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 15, 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

Conducted Research

In this section, we will cover the necessary background knowledge, review related literatures, outline the topics studied and illustrate the current challenge. We assume the reader has basic knowledge in logics and functional programming.

2.1 Our Language

We adopt Martin-Löf type theory - MLTT and category theory as the framework for our study. In particular, we use Agda, a programming language which implements MLTT, as our language in this report.

We introduce standard Agda syntax for common algebraic data types :

- `⊥` - Empty Type
- `⊤` - Unit Type
- `×` - Product Type
- `⊎` - Coproduct (Sum) Type

2.1.1 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 by outlining the development.

Simple Type Theory

Before MLTT, there is simple type theory introduced by Alonzo Church in 1940. It has algebraic types, function types and predicates, where the predicates can quantify over functions and other predicates, which is already in higher-order logics.

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 is also in a bigger universe.

MLTT

It is straight forward to define type family since we already have `Type`. A type family is essentially generalization of normal function:

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

Per Martin-Löf developed type theory further by adding type family. As a result, we can define type - `Vec n`, where `n` is the length of the vector. It also generalizes function type `→` into dependent function type `Π` and pair type `×` into dependent pair type `Σ`.

ITT and ETT

It is 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
```

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.

HoTT

The homotopy type theory provides a solution to lacking of extensionality in MLTT by adding the univalence axiom. The univalence axiom is It is introduced by Vladimir Voevodsky

Cubical TT

Finally, cubical type theory!!!

2.1.2 Category Theory

Categories

We define a category as follow:

```
record Cat : Type₁ where
  field
    Obj : Type
    Hom : Obj → Obj → Type
    id : ∀ {X} → Hom X X
    _∘_ : ∀ {X Y Z} → Hom Y Z → Hom X Y → Hom X Z
    idl : ∀ {X Y} (f : Hom X Y) → id ∘ f ≡ f
    idr : ∀ {X Y} (f : Hom X Y) → f ∘ id ≡ f
    ass : ∀ {X Y Z W} (f : Hom Z W) (g : Hom Y Z) (h : Hom X Y)
      → (f ∘ g) ∘ h ≡ f ∘ (g ∘ h)
    isSetHom : ∀ {X Y} → isSet (Hom X Y)

  |_| = Obj
  _[_,_] = Hom
```

Functors

```
record Func (ℂ 𝔻 : Cat) : Type₁ where
  open Cat
  field
    F₀ : | ℂ | → | 𝔻 |
    F₁ : ∀ {X Y} → ℂ [ X , Y ] → 𝔻 [ F₀ X , F₀ Y ]
```

Natural Transformations

```
record NatTrans {ℂ 𝔻} (F G : Func ℂ 𝔻) : Type where
  open Cat hiding (_∘_)
```

```
open Cat 𝔻 using (_∘_)
open Func F
open Func G renaming (F₀ to G₀ ; F₁ to G₁)
field
  η : (X : ∣ ℂ ∣) → 𝔻 [ F₀ X , G₀ X ]
  com : ∀ {X Y} (f : ℂ [ X , Y ]) → (η Y ∘ F₁ f) ≡ (G₁ f ∘ η X)
```

Algebra

```
record Alg {ℂ : Cat} (F : Func ℂ ℂ) : Type where
  open Cat
  open Func F
  field
    X : ∣ ℂ ∣
    α : ℂ [ F₀ X , X ]
```

2.2 Literature Review

TODO

2.3 Types and Categorical Semantics

2.3.1 Inductive Types

We give an informal definition of inductive types followed by some examples. An inductive type `A` is given by a finite number of data constructors, and a rule says how to define a function out of `A` to arbitrary type `B`.

Natural Number

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`.

```
data ℕ : Type where
  zero : ℕ
  suc : ℕ → ℕ
{-# BUILTIN NATURAL ℕ #-}
```

```
_ : ℕ ⊎ ℕ
_ = inj₁ zero

_ : ℕ × ℕ
_ = zero , zero
```

The `ℕ` itself is not so useful until we spell out its induction principle `indℕ`. In principle, whenever we need to define a function out of `ℕ`, we always using `indℕ`. That also corresponds to one of the Peano axioms.

```
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)

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

It is rather verbose to explicitly call induction every time. Fortunately, we are able to instead use pattern-matching in Agda, which implements induction internally and guarantees correctness.

```
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:

$$
\begin{tikzcd}
\mathsf{F\ A} \arrow[r, "\mathsf{F\ f}"] \arrow[d, "\alpha"] & \mathsf{F\ B} \arrow[d, "\beta"] \\
\mathsf{A} \arrow[r, "\mathsf{f}"] & \mathsf{B}
\end{tikzcd}
$$

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:

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

```
⊤⊎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
```

2.3.2 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:

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

2.4 Containers

2.4.1 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. ℕ is one of such types, and it can be trivially checked if we rewrite its constructors into an equivalent form:

```
data ℕ' : Type where
  zero : (⊤ → ⊤) → ℕ'
  suc : (⊤ → ℕ') → ℕ'
```

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.

2.4.2 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
```

2.4.3 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:

$$
\begin{tikzcd}
⟦ SP ⟧₀ (W SP) \arrow[r, "⟦ SP ⟧₁ (fold α)"] \arrow[d, "sup"] & ⟦ SP ⟧₀ X \arrow[d, "α"] \\
W SP \arrow[r, "fold α"] & X
\end{tikzcd}
$$

$$
\begin{tikzcd}
X \arrow[r, "unfold α⁻"] \arrow[d, "α⁻"] & M SP \arrow[d, "inf"] \\
⟦ SP ⟧₀ X \arrow[r, "⟦ SP ⟧₁ (unfold α⁻)"] & ⟦ SP ⟧₀ (M SP)
\end{tikzcd}
$$

2.4.4 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
```

2.5 Questions

2.5.1 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.

$$
\begin{tikzcd}
\mathtt{F} \arrow[r, "\mathtt{unfold}\ \alpha^{-}"] \arrow[d, "\alpha^{-}"] & \mathtt{Bush}_0 \arrow[d, "\mathtt{<\ head\ ,\ tail\ >}"] \\
\mathtt{H\ F} \arrow[r, "\mathtt{H}_1\ (\mathtt{unfold}\ \alpha^{-})"] & \mathtt{H\ Bush}_0
\end{tikzcd}
$$

Chapter 3

Research Outcomes

3.1 Higher-order Containers

3.1.1 Higher-order Functoriality

3.1.2 Syntax and Semantics

3.1.3 Algebraic Structure

3.1.4 As Simply-Typed λ-Calculus

Chapter 4

Future Work Plan

This is future work plan.

Chapter 5

Conclusions

This is conclusions.

Chapter 6

Appendix

This is appendix.

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.