Formalizing Dependent Types in Agda

INF513 Final Report

March 25, 2024

Yee-Jian Tan
yee-jian.tan@polytechnique.edu

Contents

1 Introduction

1.1 Dependent Types

In type theory, dependent types is a formal system used in proof assistants such as Coq and Agda, whose logical power is given by the Curry-Howard correspondence. Being at the top of the lambda cube, it allows types to depend on terms, and it is exactly this dependency that gives dependent types its expressivity, unfortunately as well as its complexity. Many noteworthy properties of dependent types, such as injectivity and normalization are crucial for it to be useful as the theory behind proof assistants.

There are many pen-and-paper proofs of these meta-theoretic properties of dependent types, but the pen-and-paper proofs are exactly what we try to eliminate by the use of proof assistants. Therefore, we want to formalize dependent types in itself, not only for showing its flexibility as a formal system, but also for the possibility of proving meta-theoretic properties using itself (minus consistency as prohibited by Gödel's second incompleteness theorem).

1.2 Previous Works

Many projects tried to formalize various flavors of dependent types by a proof assistant. A picture by Meven Lennon-Bertrand summarizes some existing efforts well:

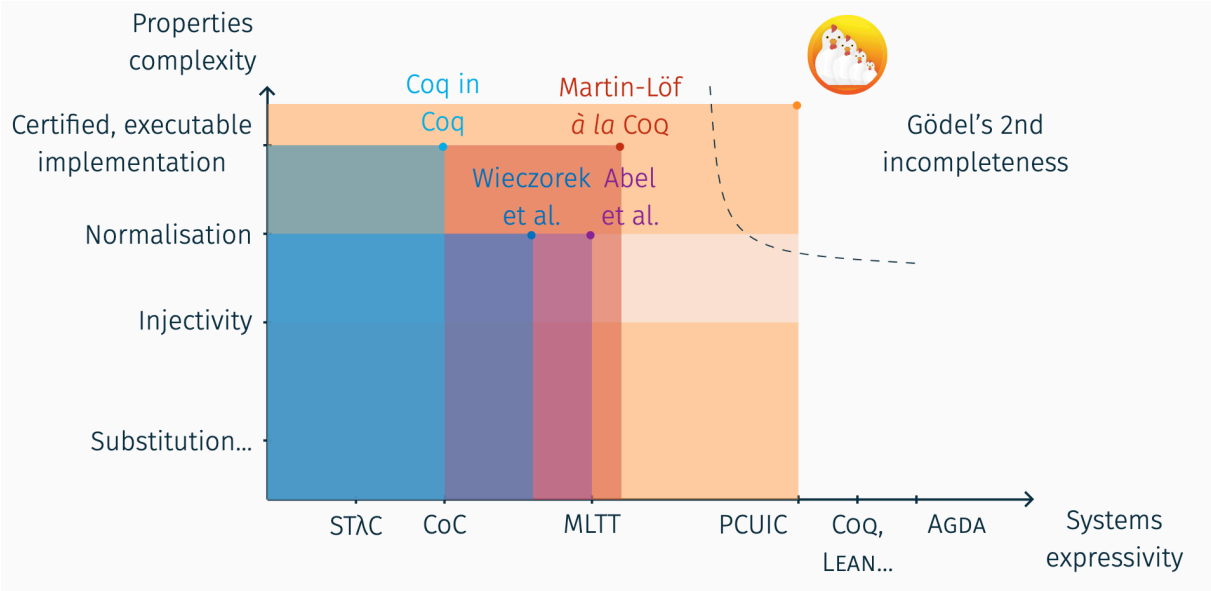

Figure 1: Previous attempts in formalizing type theory in type theory. [1]

Regardless of the properties proven, and the dependent type systems that were formalized, the attempts in Figure 1 share the same property: they take a purely syntactic approach in the formalization, starting from defining the freely-generated family of pre-terms, before selecting the well-typed terms, finally defining the semantics (reduction and conversion rules) manually.

Multiple other approaches exist: Ambrus Kaposi had an implementation of a flavor of dependent types via Categories with Families (CwFs) in Agda [2], and Guillaume Brunerie implemented dependent types via contextual categories, which was another theory of dependent types that is more abstract than CwFs, in Agda as well. Theories such as CwFs have the advantage of being a semantic model, it subsumes the syntax by just considering the syntax model, which is the initial model. A more comprehensive overview is available at Section 2.1.

Despite all these attempts, there is no one best solution for this situation, and a good formalization and how to overcome the underlying difficulties is still an open topic in the community.

1.3 Summary of results

This project is an attempt to understand the difficulties underlying the formalization of dependent types, and explore possible solutions. In this project, I have formalized in the Agda proof assistant two representations of dependent types:

- Categories with Families (CwFs) in Agda, with dependent function space, unit type, and empty type.
- A less-indexed (or less-dependent) version of CwFs with the same features as the previous, additionally with dependent pair types and booleans.

The takeaways from this project include

- A systematic study of Categories with Families (CwFs)
- The preliminary category theory knowledge needed to understand CwFs
- General understanding of the Generalized Algebraic Theories (GATs) and models of type theory
- Familiarity with the Agda proof assistant in implementing type theories
- A comparison of known approaches in formalizing dependent types
- Insights about transport hell

The remainder of the report is organized as follows: Section 2 gives an overview of the different semantic models of dependent type, and defines Categories of Families, the core semantic model we

formalized. Section 3 explains transport hell, the main difficulty faced in this project, Section 4 introduces the less-indexed version of CwFs defined, a possible solution that we explored. Finally, Section 5 summarizes the project and gives possible future work.

2 Categories with Families

2.1 Semantic models of dependent types

The operational semantics of dependent types is traditionally defined using inference rules. They have the advantage of being more straightforward but are relatively verbose to formalize. In comparison, categorical semantics of dependent types have the advantage of being more succinct and less complicated and thus pose advantages in formalization. There are many categorical models of dependent type that we know of.

Although the models represent the same formal system of dependent types, they differ slightly in the presentation, according to how closely they resemble the the syntax of dependent types that we are used to, which is the concretely defined collection of terms, types, contexts, and substitutions. It is also known as the "term model" or the "initial model" of these semantic models, since, in every such semantic model, the syntax is just a particular model.

Below is a diagram showing various categorical models of dependent types, and arranged according to their closeness to syntax.

$$\leftarrow \textbf{Semantic} - \text{Locally cartesian-closed categories (lccc)} - \text{Contextual Categories}$$
$$- \text{Awodey's Natural Model} - \text{CwF} - \text{E-CwF} - \textbf{Syntactic} \rightarrow$$

Categories with Families (CwF) was first defined by Peter Dyjber as a categorical model of dependent types [3], and has a representation that is rather close to the syntax.

We will define CwFs, and then show how we can interpret the syntax of Dependent Types in the CwF semantics. Despite that, the construction of CwF will be explicit enough to see its intended counterpart in dependent types.

2.2 Preliminary concepts in category theory

To be self-contained, we start by giving the prerequisite definitions in category theory to define CwFs. We merely state the definitions here: a more complete treatment can be found in any category theory textbook.

Definition 2.2.1 (Category): A **category** $\mathscr{C}$ consists of
- a collection $\mathrm{Ob}(\mathscr{C})$ (or simply $\mathscr{C}$) of **objects**, written as $A \in \mathscr{C}$, and
- a collection $\mathrm{Hom}_{\mathscr{C}}$ of **arrows** or **morphisms** between objects, written as $f : A \to B \in \mathrm{Hom}_{\mathscr{C}}(A, B)$, and
- a binary operation $\_ \circ \_ : \mathrm{Hom}_{\mathscr{C}}(A, B) \times \mathrm{Hom}_{\mathscr{C}}(B, C) \to \mathrm{Hom}_{\mathscr{C}}(A, C)$ between arrows.
- for every object A in $\mathscr{C}$, a morphism $\mathrm{id}_A : A \to A$

Satisfying the following axioms:
- *(identity)* for every object A in $\mathscr{C}$, id_A is both the left and right identity for the binary operation $\_ \circ \_$.
- *(associativity)* $\_ \circ \_$ is associative.

Definition 2.2.2 (Initial, terminal objects): An object A in a category $\mathscr{C}$ is said to be **initial** if for any other object $B \in \mathscr{C}$, there is a morphism $A \to B$.

An object A in a category $\mathscr{C}$ is said to be **terminal** if for any other object $B \in \mathscr{C}$, there is a morphism $B \to A$.

Definition 2.2.3 (Opposite Category): For every category $\mathscr{C}$, the **opposite category** $\mathscr{C}^{\mathrm{op}}$ is the category resulting from "reversing" all the arrows in $\mathscr{C}$ and keeping the objects intact.

One can think of functors as structure-preserving maps between categories.

Definition 2.2.4 (Functor): A **functor** $F : \mathscr{C} \to \mathscr{D}$ between two categories $\mathscr{C}$ and $\mathscr{D}$ does the following:

- associates to every object $A \in \mathscr{C}$, an object $\mathrm{FA} \in \mathscr{D}$
- associates to every morphism $h : A \to B \in \mathrm{Hom}_{\mathscr{C}}$, a morphism $\mathrm{Fh} : \mathrm{FA} \to \mathrm{FB} \in \mathrm{Hom}_{\mathscr{D}}$

satisfying

- for every object $A \in \mathscr{C}$, $F\mathrm{id}_A = \mathrm{id}_{\mathrm{FA}}$
- for every $h : A \to B$ and $g : B \to C$ in $\mathscr{C}$, $F(g \circ h) = F(g) \circ F(h)$.

2.3 CwF with Π-Types

We first give an intuition behind the definitions. Firstly, the relationship between types and the terms of a given type can be thought of as indexed sets,

$$\mathscr{T} = \{T : T \text{ type}\}, \mathrm{Tm}_T = \mathrm{Tm}(T) = \{t : t \text{ has type } T\}$$

Where Tm is a family of sets indexed by $\mathscr{T}$. Families of sets are the collection of indexed sets, expressed as a pair, and is the source of **types** and **terms**, where the terms are indexed by a type. This is defined in Definition 2.3.1.

Definition 2.3.1 (Families of sets): The category Fam contains

- as objects: pairs $\left(A, (B_a)_{a \in A}\right)$ where A is a set, and (B_a) a set for each $a \in A$.
- as morphisms: a morphism $\left(A, (B_a)_{a \in A}\right) \to \left(A', \left(B'_{a'}\right)_{a' \in A'}\right)$ consists of a reindexing function $h : A \to A'$, and a family of functions $h_a : B_a \to B_{f(a)}$ for each $a \in A$.

Then, the definition of a CwF is twofold: firstly, we require a category of **contexts** and **substitutions**, and "connect" them with the **types** and **terms** compatible with such contexts, via a functor. This can be thought of picking a family of types and terms for every context in the category of contexts. Secondly, we require the contexts to be extendable by types, from Γ to $\Gamma.A$; and the substitutions to be extendable by terms, from σ to $\langle \sigma, a \rangle$ by a term a. This property is usually stated in a "universal property"-style definition. CwF and context extension are defined together below:

Definition 2.3.2 (Categories with families):

Given a category $\mathscr{C}$ with terminal object $\diamond \in \mathscr{C}$, the category of contexts and context morphisms, *Category with Families (CwF)* is a functor

$$F = (\mathrm{Ty}, \mathrm{Tm}) : \mathscr{C}^{\mathrm{op}} \to \mathrm{Fam}$$

that sends every $\Gamma \in \mathscr{C}$ to a pair $\left(\mathrm{Ty}(\Gamma), \mathrm{Tm}(\Gamma, A)_{A \in \mathrm{Ty}(\Gamma)}\right) \in \mathrm{Fam}$ as the image of the functor, to be interpreted as the set of types in Γ and the set of terms in Γ indexed by their types.

Furthermore, the functor $F = (\mathrm{Ty}, \mathrm{Tm})$ satisfies the following axiom:

For every $\Gamma \in \mathscr{C}$ and $A \in \mathrm{Ty}(\Gamma)$, there exists
- an object $\Gamma.A \in \mathscr{C}$,
- a morphism $p : \Gamma.A \to \Gamma$ in $\mathscr{C}$,
- a term $q : \mathrm{Tm}(\Gamma.A, A[p])$

(Where the square bracket is the functorial action by context morphisms on terms or types. More concretely,

$$\begin{aligned} \_[p] &:\equiv \mathrm{Ty}(p) \ : \mathrm{Ty}(\Gamma) \to \mathrm{Ty}(\Gamma.A), \\ \_[p] &:\equiv \mathrm{Tm}(p) : \mathrm{Tm}(\Gamma, A) \to \mathrm{Tm}(\Gamma.A, A[p]), \end{aligned}$$

)

such that for any
- context $\Theta \in \mathscr{C}$,
- morphism $\sigma : \Theta \to \Gamma$,
- term $a : \mathrm{Tm}(\Theta, A[\sigma])$,

there is a unique morphism $\langle \sigma, a \rangle : \Theta \to \Gamma.A$ satisfying the following equalities:

$$\begin{aligned} p \circ \langle \sigma, a \rangle &= \sigma \\ q[\langle \sigma, a \rangle] &= a. \end{aligned}$$

The uniqueness requirement in the final axiom can be satisfied by requiring additionally the two equalities hold for all contexts Γ,

$$\begin{aligned} \langle p, q \rangle \quad &= \mathrm{id} \\ \langle \sigma, a \rangle \circ \delta &= \langle \sigma \circ \delta, a[\delta] \rangle. \end{aligned}$$

which we use in the Agda formalization.

To have some useful types and interesting operations, we require the CwF to be closed with at least the Π-type former for a dependent function space:

Definition 2.3.3 (Π-types): A CwF supports Π-types if, for any $\Gamma \in \mathscr{C}$, given $A \in \mathrm{Ty}(\Gamma)$ and $B \in \mathrm{Ty}(\Gamma.A)$, we have

- a type $\Pi(A, B) \in \mathrm{Ty}(\Gamma)$
- a constructor $\mathrm{lam} : \mathrm{Tm}(\Gamma.A, B) \to \mathrm{Tm}(\Gamma, \Pi(A, B))$,
- an eliminator $\mathrm{app} : \mathrm{Tm}(\Gamma, \Pi(A, B)) \to \mathrm{Tm}(\Gamma.A, B)$,

satisfying the following properties:

- for all $b \in \mathrm{Tm}(\Gamma.A, B)$, $\quad \mathrm{app}(\mathrm{lam}(b)) = b$
- for all $f \in \mathrm{Tm}(\Gamma, \Pi(A, B)), \mathrm{lam}(\mathrm{app}(f)) = f$
- for all $\sigma : \Gamma \to \Delta, (\Pi(A, B)[\sigma]) = \Pi\Big(A[\sigma], B\Big[\sigma^{\uparrow}\Big]\Big)$
- for all $\sigma : \Gamma \to \Delta, \mathrm{lam}(M)\Big[\sigma^{\uparrow}\Big] = \mathrm{lam}\Big(M\Big[\sigma^{\uparrow}\Big]\Big)$

where $\_^{\uparrow} := \langle \_ \circ p, q \rangle$ represents weakening (extension of the context) by a suitable type for the equations to hold.

The first two properties listed above correspond to β-reduction and η-reduction in dependent types respectively, while the last two properties state that the formers Π, lam are compatible with the context morphisms/substitutions.

Definition 2.3.4: The collection of all models of CwFs form a category. The initial object in this category is called the **syntactic model**, also known as **dependent types**, denoted by I.

In other words, the collection of all models gives all possible semantics of CwFs, and any model of CwF provides an interpretation of the syntactic model. To define a model of CwF, it suffices to give the image of the syntactic model in it.

3 Transport Hell

$$
\begin{array}{ll}
\mathrm{Con} & : \mathrm{Set} \\
\mathrm{Sub} & : \mathrm{Con} \to \mathrm{Con} \to \mathrm{Set} \\
\mathrm{Ty} & : \mathrm{Con} \to \mathrm{Set} \\
\mathrm{Tm} & : (\Gamma : \mathrm{Con}) \to \mathrm{Ty}\ \Gamma \to \mathrm{Set} \\
\mathrm{id} & : \mathrm{Sub}\ \Gamma\ \Gamma \\
\_ \circ \_ & : \mathrm{Sub}\ \Delta\ \Gamma \to \mathrm{Sub}\ \Gamma\ \Theta \to \mathrm{Sub}\ \Delta\ \Theta \\
\mathrm{idl} & : \mathrm{id} \circ \sigma = \sigma \\
\mathrm{idr} & : \sigma \circ \mathrm{id} = \sigma \\
\mathrm{assoc} & : (\delta \circ \gamma) \circ \theta = \delta \circ (\gamma \circ \theta) \\
\diamond & : \mathrm{Con} \\
\varepsilon & : \mathrm{Sub}\ \Gamma\ \diamond \\
\diamond\eta & : (\sigma : \mathrm{Sub}\ \Gamma\ \diamond) = \varepsilon
\end{array}
$$

$$
\begin{array}{ll}
\_[\_] & : \mathrm{Ty}\ \Gamma \to \mathrm{Sub}\ \Delta\ \Gamma \to \mathrm{Ty}\ \Delta \\
[\mathrm{id}] & : A[\mathrm{id}] = A \\
[\circ] & : A[\sigma \circ \delta] = A[\sigma][\delta] \\
\_[\_] & : \mathrm{Tm}\ \Gamma\ A \to (\sigma : \mathrm{Sub}\ \Delta\ \Gamma) \to \mathrm{Tm}\ \Delta\ A[\sigma] \\
[\mathrm{id}] & : a[\mathrm{id}] = a \\
[\circ] & : a[\sigma \circ \delta] = a[\sigma][\delta] \\
(\_ \triangleright \_) & : (\Gamma : \mathrm{Con}) \to \mathrm{Ty}\ \Gamma \to \mathrm{Con} \\
(\_, \_) & : \sigma : \mathrm{Sub}\ \Gamma\ \Delta \to \mathrm{Tm}\ \Gamma\ A \to \mathrm{Sub}\ \Gamma\ (\Delta \triangleright A) \\
p & : \mathrm{Sub}\ (\Gamma \triangleright A)\ \Gamma \\
q & : \mathrm{Tm}\ (\Gamma \triangleright A)\ A[p] \\
\triangleright\beta_1 & : p \circ (\sigma, t) = \sigma \\
\triangleright\beta_2 & : q \circ (\sigma, t) = t \\
,\circ & : (\sigma, t) \circ \gamma = (\sigma \circ \gamma, t[\gamma]) \\
,\eta & : (p, q) = \mathrm{id}
\end{array}
$$

Figure 2: definition of CwFs in pseudo-Agda syntax.

Figure 2 shows what the Agda formalization is supposed to be. However, these simple-looking definitions quickly turn opaque. For example, the context extension rules are written as follows in Agda:

```
56      -- context extension (by a type)
57      _▹_ : (Γ : Con) → Ty Γ → Con -- Γ.σ
58      p   : ∀{Γ A} → Sub (Γ ▹ A) Γ
59      v   : ∀{Γ A} → Tm (Γ ▹ A) (A [ p ])
60      -- context morphism extension (by a term)
61      _,_ : ∀{Γ Δ A} → (σ : Sub Δ Γ) → Tm Δ (A [ σ ]) → Sub Δ (Γ ▹ A)
62      -- satisfying axioms
63      consl   : ∀{Γ Δ} → (f : Sub Γ Δ) → (A : Ty Δ) → (a : Tm Γ (A [ f ]))
64                → p ∘ (f , a) ≡ f
65      consr   : ∀{Γ Δ} → (f : Sub Γ Δ) → (A : Ty Δ) → (a : Tm Γ (A [ f ]))
66                → transp (Tm Γ) (trans (sym ([∘] A (f , a) p)) (cong (_[_] A) (consl f A a)))
    (v t[ (f , a) ]) ≡ a
67      consnat : ∀{Γ Δ θ} → (f : Sub Γ Δ) → (A : Ty Δ) → (a : Tm Γ (A [ f ]))
68                → (g : Sub θ Γ)
69                → (f , a) ∘ g ≡ (f ∘ g) , transp (Tm θ) (sym ([∘] A g f)) (a t[ g ])
70      consid  : ∀{Δ}{A : Ty Δ} → p {Δ} {A} , v ≡ id
```

Listing 1: `cwf.agda`: Context extension defined in Agda.

Where the highlighted lines illustrate the verbosity of the statement, which is visible even without understanding what the operators mean in Agda. In fact, these definitions are not just opaque, but even stating them become some brain gymnastics that the user has to do, in order to write a definition that typechecks in Agda.

3.1.1 An example with term substitutions

So what is wrong here? In fact, we are doing the work of a compiler - to do reasonings modulo equalities by ourselves. In the mathematical pseudo-Agda formulation, our brain reasons modulo these equalities without noticing because they are natural to us, but not so obvious to the Agda compiler when stated precisely. To begin, we first give a simple example to illustrate the meaning of "transport", appeared in the definition of "substitution in terms":

```
51      t[id] : ∀{Γ A} → (t : Tm Γ A) → t ≡ transp (Tm Γ) [id] (t t[ id ])
```

Listing 2: `cwf.agda`: Terms are invariant under identity substitutions.

which essentially says that a term remains constant under the action of the identity context morphism:

$$\forall t : \mathrm{Tm}\ \Gamma\ A, t[\mathrm{id}] = t$$

In our mind, this works perfectly, but we forgot that formally, the left hand side term $t[\mathrm{id}]$ has a type of

$$A[\mathrm{id}],$$

and the right hand side term t has a type of

$$A.$$

These are inherently different, but we have arbitrarily (yet logically) defined them to be equal:

$$\forall A : \mathrm{Ty}\ \Gamma, A[\mathrm{id}] = A.$$

Even though we already had a rule previously in Agda, stating that they are equal,

```
46      [id] : ∀{Γ} → {A : Ty Γ} → A [ id ] ≡ A
```

Listing 3: `cwf.agda`: Types are invariant under identity substitutions.

Agda cannot compute the types modulo the equality defined above, showing the error:

```
(A [ id ]) ≠ A of type Ty Γ
when checking that the expression t t[ id ] has type Tm Γ A
```

We have to manually "transport" the term $t[\mathrm{id}]$ along the equality $A = A[\mathrm{id}]$, yielding a term of type A instead of $A[\mathrm{id}]$, resulting in the semantically equal, but more verbose formulation in Listing 2. Similary, the fact that term substitution commutes with composition of substitutions:

$$[\circ] : a[\sigma \circ \delta] = a[\sigma][\delta]$$

has the same issue that each side has a different type: $A[\sigma \circ \delta]$ and $A[\sigma][\delta]$ respectively; which are again equal, due to our definition. In Agda, the equality on *term* substitution has to be transported over the equality on *type* substitution, as highlighted in Listing 4:

```
47      [∘]  : ∀{Γ1 Γ2 Γ3} → (A : Ty Γ3) → (f : Sub Γ1 Γ2) → (g : Sub Γ2 Γ3) →
48             A [ (g ∘ f) ] ≡ A [ g ] [ f ]
```

```
52      t[∘]  : ∀{Γ1 Γ2 Γ3 A} → (a : Tm Γ3 A)
53              → (f : Sub Γ1 Γ2) → (g : Sub Γ2 Γ3)
54              → a t[ g ∘ f ] ≡ transp (Tm Γ1) (sym ([∘] A f g)) ((a t[ g ]) t[ f ])
```

Listing 4: `cwf.agda`: Substitution commutes with composition.

3.1.2 An example by the weakening of a substitution

```
80    -- weakening of σ by A
81    _^ : ∀{Γ Δ} → (σ : Sub Δ Γ) → {A : Ty Γ} → Sub (Δ ▹ (A [ σ ])) (Γ ▹ A)
82    _^ {Γ} {Δ} σ {A} = (σ ∘ p) , transp (Tm (Δ ▹ (A [ σ ]))) (sym ([∘] A p σ)) (v {Δ} {A [ σ ]})
```

Now, this operation of "weakening" a context morphism should behave well with the special case of identity context morphisms. It should be the identity context morphism of the weakened context!

We can thus prove the property, but the proof is unnecessarily difficult to proof, since Agda cannot compute the result modulo the equalities that we have assumed. In turn, one has to first define lemmas such as

- `coe2l` that shows transport commutes with symmetry of equality,
- `transptransp` that shows transport commutes with transitivity of equality,
- `transp,` (note the comma) that shows transport commutes with the pairing operation.

before having a (still long) proof for the simple property

```
110   id^ : ∀{Γ A} →
111     id {Γ} ^ ≡ transp (λ t → Sub t (Γ ▹ A)) (cong (_▹_ Γ) (sym [id])) id {Γ ▹ A}
112   id^ {Γ} {A} =
113     cong
114       (λ x → π₁ x , π₂ x)
115       (idl p ,= transptransp
116         (Tm (Γ ▹ (A [ id ])))
117         (sym ([∘] A p id)) {cong (A [_]) (idl p)}) ▪
118     transp, [id] ▪
119     cong (transp (λ z → Sub (Γ ▹ (A [ id ])) (Γ ▹ z)) [id]) consid ▪
120     coe2l
121       (sym (cong (λ t → Sub t (Γ ▹ A)) (cong (_▹_ Γ) (sym [id]))))
122       (coecoe
123         (cong (λ z → Sub (Γ ▹ (A [ id ])) (Γ ▹ z)) [id])
124         (sym (cong (λ t → Sub t (Γ ▹ A)) (cong (_▹_ Γ) (sym [id])))) ▪
125     trid [id])
```

Listing 5: `cwf.agda`: weakening of substitution.

These lemmas on transport not only obscure the real content of the definition or proof but are also unnecessarily effort-consuming as they do not carry substantial meaning beyond what is implied by the user-defined equalities. Therefore it is nicknamed as such by the community for its presence.

3.2 Possible solutions

3.2.1 Decrease indexing/dependency

We have seen that transport hell occurs when reasoning modulo equalities cannot be resolved automatically. The reason behind this is the presence of "dependency", a feature that ironically gives dependent types its logical power. The dependency occurs in the following settings:

1. A type depends on its context
2. A term depends on both its context *and* its type

This deep dependency often stands in the way of computation, especially user-defined equality rules. One way to resolve this is to decouple terms from its type. Instead of having

$$\mathrm{Tm} : (\Gamma : \mathrm{Con}) \to \mathrm{Ty}\ \Gamma \to \mathrm{Set},$$

we can separate the dependency on types into an additional projection ty,

$$\begin{aligned} \mathrm{Tm} &: \mathrm{Con} \to \mathrm{Set} \\ \mathrm{ty} &: \mathrm{Tm}\ \Gamma \to \mathrm{Ty}\ \Gamma \end{aligned}$$

Where ty t will give the type of t. Then any property or definition on a term must have an additional equation on its type, through the ty function. This is also the idea behind Awodey's Natural Model, an intrinsic model for Homotopy Type Theory. In practice, this approach is widely used by the community in making formal proofs, in anticipation of transport hell.

In this approach, one avoids using transports entirely, so whenever a transport is needed, we state the equality to transport over in an extra argument. This makes explicit the reasoning we make during transport. More detail is given in Section 4.

3.2.2 Generalized Rewriting and Observational Type Theory

A possible option that can alleviate this burden is the Rewriting feature in Agda, which allows rewriting of terms according to user-defined (and proven) equivalence relations. However, adding rewriting relations amounts to a stronger version of type theory than Agda's, and we would like to avoid that

as much as possible. An investigation of Rewriting Type Theory with formal verification of metatheoretical properties and an implementation is documented in [4].

Observational Type Theory [5], on the other hand, is a type system with built-in reduction rules targeting transport, and could reduce some transportation, such as the transport of a pair can be reduced into the transport of its components, and could be a suitable object of study.

4 Less-indexed CwFs

$$
\begin{array}{ll}
\mathrm{Con} & : \mathrm{Set} \\
\mathrm{Sub} & : \mathrm{Con} \to \mathrm{Con} \to \mathrm{Set} \\
\mathrm{Ty} & : \mathrm{Con} \to \mathrm{Set} \\
\mathrm{Tm} & : \mathrm{Con} \to \mathrm{Set} \\
\mathrm{ty} & : \mathrm{Tm}\ \Gamma \to \mathrm{Ty}\ \Gamma \\
\mathrm{id} & : \mathrm{Sub}\ \Gamma\ \Gamma \\
\_ \circ \_ & : \mathrm{Sub}\ \Delta\ \Gamma \to \mathrm{Sub}\ \Gamma\ \Theta \to \mathrm{Sub}\ \Delta\ \Theta \\
\mathrm{idl} & : \mathrm{id} \circ \sigma = \sigma \\
\mathrm{idr} & : \sigma \circ \mathrm{id} = \sigma \\
\mathrm{assoc} & : (\delta \circ \gamma) \circ \theta = \delta \circ (\gamma \circ \theta) \\
\diamond & : \mathrm{Con} \\
\varepsilon & : \mathrm{Sub}\ \Gamma\ \diamond \\
\diamond\eta & : (\sigma : \mathrm{Sub}\ \Gamma\ \diamond) = \varepsilon
\end{array}
\qquad
\begin{array}{ll}
\_[\_] & : \mathrm{Ty}\ \Gamma \to \mathrm{Sub}\ \Delta\ \Gamma \to \mathrm{Ty}\ \Delta \\
[\mathrm{id}] & : A[\mathrm{id}] = A \\
[\circ] & : A[\sigma \circ \delta] = A[\sigma][\delta] \\
\_[\_] & : \mathrm{Tm}\ \Gamma \to \mathrm{Sub}\ \Delta\ \Gamma \to \mathrm{Tm}\ \Delta \\
\mathrm{ty}[] & : \mathrm{ty}\ (a[\gamma]) = (\mathrm{ty}\ a)[\gamma] \\
[\mathrm{id}] & : a[\mathrm{id}] = a \\
\mathrm{ty}[\mathrm{id}] & : \mathrm{ty}\ (a[\mathrm{id}]) = \mathrm{ty}\ a \\
[\circ] & : a[\sigma \circ \delta] = a[\sigma][\delta] \\
\mathrm{ty}[\circ] & : \mathrm{ty}\ (a[\sigma \circ \delta]) = \mathrm{ty}\ (a[\sigma][\delta]) \\
(\_ \triangleright \_) & : (\Gamma : \mathrm{Con}) \to \mathrm{Ty}\ \Gamma \to \mathrm{Con} \\
(\_, \_) & : \mathrm{Sub}\ \Gamma\ \Delta \to \mathrm{Tm}\ \Gamma \to \mathrm{Sub}\ \Gamma\ (\Delta \triangleright A) \\
p & : \mathrm{Sub}\ (\Gamma \triangleright A)\ \Gamma \\
q & : \mathrm{Tm}\ (\Gamma, A) \\
\triangleright\beta_1 & : p \circ (\sigma, t) = \sigma \\
\triangleright\beta_2 & : q \circ (\sigma, t) = t \\
\mathrm{ty} \triangleright \beta_2 & : \mathrm{ty}\ (q \circ (\sigma, t)) = \mathrm{ty}\ t \\
,\circ & : (\sigma, t) \circ \gamma = (\sigma \circ \gamma, t[\gamma]) \\
,\eta & : (p, q) = \mathrm{id}
\end{array}
$$

Figure 3: Definition of less-indexed CwFs in pseudo-Agda syntax, difference in red.

For the second part of the project, I formalized a version of CwFs which terms are not indexed by types. In this approach, I avoid the using of transport at all costs. The result has a clear syntax in Agda, which the reader can verify, for example that term substitution and its related properties have no more transport in lines 57 and 60 of Listing 6:

```
30      Con : Set k
31      Sub : Con → Con → Set l
32
33      ◇ : Con
34      ε : ∀{Γ} → Sub Γ ◇
35      ◇η : ∀{Γ}{σ : Sub Γ ◇} → σ ≡ ε
36
37      id : ∀{Γ} → Sub Γ Γ
38      _∘_ : ∀{Γ1 Γ2 Γ3} → Sub Γ2 Γ3 → Sub Γ1 Γ2 → Sub Γ1 Γ3
39
40      idl : ∀{Γ Δ}{σ : Sub Γ Δ} → id ∘ σ ≡ σ
41      idr : ∀{Γ Δ}{σ : Sub Γ Δ} → σ ∘ id ≡ σ
42      assoc : ∀{Γ1 Γ2 Γ3 Γ4}{σ12 : Sub Γ1 Γ2}{σ23 : Sub Γ2 Γ3}{σ34 : Sub Γ3 Γ4}
43        → σ34 ∘ σ23 ∘ σ12 ≡ σ34 ∘ (σ23 ∘ σ12)
44
45      -- Ty, Tm, ty, and action by substitution
46      Ty   : Con → Set k
47      _[_] : ∀{Γ Δ} → Ty Γ → Sub Δ Γ → Ty Δ
48      [id] : ∀{Γ}{A : Ty Γ} → A ≡ A [ id ]
49      [∘]  : ∀{Γ1 Γ2 Γ3}{A : Ty Γ1}{σ21 : Sub Γ2 Γ1}{σ32 : Sub Γ3 Γ2}
50             → A [ σ21 ] [ σ32 ] ≡ A [ σ21 ∘ σ32 ]
51
52      Tm    : Con → Set l
53      ty    : {Γ : Con} → Tm Γ → Ty Γ
54      _[_]t : ∀{Γ Δ} → Tm Γ → (γ : Sub Δ Γ) → Tm Δ
55      ty[]t : ∀{Γ Δ} → (γ : Sub Δ Γ) → (a : Tm Γ) → ty (a [ γ ]t) ≡ (ty a) [ γ ]
56
57      [id]t   : ∀{Γ}{a : Tm Γ} → a [ id ]t ≡ a
58      ty[id]t : ∀{Γ}{a : Tm Γ} → ty (a [ id ]t) ≡ (ty a) [ id ]
59
60      [∘]t    : ∀{Γ1 Γ2 Γ3}{σ21 : Sub Γ2 Γ1}{σ32 : Sub Γ3 Γ2}{a : Tm Γ1}
61                → a [ σ21 ∘ σ32 ]t ≡ a [ σ21 ]t [ σ32 ]t
62      ty[∘]t  : ∀{Γ1 Γ2 Γ3}{σ21 : Sub Γ2 Γ1}{σ32 : Sub Γ3 Γ2}{a : Tm Γ1}
63                → ty (a [ σ21 ∘ σ32 ]t) ≡ ty a [ σ21 ∘ σ32 ]
64
65      -- context extension (by a type)
66      _▹_     : (Γ : Con) → Ty Γ → Con -- Γ.A
67      p       : ∀{Γ A} → Sub (Γ ▹ A) Γ
68      v       : ∀{Γ A} → Tm (Γ ▹ A)
69      tyv     : ∀{Γ A} → ty (v {Γ} {A}) ≡ A [ p ]
70      -- context morphism extension (by a term)
71      _,_[_]s : ∀{Γ Δ}{A : Ty Γ} → (γ : Sub Δ Γ) → (a : Tm Δ) → ty a ≡ A [ γ ] → Sub Δ (Γ
    ▹ A)
72      consl   : ∀{Γ Δ}{A : Ty Γ} → {γ : Sub Δ Γ} → {a : Tm Δ} → {e : ty a ≡ A [ γ ]}
73                → p ∘ (γ , a [ e ]s) ≡ γ
74      consr   : ∀{Γ Δ}{A : Ty Γ} → {γ : Sub Δ Γ} → {a : Tm Δ} → {e : ty a ≡ A [ γ ]}
75                → v [ γ , a [ e ]s ]t ≡ a
76      tyconsr : ∀{Γ Δ}{A : Ty Γ} → {γ : Sub Δ Γ} → {a : Tm Δ} → {e : ty a ≡ A [ γ ]}
77                → ty (v [ γ , a [ e ]s ]t) ≡ (ty v) [ γ , a [ e ]s ]
78       consnat : ∀{Γ1 Γ2 Γ3}{A : Ty Γ1} → {γ21 : Sub Γ2 Γ1} → {a : Tm Γ2} → {e : ty a ≡
    A [ γ21 ]}
79        → {γ32 : Sub Γ3 Γ2}
80        → (γ21 , a [ e ]s) ∘ γ32 ≡ (γ21 ∘ γ32) , (a [ γ32 ]t)
81          [ ty[]t γ32 a ▪ cong (λ t → t [ γ32 ]) e ▪ [∘] ]s
82      consid : ∀{Γ Δ}{A : Ty Γ} → (γ : Sub Δ Γ) → (a : Tm Δ) → p {Γ} {A} , v [ tyv ]s ≡ id
```

Listing 6: `cwfnat.agda`: definition of less-indexed CwFs.

When transport is needed, the equality to transport over is made explicit via an extra argument. As an example:

```
88    -- extension of a substitution by a type (useful!)
89    _^[_] : ∀{Γ Δ A'}{A : Ty Γ}(σ : Sub Δ Γ) → A' ≡ A [ σ ] → (Sub (Δ ▹ A') (Γ ▹ A))
90    _^[_] {Γ} {Δ} {A'} {A} σ e = σ ∘ p , v [ tyv ▪ (cong (λ t → t [ p ]) e ▪ [∘]) ]s
91
92    _^ : ∀{Γ Δ}{A : Ty Γ}(σ : Sub Δ Γ) → (Sub (Δ ▹ (A [ σ ])) (Γ ▹ A))
93    _^ {Γ} {Δ} {A} σ = σ ^[ refl ]
```

Listing 7: `cwf.agda`: weakening of a substituition by a type.

The definition in line 89 makes the requirement of $A = A[\sigma]$ explicit by requiring a type-equality proof as an argument. Line 92 then defines a shorthand whenever no transport is needed. By comparison, this is much more readable and is dozens of line simpler than the same operation in Listing 5.

4.1 Comparison with CwFs

	CwF	**Less indexed**
Types	$\mathrm{Ty}\ \Gamma$	$\mathrm{Ty}\ \Gamma$
Terms	$\mathrm{Tm}\ \Gamma\ A$	$\mathrm{Tm}\ \Gamma, \quad \mathrm{ty} : \mathrm{Tm}\ \Gamma \to \mathrm{Ty}\ \Gamma$
Conversion	$\mathrm{Tm}\ \Gamma \coloneqq \Sigma(A : \mathrm{Ty}\ \Gamma).\ \mathrm{Tm}\ \Gamma\ A$	$\mathrm{Tm}\ \Gamma\ A \coloneqq \Sigma(a : \mathrm{Tm}\ \Gamma).(\mathrm{ty}\ a = A)$

Since the notions of terms in both representations are equivalent, we can design a mapping from CwF to the less-indexed version, and in the reverse direction. This is shown in the table above. Note that

$$\mathrm{Tm}\ \Gamma \quad \mapsto \quad A \to \Sigma(a : \mathrm{Tm}\ \Gamma).(\mathrm{ty}\ a = A) \quad \mapsto \quad \Sigma(A : \mathrm{Ty}\ \Gamma).\Sigma(a : \mathrm{Tm}\ \Gamma).(\mathrm{ty}\ a = A)$$

so the two categories of terms are equivalent, but not isomorphic.

CwF	**Less indexed**
transport hell	equational reasoning
close to syntax: no extra argument	extra arguments compared to the presyntax
more lines of proofs due to opaque terms	less lines of proofs
Generalized Algebraic Theory (indexed sorts)	Essentially Algebraic Theory (partial functions)

With the less indexed version, since we can use the powerful equational reasoning mechanism of Agda (which is also true for all proof assistants), it results in fewer lines of proofs, but at the cost of having more equality arguments compared to the syntax of dependent types (eg. Listing 7). Under the categorization of Universal Algebra, since the less-indexed version has partial functions, namely due to the additional equality arguments in the definitions, it applies to only selected objects of its domain, this is an Essentiatlly Algebraic Theory instead of a Generalized Algebraic Theory.

5 Future Work and Conclusion

In this report, we have defined Categories with Families with Pi-types and given an Agda formalization. We explained "transport hell" in the process of this Agda formalization with two examples in the code, and investigated potential solutions to overcome them, such as reducing dependency à la *Natural Model*, using Agda's rewriting feature, as well as Observational Type Theory. Finally, we gave an Agda formalization of the less-indexed CwFs, and evaluated its advantages and disadvantages against

normal CwFs. This object's original goal of proving Canonicity was replaced by a more complete formalization of a different flavor of CwFs, which to me is equally rewarding.

Future possible work for this project includes proving formally the equivalence between the two formulations of CwFs, as well as proving the metatheoretical properties of dependent types on both versions (such as Canonicity and Normalisation) and compare the difficultes of the two versions of CwFs. It would also be useful to explore other methods of combating transport hell surveyed in Section 3.2.2 and cross-compare the attempts.

Last but not least, the author wishes to thank Professor Ambrus Kaposi (akaposi@inf.elte.hu) of Eötvös Loránd University for his guidance on this project.

Bibliography

[1] M. Lennon-Bertrand, "Towards a certified proof assistant kernel – What it takes and what we have," 2023. [Online]. Available: https://www.meven.ac/documents/Certified_kernels.pdf

[2] A. Kaposi, S. Huber, and C. Sattler, "Gluing for Type Theory," in *4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019)*, H. Geuvers, Ed., in Leibniz International Proceedings in Informatics (LIPIcs), vol. 131. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019, pp. 1–19. doi: 10.4230/LIPIcs.FSCD.2019.25.

[3] S. Castellan, P. Clairambault, and P. Dybjer, "Categories with families: Unityped, simply typed, and dependently typed," *Joachim Lambek: The Interplay of Mathematics, Logic, and Linguistics*, pp. 135–180, 2021.

[4] J. Cockx, N. Tabareau, and T. Winterhalter, "The taming of the rew: a type theory with computational assumptions," *Proc. ACM Program. Lang.*, vol. 5, no. POPL, Jan. 2021, doi: 10.1145/3434341.

[5] L. Pujet and N. Tabareau, "Observational equality: now for good," *Proceedings of the ACM on Programming Languages*, vol. 6, no. POPL, pp. 1–27, 2022.

[6] M. Hofmann and M. Hofmann, "Syntax and semantics of dependent types," *Extensional Constructs in Intensional Type Theory*, pp. 13–54, 1997.

[7] T. Coquand, "Canonicity and normalization for dependent type theory," *Theoretical Computer Science*, vol. 777, pp. 184–191, 2019.

[8] N. Tabareau, É. Tanter, and M. Sozeau, "Equivalences for Free: Univalent Parametricity for Effective Transport," *Proc. ACM Program. Lang.*, vol. 2, no. ICFP, Jul. 2018, doi: 10.1145/3236787.