Progression Review Report

Formalizing Higher-Order Containers

Zhili Tian

Supervised by Thorsten Altenkirch

& Ulrik Buchholtz

Functional Programming Lab

School of Computer Science

University of Nottingham

July 21, 2025

Abstract

Giving a short overview of the work in your project.[1]

Contents

Chapter 1

Introduction

1.1 Background and Motivation

The concept of types is one of the most important features in most modern programming languages. It is introduced to classify variables and functions, enabling more meaningful and readable codes as well as ensuring type correctness. Types such as *boolean*, *natural number*, *list*, *binary tree*, etc. are massively used in everyday programming.

TODO

1.2 Aims and Objectives

TODO

- bla bla
- bla bla

1.3 Progress to Date

TODO : progress and achievements during this stage, training courses, seminars and presentations.

1.4 Overview of the Report

- Chapter 2 - Literature Review: reviews the related literature, and further motivates our project
- Chapter 3 - Conducted Research: covers background knowledge, topics studied and questions. We introduce type theory, category theory, containers, etc.
- Chapter 4 - Future Work Plan: TODO

Chapter 2

Prerequisites and Settings

We assume the reader has basic knowledge in logics, functional programming and category theory. We introduce type theory and set it as our framework. In particular, we use Agda, an implemented programming language, for our study and in this report.

Assuming `a : A` and `b : B`, the following are standard Agda syntax for common algebraic data types:

- `⊥` - Empty Type, with no term
- `⊤` - Unit Type, with `tt : ⊤`
- `×` - Product Type, with `a , b : A × B`
- `⊎` - Coproduct (Sum) Type, with `inj₁ a : A ⊎ B`, and `inj₂ b : A ⊎ B`

2.1 Type Theory

Type theory is a formal language of mathematical logics, designed to serve as a foundation for mathematics and programming languages. Various flavors of type theories exist, differing in their treatment of equality, interpretation of types, computational univalence, etc. We introduce basic concepts in type theory and outline the development.

Universes

In type theory, everything has a type, even types have a type. We call it the universe and denote it as `Type`. Such that:

```
⊥ , ⊤ , ℕ , ... : Type
```

But what is the type of `Type`? It turns out if we naively postulate `Type : Type`, then it leads to Girard's paradox which causes inconsistency in the system. The solution is to build hierarchy of the universes:

```
Type : Type₁ : Type₂ : ...
```

where each universe is bigger than the previous ones. In addition, if a type is in a universe, it can be lifted into a bigger universe.

Type Families

The concepts of type families can be viewed as generalization of ordinary functions, where the codomain is allowed to be `Type`. For example a predicate that tells the evenness can be defined as a type family over `ℕ`:

```
isEven : ℕ → Type
isEven zero = ⊤
isEven (suc zero) = ⊥
isEven (suc (suc n)) = isEven n
```

MLTT

Per Martin-Löf introduced MLTT as a constructive foundations for mathematics. It is also known as the intuitionistic type theory or dependent type theory. Form a programming point of view, it adds type level control of data types. For example we can define vector:

```
data Vec (A : Type) : ℕ → Type where
  [] : Vec A 0
  _∷_ : {n : ℕ} → A → Vec A n → Vec A (suc n)
```

According to this definition, `Vec ℕ 3` should have exactly 3 numbers:

```
vec : Vec ℕ 3
vec = 1 ∷ 2 ∷ 3 ∷ []
```

Type families also generalize function type `→` into dependent function type `Π` and pair type `×` into dependent pair type `Σ`.

```
Π : (A : Type) (B : A → Type) → Type
Π A B = (a : A) → B a

record Σ (A : Type) (B : A → Type) : Type where
  constructor _,_
  field
    proj₁ : A
    proj₂ : B proj₁
```

We can now extend the Curry-Howard correspondence by interpreting `Π` as universal quantification and `Σ` as existential quantification. For example, we can show there is some even number through `Σ`:

```
∃Even : Σ ℕ isEven
∃Even = 2 , tt
```

Equalities

It is also important to distinguish different notions of equality. The definitional equality is a very strong notion of equality which two terms are reduced to the same normal forms. However, most theorems and results of practical interests do not hold definitionally. Their proofs normally involves doing case analysis, building extra lemmas, etc. We refer this weaker notion of equality the propositional equality.

In Agda, (definitional) equality is represented by identity type, which is a binary type family over any type `A`, and it is witnessed by `refl`.

```
data _≡_ {A : Type} (x : A) : A → Type where
  refl : x ≡ x
```

Therefore, if a proposition is hold directly by `refl`, then It is definitional equality. In contrast, if an explicit proof is required, then it is propositional equality.

```
thm0 : 1 + 1 ≡ 2
thm0 = refl

thm1 : (n : ℕ) → 1 + n ≡ n + 1
thm1 zero = refl
thm1 (suc n) = cong suc (thm1 n)
  where
```

```
cong : {A B : Type} (f : A → B) {x y : A} → x ≡ y → f x ≡ f y
cong f refl = refl
```

Function Extensionality

Here arises another question - what is the good notion of equality of functions? Function extensionality suggests that two functions should considered as equal if they yield the same output for every input. However, in intensional type theory, for example Agda, where equality is defined as definitional equality, the `funExt` is not provable within the system and must instead be postulated.

```
postulate
  funExt : {A : Type} {B : A → Type} {f g : (x : A) → B x} →
           ((x : A) → f x ≡ g x) → f ≡ g
```

On the other hand, an extensional type theory suggest to treat two notions of equality as the same one. That is, it forces the proposition equality back into the definitional one. However, this approach has significant drawbacks: type checking becomes undecidable, and computation loses canonicity.

UIP

Given two terms `x y : A` and two proofs of equality `p q : x ≡ y`, should we have `p ≡ q`?

The answer is, in MLTT, the identity type can have multiple district proofs, so `p ≢ q` may be possible. However, one could add the uniqueness of identity proofs, aka K, as an axiom to force `p ≡ q` always holds. From another point of view, types are treated much like sets, as in set theory two terms being equal is just a proposition and nobody talks about the equality of propositions.

HoTT

The homotopy type theory extends MLTT further and provides an alternative interpretation of types. By adopting the concepts from homotopy theory, it types are viewed as spaces and equalities are viewed as paths between spaces.

We will omit a discussion of the univalence axiom, as it falls outside the scope of this report.

H-Levels

UIP does not hold in HoTT and it is not allowed to be postulated. Instead, HoTT introduces the concepts of h(omotopy)-levels in order to interpret and fix this nested equalities issue. The idea is to classify and assign each type (or space) to a h-level.

At the base, a (-2)-type, or the contractible, is one in which has exactly one term (or center point), up to equality (or homotopy):

```
record isContr (A : Type) : Type where
  field
    center : A
    paths : (x : A) → center ≡ x
```

A (-1)-type, or the proposition, is one that all terms are equal:

```
isProp : Type → Type
isProp A = (x y : A) → x ≡ y
```

A 0-type, or set, is one in which all equalities are equal.

```
isSet : Type → Type
isSet A = (x y : A) (p q : x ≡ y) → p ≡ q
```

Indeed, one could inductively define `isHLevel` over `ℕ` to get general form of higher homotopy type.

```
isOfHLevel : ℕ → Type → Type
isOfHLevel zero A = isContr A
isOfHLevel (suc n) A = (x y : A) → isOfHLevel n (x ≡ y)
```

CTT

One of the major drawbacks of HoTT is its lack of computability. Cubical type theory is then developed as a computable version of HoTT. In particular, equality (or path) of type `A` is redefined as a function `I → A`, where `I` is a primitive type called interval. There are two terms in an interval - `i0 , i1 : I`, being used to parametrize two endpoints of a path. In Cubical Agda, path type is defined as follow.

```
Path : (A : Type) → A → A → Type
Path A a₀ a₁ = I → A
```

A path `p` comes also with two side constraints: `p i0 = a₀` and `p i1 = a₁` We can explicitly define path of path to capture equalities of equalities:

```
Square : (A : Type) {a₀₀ a₀₁ a₁₀ a₁₁ : A}
  → Path A a₀₀ a₀₁ → Path A a₁₀ a₁₁
  → Path A a₀₀ a₁₀ → Path A a₀₁ a₁₁
  → Type
Square A top bottom left right = I → I → A
```

Again a square `s` should satisfy the cubical side conditions which makes the function evaluation to each `a` point-wisely. Finally, one could repeat this to define cubes and general higher cubes.

2.2 Non-Cubical Settings

Terence Tao once describes three stages in mathematical learning and practice: pre-rigorous, rigorous, and post-rigorous. The idea of post-rigorous is that, when one already know how to do thins rigorously, then he can move fluently between intuition, informal reasoning, and formal rigor as needed.

In our context, we conduct theoretical research within HoTT, exploring concepts such as the interpretation of containers as endofunctors on h-level sets. This requires us to explicitly track h-level fields in Cubical Agda. We did formalizations in both MLTT and CTT and decided to continue and present our code using vanilla Agda.

We now explicitly state our working assumptions. H-level checking is hidden, therefore equalities between equalities within set-level structures are ignored. We also assume function extensionality and minimize the use of universe levels for simplicity.

Chapter 3

Conducted Research

In this section, we will first review related literatures. Then we outline the topics studied, such as categorical semantics of inductive and coinductive types, containers, W and M types, etc. Finally, we illustrate the current challenge.

3.1 Literature Review

TODO

3.2 Types and Categorical Semantics

3.2.1 Inductive Types

We now give a definition of inductive types. An inductive type `A` is specified by a finite set of its data constructors, and a function `f : A → B`, where `B` is arbitrary type.

Natural Number

We can define natural number `ℕ = {0, 1, 2, 3, ...}` as an inductive type:

```
data ℕ : Type where
  zero : ℕ
  suc : ℕ → ℕ
```

The definition of natural number `ℕ` follows exactly the Peano axiom. The first constructor says `zero` is a `ℕ`. The second constructor is a function that sends `ℕ` to `ℕ`, which in other word, if `n` is a `ℕ`, so is the `suc n`.

We also need to spell out the induction principle, aka the eliminator, of ℕ:

```
indℕ : (P : ℕ → Type)
  → P zero
  → ((n : ℕ) → P n → P (suc n))
  → (n : ℕ) → P n
indℕ P z s zero = z
indℕ P z s (suc n) = s n (indℕ P z s n)
```

In principle, we always need to call `indℕ` explicitly to define a function out of ℕ. It is a rather verbose process. Fortunately, we are able to instead use pattern-matching to define functions, which implements induction internally and guarantees correctness.

```
double : ℕ → ℕ
double = indℕ (λ _ → ℕ) zero (λ n dn → suc (suc dn))

double' : ℕ → ℕ
double' zero = zero
double' (suc n) = suc (suc n)
```

Inductive Types are Initial Algebras

The category of algebra in `Type` of a given endofunctor `F : Type → Type` is defined as:

- Objects are:
  - A carrier type `A : Type`, and
  - A function `α : F A → A`
- Morphisms of `(A , α)` and `(B , β)` are:
  - A morphism `f : A → B`, and
  - A commuting diagram: $\begin{tikzcd} F\ A \arrow[r, "F\ f"] \arrow[d, "\alpha"] & F\ B \arrow[d, "\beta"] \\ A \arrow[r, "f"] & B \end{tikzcd}$

After proving identity and composition of morphisms, we obtain a category of algebras. It turns out that in every such category, there exists an initial object which corresponds to an inductive type. We show a concrete example and discuss the general theory in later section.

Natural Number is Initial Algebra of Maybe

Natural number (together with its constructor) is the initial algebra of the `⊤⊎_` functor, which is also known as the `Maybe`. To prove this, we first combine and rewrite `zero` and `suc` into equivalent form:

```
[z,s] : ⊤ ⊎ ℕ → ℕ
[z,s] (inj₁ tt) = zero
[z,s] (inj₂ n) = suc n
```

To show the initiality, we show for arbitrary algebra, there exists a unique morphism form algebra ℕ. That is to undefine `fold`:

```
fold : (⊤ ⊎ X → X) → ℕ → X
fold α zero = α (inj₁ tt)
fold α (suc n) = α (inj₂ (fold α n))
```

Then we construct the morphism mapping part of `⊤ ⊎_` and check the following diagram commutes:

$$\begin{tikzcd} \top \uplus \mathbb{N} \arrow[r, "\top\uplus_1\ (\mathrm{fold}\ \alpha)"] \arrow[d, "{[z,s]}"'] & \top \uplus X \arrow[d, "\alpha"] \\ \mathbb{N} \arrow[r, "\mathrm{fold}\ \alpha"] & X \end{tikzcd}$$

```
⊤⊎₁ : (X → Y) → ⊤ ⊎ X → ⊤ ⊎ Y
⊤⊎₁ f (inj₁ tt) = inj₁ tt
⊤⊎₁ f (inj₂ x) = inj₂ (f x)

commute : (β : ⊤ ⊎ X → X) (x : ⊤ ⊎ ℕ)
  → fold β ([z,s] x) ≡ β (⊤⊎₁ (fold β) x)
commute β (inj₁ tt) = refl
commute β (inj₂ n) = refl
```

3.2.2 Coinductive Types

One of the greatest power of category theory is that, whenever we define something, we always get an opposite version for free. Indeed, if we inverse all the morphisms in above diagram, then we will:

- talk about the category of coalgebras;

- obtain a definition of conatural number `ℕ∞`;
- derive that `ℕ∞` is the terminal coalgebra of `⊤⊎_`

The definition of coalgebras is trivial. We define conatural number as a coinductive type:

```
record ℕ∞ : Type where
  coinductive
  field
    pred∞ : ⊤ ⊎ ℕ∞
open ℕ∞
```

We also define the inverse of `fold`, which is `unfold`:

```
unfold : (X → ⊤ ⊎ X) → X → ℕ∞
pred∞ (unfold α⁻ x) with α⁻ x
... | inj₁ tt = inj₁ tt
... | inj₂ x' = inj₂ (unfold α⁻ x')
```

That gives rise to the following commutative diagram:

$$\begin{tikzcd}
\mathtt{X} \arrow[r, "\mathtt{unfold}\ \alpha^{-}"] \arrow[d, "\alpha^{-}"'] & \mathbb{N}\infty \arrow[d, "\mathtt{pred}\infty"] \\
\top \uplus \mathtt{X} \arrow[r, "\top\uplus_1\ (\mathtt{unfold}\ \alpha^{-})"] & \top \uplus \mathbb{N}\infty
\end{tikzcd}$$

3.3 Containers

3.3.1 Strict Positivity

Containers are categorical and type-theoretical abstraction to describe strictly positive datatypes. A strictly positive type is the type where all data constructors do not include itself on the left-side of a function arrow in the domain. One typical counterexample is `Weird`:

```
{-# NO_POSITIVITY_CHECK #-}
data Weird : Set where
  foo : (Weird → ⊥) → Weird

¬weird : Weird → ⊥
¬weird (foo x) = x (foo x)
```

```
bad : ⊥
bad = ¬weird (foo ¬weird)
```

`Weird` is not strictly positive as in the domain of `foo`, `Weird` itself appears on the left-side of →. Losing strict positivity can cause issues like non-normalizable, non-terminating, inconsistency, etc. In this case, we can construct an term in the empty type using `¬weird` which is inconsistent to the definition.

3.3.2 Syntax and Semantics

A container is given by a shape `S : Type` with a position `P : S → Type`:

```
record Cont : Type₁ where
  constructor _◃_
  field
    S : Type
    P : S → Type
```

A container should give rise to a endofunctor `Type ⇒ Type`. Again we explicitly distinguish mapping objects part and mapping morphisms part of functors:

```
record ⟦_⟧₀ (SP : Cont) (X : Type) : Type where
  constructor _,_
  open Cont SP
  field
    s : S
    k : P s → X

⟦_⟧₁ : (SP : Cont) → (X → Y) → ⟦ SP ⟧₀ X → ⟦ SP ⟧₀ Y
⟦ SP ⟧₁ f (s , k) = s , f ∘ k
```

3.3.3 Categorical Structure

Containers and their morphisms form a category. The morphisms are defined as follow:

```
record ContHom (SP TQ : Cont) : Type where
  constructor _◃_
  open Cont SP
  open Cont TQ renaming (S to T; P to Q)
```

```
  field
    f : S → T
    g : (s : S) → Q (f s) → P s
```

where we can also show that it has identity and composition.

```
⟦_⟧Hom : ContHom SP TQ → (X : Set) → ⟦ SP ⟧₀ X → ⟦ TQ ⟧₀ X
⟦ f ◃ g ⟧Hom X (s , k) = f s , k ∘ g s
```

3.3.4 Semiring Structure

Containers are also known as **polynomial functors** as they forms a semiring structure. Namely, we can define zero, one, addition and multiplication for containers:

```
zero-C : Cont
zero-C = ⊥ ◃ λ ()

one-C : Cont
one-C = ⊤ ◃ λ{ tt → ⊥ }

_×C_ : Cont → Cont → Cont
(S ◃ P) ×C (T ◃ Q) = (S × T) ◃ λ (s , t) → P s ⊎ Q t

_⊎C_ : Cont → Cont → Cont
(S ◃ P) ⊎C (T ◃ Q) = (S ⊎ T) ◃ λ{ (inj₁ s) → P s ; (inj₂ t) → Q t }
```

such that semiring laws should hold. In fact, both multiplication and addition are commutative, associative and left- and right-annihilated by their units. Even better, we can define Π and Σ for containers which are the infinite generalization of × and ⊎:

```
ΠC : (I → Cont) → Cont
ΠC {I} C⃗ = ((i : I) → C⃗ i .S) ◃ λ f → Σ[ i ∈ I ] C⃗ i .P (f i)

ΣC : (I → Cont) → Cont
ΣC {I} C⃗ = (Σ[ i ∈ I ] C⃗ i .S) ◃ λ (i , s) → C⃗ i .P s
```

3.3.5 W and M

We now give the definition of general form of inductive types, which is the `W` type:

```
data W (SP : Cont) : Type where
  sup : ⟦ SP ⟧₀ (W SP) → W SP
```

From the definition, `W` is an inductive type that specified by a container. In another word, any inductive type can be specified by a strictly positive functor! We can retrieve the definition of `ℕ` through the `⊤ ⊎` functor:

```
⊤⊎Cont : Cont
⊤⊎Cont = S ◃ P
  where
  S : Type
  S = ⊤ ⊎ ⊤

  P : S → Type
  P (inj₁ tt) = ⊥
  P (inj₂ tt) = ⊤

ℕW : Type
ℕW = W ⊤⊎Cont
```

Dually, the general form of coinductive types - `M` type is just the terminal coalgebra of containers, and we can redefine `ℕ∞` using `M`:

```
record M (SP : Cont) : Type where
  coinductive
  field
    inf : ⟦ SP ⟧₀ (M SP)

ℕ∞M : Type
ℕ∞M = M ⊤⊎Cont
```

Commutative diagrams:

$$\begin{tikzcd}
\llbracket\ SP\ \rrbracket_0\ (W\ SP) \arrow[r, "\llbracket\ SP\ \rrbracket_1\ (fold\ \alpha)"] \arrow[d, "sup"] & \llbracket\ SP\ \rrbracket_0\ X \arrow[d, "\alpha"] \\
W\ SP \arrow[r, "fold\ \alpha"] & X
\end{tikzcd}$$

$$
\begin{tikzcd}
X \arrow[r, "\mathsf{unfold}\ \alpha^-"] \arrow[d, "\alpha^-"] & \mathsf{M}\ \mathsf{SP} \arrow[d, "\mathsf{inf}"] \\
[\![\ \mathsf{SP}\ ]\!]_0\ X \arrow[r, "{[\![\ \mathsf{SP}\ ]\!]_1\ (\mathsf{unfold}\ \alpha^-)}"] & [\![\ \mathsf{SP}\ ]\!]_0\ (\mathsf{M}\ \mathsf{SP})
\end{tikzcd}
$$

3.4 Questions

3.4.1 Bush

```
record Bush₀ (X : Type) : Type where
  coinductive
  field
    head : X
    tail : Bush₀ (Bush₀ X)
open Bush₀

{-# TERMINATING #-}
Bush₁ : (X → Y) → Bush₀ X → Bush₀ Y
head (Bush₁ f a) = f (head a)
tail (Bush₁ f a) = Bush₁ (Bush₁ f) (tail a)
```

We now look at the definition of a coinductive type `Bush`. Is `Bush X` a terminal coalgebra of any endofunctor? It turns out that previews scheme is no longer applicable. Th solution is to lift the space from `Type` to `Type → Type` and observe whether `Bush` is the terminal coalgebra of a higher endofunctor.

$$
\begin{tikzcd}
F \arrow[r, "\mathsf{unfold}\ \alpha^-"] \arrow[d, "\alpha^-"] & \mathsf{Bush}_0 \arrow[d, "{<\ \mathsf{head}\ ,\ \mathsf{tail}\ >}"] \\
H\ F \arrow[r, "H_1\ (\mathsf{unfold}\ \alpha^-)"] & H\ \mathsf{Bush}_0
\end{tikzcd}
$$

Bibliography

[1] ABBOTT, M., ALTENKIRCH, T., AND GHANI, N. Containers: Constructing strictly positive types. *Theoretical Computer Science 342*, 1 (2005), 3–27. Applied Semantics: Selected Topics.