vdash P : (b : 2) \rightarrow \mathcal{U}} \quad \overline{\Gamma \vdash t : P \ \mathbf{true}} \quad \overline{\Gamma \vdash f : P \ \mathbf{false}}}{\overline{\Gamma \vdash \text{elim}_2(P, t, f) : (x : 2) \rightarrow Px}} \text{elim}_2$$

$$\overline{\Gamma \vdash \text{elim}_2(P, t, f) \ \mathbf{true} = t} \quad \overline{\Gamma \vdash \text{elim}_2(P, t, f) \ \mathbf{false} = f}$$

The rule elim_2 is an induction principle. This kind of rule is everywhere in type theory. We won’t spell out the rules for the empty type $\perp$ or the unit $\mathbb{1}$, which are structurally similar. In fact, we can see that most types are defined in the same way. We have

- formation rules that state when this type exists
- introduction rules that state how to build terms of it
- elimination rules that state how to use terms of it
- computation rules that state how to transform terms

The natural numbers have more interesting constructors and eliminators:

$$\frac{\overline{\Gamma \vdash \mathbb{N} \text{ type}} \quad \overline{\Gamma \vdash 0 : \mathbb{N}} \quad \overline{\Gamma, x : \mathbb{N} \vdash S\ x : \mathbb{N}} \quad \overline{\Gamma \vdash P : (n : \mathbb{N}) \rightarrow \mathcal{U}} \quad \overline{\Gamma \vdash P_0 : P\ 0} \quad \overline{\Gamma \vdash P_S : (n : \mathbb{N}) \rightarrow (P\ n \rightarrow P\ (S\ n))}}{\overline{\Gamma \vdash \text{elim}_{\mathbb{N}}(P, P_0, P_S) : (n : \mathbb{N}) \rightarrow P\ n}} \text{elim}_{\mathbb{N}}$$

$$\overline{\Gamma \vdash \text{elim}_{\mathbb{N}}(P, P_0, P_S)\ 0 = P_0} \quad \overline{\Gamma \vdash \text{elim}_{\mathbb{N}}(P, P_0, P_S)\ (S\ n) = P_S\ n\ (\text{elim}_{\mathbb{N}}(P, P_0, P_S)\ n)}$$

$\text{elim}_{\mathbb{N}}$ is exactly the induction scheme for natural numbers in Peano arithmetic.

1.1.5 Dependent pairs

Similarly to how we allowed the return type of functions to depend on the argument, we also allow the type of second component of a pair to depend on the first component. We denote such dependent pairs by $\Sigma_{a:A} Ba$, and can recover the non-dependent pairs as $A \times B := \Sigma_{\cdot:A} B$. We will also use the notation $(a : A) \times Ba$, which shows more clearly that these are dependent pairs.

$$\overline{\Gamma \vdash (a, b) : (a : A) \times Ba} \quad \overline{\Gamma \vdash \pi_1 : (a : A) \times Ba \rightarrow A} \quad \overline{\Gamma \vdash \pi_2 : (x : (a : A) \times Ba) \rightarrow B(\pi_1 x)}$$

$$\overline{\Gamma \vdash \pi_1(a, b) = a : A} \quad \overline{\Gamma \vdash \pi_2(a, b) = b : Ba} \quad \overline{\Gamma \vdash (\pi_1 x, \pi_2 x) = x : (a : A) \times Ba}$$

1.1.6 Equality

Generally speaking, the most important relation to reason about is equality. We want to be able to prove things about equality inside of our type theory, when we use it as a proof assistant. Using our propositions-as-types intuition, we would thus introduce a new type $\text{Eq}(A, a, b)$ for all $a, b : A$. Then, we can add rules to our type theory saying that this type is inhabited if and only if a and b are equal: $\text{Eq}(A, a, b) = \{\star \mid a = b\}$. Unfortunately, such an extensional notion of equality makes typechecking undecidable. In classical Martin-Löf type theory, it is usually intensional identity that is used. The idea is to define equality inductively. There is a constructor for reflexivity, refl , and there is an eliminator which basically says that, when we reason on an

equality, it suffices to work on the case where it is refl. As we work in Cubical Agda, where equality works differently, we will not detail this more.

Generally, the identity type of $a, b : A$ will be noted $a \equiv b$.

1.2 Univalence

We will now briefly introduce homotopy type theory (HoTT), at an informal level. Its core idea is based on the observation that types, when equipped with the identity type, form ∞ -groupoids. Indeed, we have objects in the type A , and identifications (terms of $a \equiv b$) form morphisms. We can see that they compose, associate, and are invertible, with refl as identity morphism. Thus A can be seen as a groupoid. Furthermore, given two paths $p, q : a \equiv b$, we can also form the type $p \equiv q$. We thus have an infinite tower of groupoids, an ∞ -groupoid.

According to Grothendieck’s homotopy hypothesis, ∞ -groupoids are spaces. Thus, we are led to consider types as spaces, where terms are points and identities are paths. Non-dependent functions are continuous maps. We can also consider homotopy fibers and homotopy equivalences. In fact, many homotopy-theoretic notions naturally translate into type theory, and vice-versa!

1.2.1 The univalence axiom

Consider a universe $\mathcal{U}$. It is itself a type, i.e. a space, and thus we can consider paths in it. How do these relate to homotopy equivalences? It is easy to create an equivalence $A \simeq B$ from a path $A \equiv B$. But there is no obvious way to go in the other direction. Thus, we introduce the *univalence axiom*, which asserts that $(A \simeq B) \equiv (A \equiv B)$.

The intuition here is that univalence captures the practice of identifying isomorphic things. In mathematics, it is common to reason “up to equivalence”. The univalence axiom makes this notion precise. Adding univalence to type theory has allowed the development of the field of synthetic homotopy theory, a novel approach to homotopy theory. More generally, HoTT can be used as foundations for mathematics, replacing set theory [1].

1.2.2 The h-level hierarchy

As we said, the core idea of HoTT is that types are ∞ -groupoids. This makes equality a complicated notion with a lot of higher structure. In mathematics, it is common to define a notion by data and some property of the data. For instance, a bijection is a function (data) that is bijective (property). What really matters is the data. But in HoTT, this is more complicated. For instance, if there are two different proofs that the function is bijective, then there are two different bijections with the same underlying function! We need a notion of the “complicatedness” of a type, something that illustrates how much higher structure it has. This notion is the h-level, which is defined by induction.

At the bottom of the hierarchy are contractible types, defined by this predicate:

$$\text{isContr } A = (a : A) \times ((x : A) \rightarrow x \equiv a)$$

So a type is contractible if it is, homotopically, a disk, with a center and paths between it and all points. Alternatively, they can be seen as singletons (up to a path).

h-levels are then defined by

$$\begin{aligned} \text{isOfHLevel } 0 \ A &= \text{isContr } A \\ \text{isOfHLevel } (\text{succ } n) \ A &= (x \ y : A) \rightarrow \text{isOfHLevel } n \ (x \equiv y) \end{aligned}$$

meaning that a type is at level $n + 1$ if its paths are at level n . So h-level 1 contains types where equality is contractible, i.e. all points have exactly one path between them (up to homotopy). These types are thus either empty or singletons, and are called homotopy propositions, h-props. Most things we would call logical “propositions” or “properties” are indeed h-props. At h-level 2 are types where equality is propositional: there is at most one way in which two terms are equal.

These are called homotopy sets, h-sets. For instance the type of natural numbers is an h-set. At h-level 3 are h-groupoids, and so on. Note that types at h-level n are also at h-level $n + 1$.

1.2.3 Cubical type theory

In the classic formulation of HoTT [1], univalence is introduced as an axiom on top of Martin-Löf type theory. However, this causes computational issues: terms involving univalence do not reduce. A famous example [16] is a proof in HoTT that $(n : \mathbb{N}) \times (\pi_4(\mathbb{S}^3) \simeq \mathbb{Z}/n\mathbb{Z})$. The actual value of n cannot be known because the evaluation of the proof term gets stuck.

Cubical type theory [2] is a type theory that makes univalence computational. One of its other benefits is that it also gives computational content to higher inductive types.

Informally, the idea is to add an interval type $\mathbb{I}$, with two endpoints i_0 and i_1 , and take equality $a \stackrel{A}{=} b$ to be paths, i.e. maps $p : \mathbb{I} \rightarrow A$ subject to the boundary condition that $p \ i_0 = a$ and $p \ i_1 = b$. Other primitives allow manipulation of interval variables, composition of paths, and transport. Finally, a primitive, glue types, can be added to obtain univalence, without any non-computational axioms.

In this particular work, the use of cubical type theory did not particularly help, though it may prove useful in future work, see Section 4. Still, most of the other attempts to formalize these models of type theory are done assuming uniqueness of identity proofs or rewrite rules: we show that these are not needed, illustrating the possibilities offered by univalent type theory.

1.3 Modeling type theory

Fundamentally, the notion of model of type theory is mostly bureaucracy. A model is a set of contexts, a set of types, and a set of terms, with many laws and axioms relating them; it is just a translation of the syntactical rules.

Definition 1 Model of type theory [15]

A *model* $\mathcal{M}$ is

- a set Ctx of *contexts*
- for $\Gamma, \Delta \in \text{Ctx}$, a set $\text{Sb}(\Gamma, \Delta)$ of *substitutions*
- for $\Gamma \in \text{Ctx}$, a set $\text{Ty } \Gamma$ of *types*
- for $\Gamma \in \text{Ctx}, A \in \text{Ty } \Gamma$, a set $\text{Tm } \Gamma A$ of *terms*
- an *empty context* $\langle \rangle \in \text{Ctx}$
- for $\Gamma \in \text{Ctx}, A \in \text{Ty } \Gamma$, a *context extension* $\Gamma \triangleright A \in \text{Ctx}$

There are substitution operators:

- for $\sigma \in \text{Sb}(\Delta, \Gamma)$ and $A \in \text{Ty } \Gamma$, a type $A[\sigma] \in \text{Ty } \Delta$
- for $\sigma \in \text{Sb}(\Delta, \Gamma)$ and $a \in \text{Tm } \Gamma A$, a term $a[\sigma] \in \text{Tm } \Delta A[\sigma]$

Substitutions can be composed. There is an identity substitution and composition is associative. We ask that $A[\sigma \cdot \sigma'] = A[\sigma][\sigma']$. Context extension must also respect substitution. Substituting from $\Gamma \triangleright A$ must be equivalent to substituting from Γ and providing an object in $\text{Tm } \Gamma A$.

This only models the core of type theory. There are no type constructors like Π -types, for instance; this only captures the core calculus of substitution and typing. But it already has a lot of rules, and in fact writing them down explicitly takes a lot of time! Fortunately, we can observe that the language of category theory allows us to express all this very concisely using the notion of *category with families* [17], [18].

We denote by $\hat{C}$ the category of presheaves on C , i.e the functor category $C^{\text{op}} \rightarrow \mathbf{Set}$, and by $\int F$ the category of elements of a presheaf F .

Definition 2 Category with Families (CwF)

A *category with families* is a category Ctx of *contexts* and *substitutions*, equipped with

- a terminal object $\langle \rangle : \text{Ctx}$, the *empty context*
- a presheaf $\text{Ty} : \widehat{\text{Ctx}}$ of *types*
- a presheaf $\text{Tm} : \widehat{\int \text{Ty}}$ of *terms*

such that for all $\Gamma, A \in \int \text{Ty}$, the following presheaf is representable:

$$\text{Ext } (\Gamma, A) : \text{Ctx}^{\text{op}} \rightarrow \mathbf{Set}$$

$$\Delta \mapsto (\sigma : \Delta \rightarrow \Gamma) \times \text{Tm } \Delta (A[\sigma])$$

$$\tau \mapsto (j, i) \mapsto j \circ \tau, i[\tau]$$

Using categories and presheaves allows us to capture type and term substitution concisely. It is perhaps less obvious that the representability of Ext captures the required properties of context extension. The idea is that Ext maps each context to what we want $\Delta \rightarrow \Gamma \triangleright A$ to be, and its representability makes $\text{Ext } \Delta$ indeed equivalent to $\Delta \rightarrow \Gamma \triangleright A$.

1.4 Agda

Agda [10] is a dependently typed functional programming language. Alternatively, it is a proof assistant based on dependent type theory. Its syntax resembles that of Haskell, with which it also shares a powerful pattern matching feature. Among other proof assistants, it is distinguished by its lack of a tactic language. Proofs are written entirely as terms in type theory. While this prevents the extensive proof automation techniques found in e.g. Rocq, it is very useful when working on type theory itself rather than using it as foundations for other kinds of mathematics.

By default, Agda implements a flavor of Martin-Löf type theory. But it also has a cubical mode, implementing cubical type theory.

1.4.1 Cubical Agda

We briefly describe the primitives of Cubical Agda, showing how they differ from normal, MLTT-based Agda. For a more detailed account see C. Cohen, T. Coquand, S. Huber, and A. Mörtberg [2] or A. Vezzosi, A. Mörtberg, and A. Abel [19].

At the core of the theory is the interval type $\mathbb{I}$ (actually it is not really a type, but it mostly behaves like one). It can be visualized as the interval $[0, 1]$, and, like it, it has two endpoints $i0$ and $i1$.

The most important notion is that of dependent paths. A dependent path is a kind of heterogeneous equality, between terms of two types that have a path between them:

$$\text{PathP} : (A : \mathbb{I} \rightarrow \text{Type}) (a : A \ i0) (b : A \ i1) \rightarrow \text{Type}$$

Such paths are what equality means in cubical type theory. This captures the homotopy point of view that equalities are paths: paths in a space A in classic set-based homotopy theory are defined as continuous maps $[0, 1] \rightarrow A$.

To build a term in $\text{PathP } A \ a \ b$, we must introduce a path variable i then build a term $a_i : A \ i$, subject to the boundary condition that when i is $i0$, a_i must be definitionally equal to a , and similarly for the other endpoint of the interval, $i1$. In practice, this looks like defining a term $(i : \mathbb{I}) \rightarrow A \ i$, which we can do with a λ abstraction. The difference is that Agda checks the boundary conditions. Normal non-dependent paths are then defined as $a \equiv b := \text{PathP } (\lambda _ \rightarrow A) \ a \ b$.

We have reflexivity:

$$\text{refl} : \{A : \text{Type}\} \{a : A\} \rightarrow a \equiv a$$

$$\text{refl } \{A\} \{a\} \ i = a$$

$\mathbb{I}$ behaves like a De Morgan algebra: there are operations $\vee, \wedge, \sim$ with the expected laws of distributivity, commutativity, etc. The only ones that don't hold are that $i \wedge \sim i$ is not $i0$, and $i \vee \sim i$ is not $i1$: we want them to behave like a minimum or maximum on the interval. For instance, we can define the opposite of a path:

$$\begin{aligned} \text{sym} &: \{A : \text{Type}\} \{a \ b : A\} \rightarrow a \equiv b \rightarrow b \equiv a \\ \text{sym } p \ i &= p \ (\sim i) \end{aligned}$$

In normal Agda with inductive equality, the transport operation is defined by induction as

$$\begin{aligned} \text{transport} &: \{A \ B : \text{Type}\} \rightarrow A \equiv B \rightarrow A \rightarrow B \\ \text{transport refl } x &= x \end{aligned}$$

But in cubical type theory we don't have induction on equality. Transport is thus based on a primitive operation of generalized transport

$$\text{transp} : (A : \mathbb{I} \rightarrow \text{Type}) \rightarrow (r : \mathbb{I}) \rightarrow A \ i0 \rightarrow A \ i1$$

with the condition, checked by Agda, that A is a constant family if r is $i1$. We have the equality $\text{transp } A \ i1 \ a = a$, which is why this condition exists. Transport can then be defined as

$$\begin{aligned} \text{transport} &: \{A \ B : \text{Type}\} \rightarrow A \equiv B \rightarrow A \rightarrow B \\ \text{transport } p \ a &= \text{transp } (\lambda i \rightarrow p \ i) \ i0 \ a \end{aligned}$$

With these in hand we can already do a lot of the things allowed in normal Agda! There is still however one problem: path composition. In normal Agda we have

$$\begin{aligned} _ \bullet _ &: \{A : \text{Type}\} \rightarrow \{a \ b \ c : A\} \rightarrow a \equiv b \rightarrow b \equiv c \rightarrow a \equiv c \\ \text{refl } \bullet \text{ refl} &= \text{refl} \end{aligned}$$

which again relies on path induction. The solution in cubical type theory is complicated and long, and we will not detail it. It involves a calculus of interval formulas and subtypes, and a new primitive, defined manually for each type former, called `hcomp` that allows extending cubes to give them higher dimensions. For the most part, these internals are not used in day-to-day use of Cubical Agda. What matters is that we recover the $_ \bullet _$ operation.

The reward for all this complexity is that we can add primitives for a computational univalence, but we will not detail them either. We also gain computational function extensionality, as will be shown in Section 1.4.2. However, we also lose some definitional equalities that hold in normal Agda but not in the cubical mode:

- $\text{transport refl } a$ is not definitionally a .
- It is possible to define path induction, but again it is not definitional.
- $\text{refl } \bullet \text{ refl}$ is not definitionally refl .

These equalities still hold up to a path. But, when they are encountered in a proof, we must manually work around them instead of Agda reducing them automatically.

1.4.2 Example of interactive proving in Agda

As a little example of both cubical type theory and practical Agda usage in the Emacs text editor, let us prove function extensionality. We provide a type signature and start writing the proof, leaving it as a hole for now.

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

`{!!}` is a hole that we must fill. Pressing C-c C-, with the cursor in the hole tells us about it:

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

Let's press C-c C-c inside the hole to introduce the variables f , g , the proof p , and an interval variable i . In Cubical Agda, the syntax of paths, and the allowed pattern matchings, closely resemble normal functions.

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

Pressing C-c C-, tells us what the current goal is:

```
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
```

The term we will write in the hole must satisfy the boundary condition. This means that, when we substitute $i = i0$, it must reduce to $\lambda x \rightarrow f x$. Such boundaries are what distinguish paths from normal functions.

Let's press C-c C-c to introduce x . Then we can finally fill the hole with the term $p x i$, pressing C-c C-r. Agda typechecks and ensures that this term satisfies the boundaries. We end up with this proof:

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

This very simple proof means that function extensionality computes very well in cubical type theory, unlike in classic MLTT-based HoTT where it is either postulated or proven from univalence and in either case does not reduce.

2 Internal models of type theory

The goal of this internship was to formalize the presheaf model of type theory, as well as the (iterative) set model in passing. First, we describe how we define categories with families.

2.1 Categories with Families

Previous attempts at formalizing the CwF of iterative sets [13] got stuck when trying to build the additional type formers, as the calculus of substitutions required complex paths and transports. In his master thesis, F. L. Grubmüller [20] tried adapting their work to Cubical Agda in the hope of this path algebra becoming more tractable, but also got stuck. It appears that this problem was very dependent on the precise statement of the notion of CwF. To try to address this, we used a very unfolded definition, exposing every coherence rule, and then proved this version to be equivalent to the usual, more packaged-up categorical definitions of CwF.

First, let's see how to formalize the categorical version.

Theorem 1

The representability of $\text{Ext}(\Gamma, A)$ is equivalent to requiring

- an object $\Gamma \triangleright A \in \text{Ctx}$
- $p : \Gamma \triangleright A \rightarrow \Gamma$
- $q : \text{Tm}(\Gamma \triangleright A)(A[p])$
- that the following function is an equivalence:

$$\begin{aligned} \langle p, q \rangle : (\tau : \Delta \rightarrow \Gamma \triangleright A) &\rightarrow (\sigma : \Delta \rightarrow \Gamma) \times \text{Tm} \Delta (A[\sigma]) \\ \tau &\mapsto p \circ \tau, q[\tau] \end{aligned}$$

Proof.

$\Rightarrow$. By definition, representability of Ext means that there is an object $\Gamma \triangleright A$ and a natural isomorphism $\iota : \text{Hom}(-, \Gamma \triangleright A) \simeq \text{Ext}$. By Yoneda's lemma this natural transformation ι gives an object $(p, q) \in \text{Ext}(\Gamma \triangleright A)$, defined by $(p, q) := \iota(\text{id}_{\Gamma \triangleright A})$. Furthermore, given $\tau : \Delta \rightarrow \Gamma \triangleright A$, naturality makes the following diagram commute:

$$\begin{array}{ccccc} & & \langle p, q \rangle \tau & \xleftarrow{\quad} & (p, q) \\ & \nearrow & & & \uparrow \\ \iota_{\Delta}(\tau) & & \text{Ext } \Delta & \xleftarrow{- \circ \tau, -[\tau]} & \text{Ext}(\Gamma \triangleright A) \\ \uparrow & \nearrow \iota_{\Delta} & & \nearrow \iota_{\Gamma \triangleright A} & \\ \Delta \rightarrow \Gamma \triangleright A & \xleftarrow{- \circ \tau} & \Gamma \triangleright A \rightarrow \Gamma \triangleright A & & \\ \uparrow & & & & \uparrow \\ \tau & \xleftarrow{\quad} & \text{id}_{\Gamma \triangleright A} & & \end{array}$$

So $\langle p, q \rangle = \iota$, which is an equivalence.

$\Leftarrow$. The proof follows pretty much the same steps: unfold the definition of the Yoneda lemma to build the natural transformation, draw the same diagram, observe that we have an isomorphism. ■

We will use this version as our “ground truth” definition of a CwF , and call it a *categorical CwF* (CCwF). It can be expressed in Agda $\hookrightarrow$ as:

```
record CwF (C : Category) : Type where
  field
    () : Terminal C
    Ty : Presheaf C
    Tm : Presheaf (∫ Ty)
    _▷_ : (Γ : Ctx) (A : Ty[ Γ ]) → Ctx
    p   : {Γ : Ctx} {A : Ty[ Γ ]} → Γ ▷ A → Γ
    q   : {Γ : Ctx} {A : Ty[ Γ ]} → Tm[ Γ ▷ A , A [ p ]Ty ]

  ⟨p,q⟩ : {Γ Δ : Ctx} (A : Ty[ Γ ])
    → (Δ → Γ ▷ A) → Σ[ σ ∈ Δ → Γ ] Tm[ Δ , A [ σ ]Ty ]
  ⟨p,q⟩ {Δ = Δ} A τ =
    p ∘ τ , subst (λ T → Tm[ Δ , T ]) (sym (funExt- (Ty .F-seq p τ) A)) (q [ τ ]Tm)

  field
    ctxExtRepr : {Γ Δ : Ctx} (A : Ty[ Γ ]) → isEquiv ((p,q) {Γ} {Δ} A)
```

Functors, natural transformations, and other categorical notions are constructions that package together both computational and logical content. We unfold all that information and obtain the *algebraic CwF* (ACwF) $\hookrightarrow$, inspired by [21]:

```

record CwF (C : Category) : Type where
  field
    () : Terminal C
    Ty : (Γ : Ctx) → Type
    Tm : (Γ : Ctx) (A : Ty Γ) → Type

    isSetTy : (Γ : Ctx) → isSet (Ty Γ)
    isSetTm : (Γ : Ctx) (A : Ty Γ) → isSet (Tm Γ A)

    -- substitutions
    _[_]Ty : (A : Ty Γ) (σ : Δ → Γ)
      → -----
      Ty Δ
    [id]Ty : (A : Ty Γ)
      → -----
      A [ id ]Ty ≡ A
    [][_]Ty : (A : Ty Γ) (σ' : θ → Δ) (σ : Δ → Γ)
      → -----
      A [ σ ∘ σ' ]Ty ≡ (A [ σ ]Ty) [ σ' ]Ty
    _[_]Tm : {A : Ty Γ} (a : Tm Γ A) (σ : Δ → Γ)
      → -----
      Tm Δ (A [ σ ]Ty)
    [id]Tm : {A : Ty Γ} (a : Tm Γ A)
      → -----
      PathP (λ i → Tm Γ ([id]Ty A i)) (a [ id ]Tm) a
    [][_]Tm : {A : Ty Γ} (a : Tm Γ A) (σ' : θ → Δ) (σ : Δ → Γ)
      → -----
      PathP (λ i → Tm θ ([][_]Ty A σ' σ i))
        (a [ σ ∘ σ' ]Tm)
        (a [ σ ]Tm [ σ' ]Tm)

    -- context extension
    _▷_ : (Γ : Ctx) (A : Ty Γ) → Ctx
    p : {A : Ty Γ} → Γ ▷ A → Γ
    q : {A : Ty Γ} → Tm (Γ ▷ A) (A [ p ]Ty)

    -- now, a bunch of coherence laws
    _+ : {A : Ty Γ} (σ : Δ → Γ)
      → -----
      Δ ▷ A [ σ ]Ty → Γ ▷ A
    ( _ ) : {A : Ty Γ} (a : Tm Γ A)
      → -----
      Γ → (Γ ▷ A)
    ( ) ∘ : {A : Ty Γ} (a : Tm Γ A) (σ : Δ → Γ)
      → -----
      ( a ) ∘ σ ≡ σ + ∘ ( a [ σ ]Tm )
    p ∘ (q) ≡ id : {A : Ty Γ}
      → -----
      p + ∘ ( q ) ≡ id {Γ ▷ A}
    ∘+ : {A : Ty Γ} (σ' : θ → Δ) (σ : Δ → Γ)
      → -----
      PathP (λ i → θ ▷ [][_]Ty A σ' σ i → Γ ▷ A) ((σ ∘ σ') +) (σ + ∘ σ' +)
    id+ : {A : Ty Γ}
      → -----
      PathP (λ i → Γ ▷ [id]Ty A i → Γ ▷ A) (id +) id
    p ∘+ : {A : Ty Γ} (σ : Δ → Γ)
      → -----
      p {A = A} ∘ σ + ≡ σ ∘ p
    [p][+]Ty : {A : Ty Γ} (B : Ty Γ) (σ : Δ → Γ)
      → -----

```

$$\begin{array}{l}
\text{q}[+] \text{Tm} : \{A : \text{Ty } \Gamma\} (\sigma : \Delta \rightarrow \Gamma) \\
\rightarrow \text{PathP } (\lambda i \rightarrow \text{Tm } (\Delta \triangleright A [\sigma] \text{Ty}) ([p][+] \text{Ty } A \sigma i)) (q [\sigma +] \text{Tm}) q \\
\text{p} \circ \langle \rangle \equiv \text{id} : \{A : \text{Ty } \Gamma\} (a : \text{Tm } \Gamma A) \\
\rightarrow \text{p} \circ \langle a \rangle \equiv \text{id} \\
[p][\langle \rangle] \text{Ty} : \{A : \text{Ty } \Gamma\} (B : \text{Ty } \Gamma) (a : \text{Tm } \Gamma A) \\
\rightarrow \text{B } [p] \text{Ty } [\langle a \rangle] \text{Ty} \equiv B \\
\text{q}[\langle \rangle] \text{Tm} : \{A : \text{Ty } \Gamma\} (a : \text{Tm } \Gamma A) \\
\rightarrow \text{PathP } (\lambda i \rightarrow \text{Tm } \Gamma ([p][\langle \rangle] \text{Ty } A a i)) (q [\langle a \rangle] \text{Tm}) a
\end{array}$$

None of the fields require transports, only dependent paths. This trend will continue in type constructors and is a main strength of this definition, although even with it some complicated path algebra remains.

The exact set of coherence laws to require is perhaps not obvious at first glance, hence the following theorem:

Theorem 2



Given a category C , the types $\text{CCwF } C$ and $\text{ACwF } C$ are equivalent.

Proof.

The proof is somewhat long and tedious, but nothing really interesting happens. Most of the equalities are in h-sets, making things easier. We refer the reader to the formalization. ■

Despite its simplicity on paper, the formalized proof features quite a lot of annoying path algebra, especially in the direction from CCwF to ACwF . We therefore used Claude Opus 5 to fill most holes in it, which it managed very well.

Note that it is necessary that types and terms form h-sets for the metatheory to be well-behaved [22]. In particular, it is necessary for equality of terms to be decidable, which is important in some models. Furthermore, even if it wasn't the case, the fact that some equalities are in h-sets helps tremendously in many places in the formalization, and in fact it is not known how we could do without them.

2.2 Type structures

The CwF we described only models the core calculus of type theory, which states how to manipulate contexts and substitutions. We want to support all the types we outlined: Π -types, natural numbers, etc. In this section, we describe how to add these additional structures to an ACwF on Ctx .

2.2.1 Σ -structure

A Σ -structure is the necessary data for a CwF to support Σ -types. For the definition below, remember for $a : \text{Tm } \Gamma A$, there is $\langle a \rangle : \Gamma \rightarrow \Gamma \triangleright A$.

Definition 3 Σ -structure



A Σ -structure is an assignment of $A : \text{Ty } \Gamma$, $B : \text{Ty } (\Gamma \triangleright A)$ to some $\Sigma AB : \text{Ty } \Gamma$, that is natural in A and B . There must be a natural isomorphism $\text{Tm } \Gamma (\Sigma AB) \simeq (a : \text{Tm } \Gamma A) \times \text{Tm } \Gamma B[\langle a \rangle]$.

2.2.2 Π -structure

A Π -structure on a CwF allows it to represent Π -types.

Definition 4 Π -structure



A Π -structure is an assignment of $A : \text{Ty } \Gamma$, $B : \text{Ty } (\Gamma \triangleright A)$ to a type $\Pi A B : \text{Ty } \Gamma$ that is natural in A and B , with a natural isomorphism $\text{Tm } \Gamma (\Pi A B) \simeq \text{Tm } (\Gamma \triangleright A) B$.

2.2.3 Other structures

Other type formers of interest are unit types, empty types, natural numbers, and booleans. These should be quite straightforward, but we have not yet formalized them. There is also equality, in the many different formulations of it. We have formalized extensional equality, which is defined by

Definition 5 Eq-structure



An Eq-structure is an assignment of $A : \text{Ty } \Gamma$, $a, b : \text{Tm } \Gamma A$ to a type $\text{Eq } A a b : \text{Ty } \Gamma$, natural in all variables, and with a natural isomorphism $\text{Tm } \Gamma (\text{Eq } A a b) \simeq a \equiv b$.

3 The set and presheaf models

We now formalize two models of type theory in Cubical Agda: the set model and the presheaf model. These cannot be straightforwardly formalized internally in type theory. The reason is that we need a universe of sets that is itself a set, as we explained in Section 2.1, to be able to build the Ty and Tm presheaves.

3.1 Sets in homotopy type theory

We have mentioned a notion of set in HoTT , the h-set . Homotopy sets behave very much like sets: in fact, h-sets are a ΠW -pretopos, and with the propositional resizing axiom, they form an elementary topos [23]. Assuming the axiom of choice, we actually get Lawvere’s elementary theory of the category of sets [24]. However, the problem we encounter with h-sets is that the type of h-sets is not itself an h-set [1].

Fortunately, other notions of set in type theory have been proposed. In this work we use the notion of iterative sets, whose type does form a h-set .

3.1.1 Iterative sets

We will use Aczel’s iterative sets [3], which predate HoTT . The idea is to define sets in a bottom-up fashion. Each set will be a family of sets (sitting lower in the hierarchy) indexed by some type. The type of iterative sets is thus defined as

$$V^\infty = W_{A : \text{Type}} A$$

or in Agda syntax

```
data V∞ where
  sup : (A : Type) → (f : A → V∞) → V∞
```

The canonical elements of this inductive type are of the form $\text{sup } A f$ where $A : \text{Type}$ and $f : A \rightarrow V^\infty$. This is interpreted as a set whose elements are other sets and are indexed by A . We can inductively define index and elements by

$$\begin{aligned}
&\text{index} : V^\infty \rightarrow \text{Type} \\
&\text{index} (\sup A f) = A \\
&\text{elements} : (X : V^\infty) \rightarrow \text{index } X \rightarrow V^\infty \\
&\text{elements} (\sup A f) = f
\end{aligned}$$

H. R. Gylterud [4] realized that in the context of HoTT, these form not sets, but multisets. Indeed, the map $f : A \rightarrow V^\infty$ can map multiple elements of A to the same set, counting it multiple times. To recover sets, we will truncate this map to prevent it from picking the same element multiple times.

The multiplicity of an element X of Y can be defined as

$$X \in^\infty Y := \text{fiber}_{\text{elements } Y} X$$

We can then say that Y is an n -set if all its fibers are n -types and its elements are transitively n -sets. In particular, we are most interested in V^0 , which sits at the bottom of this hierarchy and is defined by

$$\begin{aligned}
&\text{is0Set} : V^\infty \rightarrow \text{Type} \\
&\text{is0Set} (\sup A f) = \text{isEmbedding } f \times ((a : A) \rightarrow \text{is0Set } (f a)) \\
&V^0 : \text{Type} \\
&V^0 = (X : V^\infty) \times \text{is0Set } X
\end{aligned}$$

The predicate is0Set is an h-prop. We extend the operations index , elements , and $\in$ to iterative sets. A core first result about V^0 is that it is equivalent to embeddings into itself. Then, we can recall one very important theorem, which shows that V^0 supports the set-theoretical axiom of extensionality.

Theorem 3 H. R. Gylterud [4]

For $x, y : V^0$, we have

$$x \equiv y \simeq (z : V^0) \rightarrow (z \in x \leftrightarrow z \in y)$$

Proof.

The core of the proof is the *fibration identity principle* [4], where fibrations into a type A are the elements of the type $\text{Fib } A = (X : \text{Type}) \times (X \rightarrow A)$:

$$(X, f) \equiv (Y, g) \simeq (a : A) \rightarrow \text{fiber}_f a \simeq \text{fiber}_g a$$

This can then be applied to V^0 since it is equivalent to $\text{Emb}(V^0)$. Finally, remember that $\in$ is defined in terms of fibers and that since we have embeddings, these fibers are propositional; therefore, equivalence becomes biimplication. ■

More broadly, while it is not our goal here, V^0 permits developing material set theory internally in HoTT [25].

3.1.2 Tarski universes

A Tarski universe, generally speaking, is a type of types. More precisely, it is a type of *codes* for types, equipped with a decoding function that can generate a type from a code. For our purposes, we will furthermore require that the universe and its decodings are h-sets, giving the following definition of a Tarski universe $\mathcal{U}$:

```

record TarskiUniverse-Base : Type where
  field
    U : Type
    El : U → Type
    isSetU : isSet U
    isSetEl : (x : U) → isSet (El x)

```

The motivation for this is the following:

Theorem 4



V^0 is a Tarski universe.

Proof.

For the decoding function, we take $\text{El} := \text{index} : V^0 \rightarrow \text{Type}$. The fact that V^0 is an h-set follows from the extensional characterization of equality in V^0 , because elementhood is a proposition. The setness of decodings comes from the fact that, by definition, they embed into V^0 , which is an h-set. ■

The above theorem might come as a surprise: V^0 is a kind of set of (small) sets! This is the feature that motivates its use in our work and distinguishes it from regular h-sets. We will see later that the setness conditions are essential to be able to define categories with families.

A raw Tarski universe is not really interesting. In fact, the empty type is a Tarski universe! We want it to contain at least some basic types. At minimum, we will require that all our Tarski universes possess Unit-structures and Σ -structures. We require codes for unit and pairs, and that their decodings are isomorphic to the corresponding types $\hookrightarrow$.

```
record TarskiUniverse-Unit : Type where
  field
    Unit : U
    isContrElUnit : isContr (El Unit)
record TarskiUniverse-Sig : Type where
  field
    Sig : (A : U) → (El A → U) → U
    SigIso : (A : U) (B : El A → U)
      → Iso (El (Sig A B)) (Σ[ x ∈ El A ] El (B x))
```

Iterative sets support these type formers $\hookrightarrow$. But there are other examples of Tarski universes, for instance the finite sets form one $\hookrightarrow$.

Just like there is a category of sets, each Tarski universe U gives rise to a category $\mathcal{U}$ with codes as objects and $\text{hom}(a, b) = \text{El } a \rightarrow \text{El } b$. Like **Set**, it has finite products and a terminal object, given by the Σ and unit structures $\hookrightarrow$.

3.2 The set model

The set model bridges the gap between set theory and type theory by allowing us to view types as sets. In it, contexts are sets, types are families of sets indexed by the context (so, in the empty context, types are sets), and terms are their elements. Substitutions are functions. Context extension is dependent pairing.

Theorem 5



For every Tarski universe U , $\mathcal{U}$ can be equipped with an ACwF structure $\text{UCwF } U$, where

- $\langle \rangle = \text{Unit}$
- $\text{Ty } \Gamma = \text{El } \Gamma \rightarrow U$
- $\text{Tm } \Gamma A = (\rho : \text{El } \Gamma) \rightarrow \text{El } (A \rho)$
- $\Gamma \triangleright A = \text{Sig } \Gamma A$

Proof.

Most of the coherence laws are proven by refl or applications of the Sig isomorphism.

We also specialized the proof to the case of iterative sets $\mathcal{U}$, where the isomorphism is definitional. There, everything is refl. ■

Theorem 6



- U-CwF U has a Σ -structure
- If U has a Π -structure, U-CwF U has a Π -structure
- If U has an Eq-structure, U-CwF U has an Eq-structure

Proof.

Here too, we specialized the proofs to $V^0 \mathcal{U}$, and they were extremely simple. For general Tarski universes we must in addition weave the isomorphisms around. The case of Σ is the only one to pose a difficulty, being the only place in the definition of our structures where transports remain. ■

For comparison, previous attempts to formalize the CwF of iterative sets had been unable to construct Σ -types [13], [20]. Our approach based on the equivalence of ACwF and CCwF makes things much easier by replacing most transports with dependent paths.

3.3 The presheaf model

We now turn to a more complicated and interesting model. Let C be a category. Here, contexts are presheaves on C , types are presheaves on the category of elements of the context (so that in the empty context they are presheaves on C), and terms are pointwise elements satisfying certain restriction conditions [5]. The choice of C yields various useful models: simplicial sets model MLTT-based HoTT, and cubical sets model cubical type theory [6]. Formalizing these important results internally in type theory is thus of interest.

3.3.1 Core CwF

Let us fix C a category, U a Tarski universe. We denote $\hat{C} := C^{\text{op}} \rightarrow U$ the presheaves valued in U . The implicit variables $I, J, \dots$ are objects of C .

We can then prove our first main result:

Theorem 7



There is an ACwF on $\hat{C}$ defined by

- $\langle \rangle$ is the constant presheaf with value Unit
- $\text{Ty } \Gamma$ is the set of presheaves $(\int \Gamma)^{\text{op}} \rightarrow U$
- $\text{Tm } \Gamma A$ is the set of natural transformations from the unit type to A , where the unit type in context Γ is the constant presheaf with value Unit
- Context extension is defined by $(\Gamma \triangleright A)I := (\rho : \text{El } (\Gamma I)) \times A(I, \rho)$.

Proof.

Most of the equalities to check can be computed by hand and translate quite straightforwardly to Agda. The hardest part is to prove that $\Gamma \triangleright A$ is a presheaf. But again, the difficulties are mostly technical once we have the following lemma $\mathcal{U}$:

$$\begin{aligned}
\text{F-hom-PathP} &: \{x \ x' \ y \ y' : \int \Gamma\} \\
&\rightarrow (f : (\int \Gamma)[x, y]) (g : (\int \Gamma)[x', y']) \\
&\rightarrow (x \equiv x' : x \equiv x') (y \equiv y' : y \equiv y') \\
&\rightarrow \text{PathP } (\lambda i \rightarrow C[y \equiv y' i . \text{fst}, x \equiv x' i . \text{fst}]) (f . \text{fst}) (g . \text{fst}) \\
&\rightarrow \text{PathP } (\lambda i \rightarrow \text{El } (A (x \equiv x' i)) \rightarrow \text{El } (A (y \equiv y' i))) \\
&\quad (A f) (A g)
\end{aligned}$$

where A is a functor from $\int \Gamma$ to $\mathcal{U}$.

The proof is again much easier when specialized to V^0 , as many more equalities are definitional $\mathcal{U}$. ■

The formulation of terms as natural transformations can be seen to be equivalent to the unfolded definition where terms are families of elements of the type satisfying a restriction condition [5]:

$$\text{Tm } \Gamma A = \{M : (I : \mathcal{C})(\rho : \Gamma I) \rightarrow A(I, \rho) \mid \forall f : J \rightarrow I, M I \rho = M J (\Gamma f \rho)\}$$

but turned out to be easier to manipulate in Agda.

We can then start adding type formers:

Theorem 8



The CwF on $\hat{C}$ has a Σ -structure.

Proof.

It is defined pointwise by $(\Sigma AB)(I, \rho) = (a : A(I, \rho)) \times B(I, (\rho, a))$, which can be shown to be functorial. This proof is another example of something that is easy on paper but different in Cubical Agda because of complex path algebra.

Interestingly, specializing to V^0 seemed to be much easier. Unfortunately, we were unable to complete the specialized formalization for V^0 because of a performance issue in the implementation of iterative sets. The proofs in U require a lot of weaving the SigIso in and out. Again, we used Claude to fill in almost all of these technical holes. ■

3.3.2 The Π structure of the presheaf model

Unlike for the Σ -structure, the Π -structure needs additional coherences with respect to restriction. So, at stage (I, ρ) , we want to define [5]

$$\begin{aligned}
(\Pi AB)(I, \rho) &:= \{w : (J : \mathcal{C})(f : J \rightarrow I)(a : A(J, \Gamma f \rho)) \rightarrow B(J, (\Gamma f \rho, a)) \\
&\quad \mid \forall g : K \rightarrow J, Bg(wfa) = w(f \cdot g)(Aga) \\
&\quad \} \\
&= \int_{f : J \rightarrow I} (a : A(J, \Gamma f \rho)) \rightarrow B(J, (\Gamma f \rho, a))
\end{aligned}$$

In the reformulation as an end, the “integration variable” is actually (J, f) in the slice category $C_{/I}$. This equality follows from the characterization of equalizers in **Set**.

There are two things at play here. First, quantifying over all f is what gives functoriality, making this into a presheaf (and thus a type). If we simply defined, stagewise,

$$(\Pi AB)(I, \rho) = (a : A(I, \rho)) \rightarrow B(I, (\rho, a))$$

there would be no way to go from $(\Pi AB)(I, \rho)$ to $(\Pi AB)(J, \Gamma f \rho)$. Second, restricting the family w to add the coherence condition is what gives the equivalence $\text{Tm } \Gamma(\Pi AB) \simeq \text{Tm } (\Gamma \triangleright A)B$. In the forward direction, we have the map

$$\begin{aligned} \text{Tm } \Gamma(\Pi AB) &\rightarrow \text{Tm } (\Gamma \triangleright A)B \\ w \mapsto (I, (\rho, a)) &\mapsto w (I, \rho) \text{ id } a \end{aligned}$$

The coherence condition is precisely the naturality condition that terms must respect. The reformulation as an end captures both properties.

3.3.2.1 The Tarski universe case

Unfortunately, we fail to express this construction in type theory. Now, instead of sets, we have either iterative sets or Tarski codes. The issue is that the family w is indexed by morphisms $f : J \rightarrow I$: we need to quantify over J and $J \rightarrow I$. We must build a Tarski code from this quantification.

So, we would need to consider indexing categories C whose set of objects and hom-sets are not h-sets but Tarski codes. This is not morally a problem: for all the instances we have in mind, like simplicial sets, the indexing categories are simple enough that we can describe them in iterative sets or some other Tarski universes. See Appendix A for a counterexample of an indexing category and a Tarski universe where there is no Π -structure $\mathcal{U}$. The work to formalize this notion of internal category and construct the Π -structure in this case is currently in progress.

As a side note, assuming the axiom of choice, all h-sets are iterative sets [13] (more precisely, the decoding function $\text{El}^0 : V^0 \rightarrow \mathcal{U}$ is essentially surjective). The axiom of choice would thus allow us to use any indexing category C . This also illustrates why our counterexample uses finite sets instead of iterative sets: we would need to do this metatheoretically, using a model of cubical type theory without choice.

4 Future work

There are a number of type formers that we have not yet formalized. Both the set model and the presheaf model could be extended with unit types, empty types, natural numbers, and booleans. These should not be very hard to add. The presheaf model should also have its Π -structure added. We are currently working on these.

In the future, we could develop more metatheory. In particular, one interesting development could be the formalization of the syntactic model and the proof of its initiality. This can be done using higher inductive-inductive types [22]; the fact that cubical type theory allows computational higher inductive types could be useful. Eventually, the end goal of all this is to formalize the standard results of consistency, canonicity, normalization, etc.

Conclusion

During this internship, we were for the first time able to formalize the set and the presheaf model of type theory, as well as some of their type formers as a CwF in homotopy type theory. However, we encountered a limitation of our use of iterative sets and Tarski universes, which somewhat limits which indexing categories can be used, even though it should not be a problem for most presheaf models.

Our use of cubical type theory over MLTT+univalence seems to provide little benefit for now, but may be useful if we try to study the initial model. We gain better support for dependent paths without transports, but lose definitional path induction and transports, so it is not clear if it is really better or worse.

Bibliography

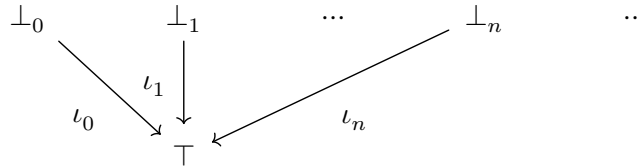
- [1] The Univalent Foundations Program, *Homotopy Type Theory: Univalent Foundations of Mathematics*. arXiv, 2013. doi: 10.48550/ARXIV.1308.0729.
- [2] C. Cohen, T. Coquand, S. Huber, and A. Mörtberg, “Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom,” *LIPICs, Volume 69, TYPES 2015*, vol. 69, p. 5:1–5:34, 2018, doi: 10.4230/LIPICs.TYPES.2015.5.
- [3] P. Aczel, “The Type Theoretic Interpretation of Constructive Set Theory*,” in *Logic Colloquium '77*, vol. 96, A. Macintyre, L. Pacholski, and J. Paris, Eds., in *Studies in Logic and the Foundations of Mathematics*, vol. 96., Elsevier, 1978, pp. 55–66. doi: 10.1016/S0049-237X(08)71989-X.
- [4] H. R. Gylterud, “From Multisets to Sets in Homotopy Type Theory,” *The Journal of Symbolic Logic*, vol. 83, no. 3, pp. 1132–1146, Sep. 2018, doi: 10.1017/jsl.2017.84.
- [5] M. Hofmann, “Syntax and Semantics of Dependent Types,” in *Semantics and Logics of Computation*, A. M. Pitts and P. Dybjer, Eds., in *Publications of the Newton Institute.*, Cambridge: Cambridge University Press, 1997, pp. 79–130. doi: 10.1017/CBO9780511526619.004.
- [6] T. Coquand, “A Survey of Constructive Presheaf Models of Univalence,” *ACM SIGLOG News*, vol. 5, no. 3, pp. 54–65, Jul. 2018, doi: 10.1145/3242953.3242962.
- [7] C. Kapulkin and P. L. Lumsdaine, “The Simplicial Model of Univalent Foundations (after Voevodsky),” *Journal of the European Mathematical Society*, vol. 23, no. 6, pp. 2071–2126, Mar. 2021, doi: 10.4171/JEMS/1050.
- [8] The Rocq Development Team, “The Rocq Prover.” Accessed: Jun. 23, 2026. [Online]. Available: <https://rocq-prover.org/>
- [9] L. de Moura and S. Ullrich, “The Lean 4 Theorem Prover and Programming Language,” in *Automated Deduction – CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings*, Berlin, Heidelberg: Springer-Verlag, Jul. 2021, pp. 625–635. doi: 10.1007/978-3-030-79876-5_37.
- [10] Agda Developers, “Agda.” Accessed: Jun. 23, 2026. [Online]. Available: <https://agda.readthedocs.io/>
- [11] The Agda Community, “Cubical Agda Library.” Accessed: Jun. 23, 2026. [Online]. Available: <https://github.com/agda/cubical>
- [12] The 1Lab Development Team, “The 1Lab.” Accessed: Jun. 23, 2026. [Online]. Available: <https://1lab.dev/>
- [13] D. Gratzer, H. 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, pp. 945–970, Oct. 2024, doi: 10.1017/S0960129524000288.
- [14] P. Martin-Löf, “An Intuitionistic Theory of Types: Predicative Part,” in *Logic Colloquium '73*, vol. 80, H. E. Rose and J. C. Shepherdson, Eds., in *Studies in Logic and the Foundations of Mathematics*, vol. 80., Elsevier, 1975, pp. 73–118. doi: 10.1016/S0049-237X(08)71945-1.
- [15] C. Angiuli and D. Gratzer, “Principles of Dependent Type Theory.” [Online]. Available: <https://www.carloangiuli.com/papers/type-theory-book.pdf>
- [16] G. Brunerie, “The James Construction and $\pi_4(\mathbb{S}^3)$ in Homotopy Type Theory,” *Journal of Automated Reasoning*, vol. 63, no. 2, pp. 255–284, Aug. 2019, doi: 10.1007/s10817-018-9468-2.
- [17] P. Dybjer, “Internal Type Theory,” in *Types for Proofs and Programs*, vol. 1158, S. Berardi, M. Coppo, G. Goos, J. Hartmanis, and J. Leeuwen, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 120–134. doi: 10.1007/3-540-61780-9_66.

- [18] S. Castellan, P. Clairambault, and P. Dybjer, “Categories with Families: Untyped, Simply Typed, and Dependently Typed.” Accessed: Aug. 06, 2026. [Online]. Available: <http://arxiv.org/abs/1904.00827>
- [19] A. Vezzosi, A. Mörtberg, and A. Abel, “Cubical Agda: A Dependently Typed Programming Language with Univalence and Higher Inductive Types,” *Proceedings of the ACM on Programming Languages*, vol. 3, no. ICFP, pp. 1–29, Jul. 2019, doi: 10.1145/3341691.
- [20] F. L. Grubmüller, “The Category of Iterative Sets in Cubical Agda,” 2026. [Online]. Available: <https://grubmueller.dev/publications/2026-mt-iterative-sets/>
- [21] T. Altenkirch, A. Kaposi, and S. Xie, “The Groupoid-Syntax of Type Theory Is a Set.” Accessed: Aug. 03, 2026. [Online]. Available: <http://arxiv.org/abs/2509.14988>
- [22] T. Altenkirch and A. Kaposi, “Type Theory in Type Theory Using Quotient Inductive Types,” in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, in POPL '16. New York, NY, USA: Association for Computing Machinery, Jan. 2016, pp. 18–29. doi: 10.1145/2837614.2837638.
- [23] E. Rijke and B. Spitters, “Sets in Homotopy Type Theory,” *Mathematical Structures in Computer Science*, vol. 25, no. 5, pp. 1172–1202, Jun. 2015, doi: 10.1017/S0960129514000553.
- [24] F. W. Lawvere, “An Elementary Theory of the Category of Sets,” *Proceedings of the National Academy of Sciences*, vol. 52, no. 6, pp. 1506–1511, Dec. 1964, doi: 10.1073/pnas.52.6.1506.
- [25] H. R. Gylterud and E. Stenholm, “Univalent Material Set Theory,” *Annals of Pure and Applied Logic*, vol. 177, no. 2, p. 103662, Feb. 2026, doi: 10.1016/j.apal.2025.103662.

Appendix A – A category C such that $\hat{C}$ doesn't have Π -types

Let's construct an example $\mathcal{U}$ of a Tarski universe U and a category C such that there is no Π -structure for the CwF on $C^{\text{op}} \rightarrow U$.

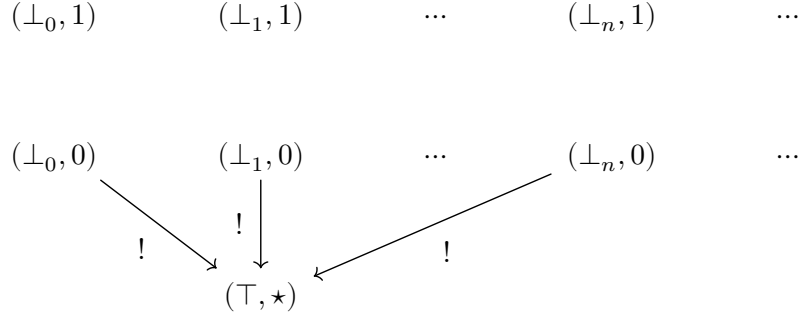
For U , we take the universe of finite sets, which has codes for Π types. For C , we take an infinite cone with singletons $\{\iota_i\}$ for hom-sets:



Now let us choose some Γ, A, B such that the existence of $\Pi A B$ is a contradiction.

For Γ , we take the empty context. We have the following lemma: terms of a type in Γ are only defined by their value at $\top$. This is imposed by the naturality condition of terms $\mathcal{U}$. In particular, since we are in finite sets, a type in Γ only has a finite number of terms, as the set of values at $\top$ is finite.

For $A : \text{Ty } \Gamma$, we take 2 for each $\perp$, and 1 for the $\top$. Therefore, $\Gamma \triangleright A$ looks like



Now we define $B : \text{Ty } (\Gamma \triangleright A)$ as the constant functor that is 2 everywhere. Terms in B must have one constant value in the cone part, but have ω free choices on the unconnected parts. We can define an infinite family of distinct terms $t_n : \text{Tm } (\Gamma \triangleright A)B$ by

$$\begin{aligned}
t_n (\top, _) &= 0 \\
t_n (\perp_i, 0) &= 0 \\
t_n (\perp_i, 1) &= 1 \text{ if } n \equiv i \text{ else } 0
\end{aligned}$$

We have that $\text{Tm } (\Gamma \triangleright A)B \simeq \text{Tm } \Gamma(\Pi AB)$ by definition of the Π -structure. But $\text{Tm } (\Gamma \triangleright A)B$ is infinite while $\text{Tm } \Gamma(\Pi AB)$ is finite, so we have a contradiction.