University of Nottingham
UK | CHINA | MALAYSIA

My Report!

First year review report

Zhili Tian
Supervised by Prof. Thorsten Altenkirch
& Prof. Ulrik Buchholtz

Functional Programming Lab
School of Computer Science
University of Nottingham

July 9, 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.

We can even classify types according to their "types", aka the **kind**. To continue the previous example, Boolean and integer are stand-alone types and we say *"true" is term of boolean* directly. In contrast, list and binary tree are higher-kinded types (or parameterized datatypes as they need to be parameterized by other types), and we say *[1,2,3]"is a term of list of natural numbers*.

It is usually more interesting to work on higher-kinded types parameterized by an arbitrary type X instead of a concrete type, which has led to the emergence of **(parametric) polymorphism**. The `flatten` function, as the last step of tree sorting algorithm, which converts binary tree to list is a typical example such polymorphic functions, cause there is no special constraints upon the internal type X.

```
flatten : BTree X → List X
flatten leaf = []
flatten (node lt x rt) = flatten lt ++ (x :: flatten rt)
```

However, not all types are well-behaved in our language. Here is a typical counterexample:

```
{-# NO_POSITIVITY_CHECK #-}
data Λ : Set where
  lam : (Λ → Λ) → Λ

app : Λ → Λ → Λ
app (lam f) x = f x

self-app : Λ
self-app = lam (λ x → app x x)

Ω : Λ
Ω = app self-app self-app
```

The Ω represents the famous non-terminating λ-term `(λx.x x)(λx.x x)` which reduces to itself infinitely. It is valid in untyped λ-calculus but not typable in simply-typed λ-calculus.

1.2 Aims and Objectives

Using the language of **Type Theory** and adopting the semantics of **Category Theory**, we wish to achieve the following objectives:

- To develop the syntax and semantics of **Higher(-Kinded) Functors** and their natural transformations, which capture the definition of higher types and higher polymorphic functions
- To develop the syntax and semantics of **Higher(-Kinded) Containers** and their morphisms, which should give rise higher functors and their natural transformations
- To show higher container model is simply-typed category with family
- ...

1.3 Overview of the Report

In the rest of report, I will cover:

- Chapter 2 - Prerequisites: ...
- Chapter 3 - Literature Review: ...

- Chapter 4 - Conducted Research: Literature review, and topics studied. We introduce inductive types, containers, category with families, hereditary substitution.
- Chapter 5 - Future Work Plan: Our future plan!

Chapter 2

Prerequisites

We assume basic knowledge in logics and functional programming. We give overview of background materials and fix terminology by introducing type theory (with Agda) and category theory.

2.1 Type Theory

Martin-Löf Type Theory - MLTT is a formal language in mathematics logics, where all objects and functions are assigned to some types. Additionally, it introduces advanced concepts like dependent types, universe size, strong normalization, etc. avoiding paradoxes and being used as foundations of mathematics and programmings.

Inference rules are the fundamental set of rules that regulate all valid definitions and computations. Especially, we are interested in rules about the formation of contexts, types and terms. Take natural number as an example:

$$\frac{}{\vdash \mathbb{N}\ \mathsf{type}}$$

$$\frac{}{\vdash \mathsf{zero} : \mathbb{N}} \qquad \frac{}{\vdash \mathsf{suc} : \mathbb{N} \to \mathbb{N}}$$

$$\frac{\begin{array}{l}\Gamma, n : \mathbb{N} \vdash P(n)\ \mathsf{type} \\ \Gamma \vdash p_0 : P(\mathsf{zero}) \\ \Gamma \vdash p_s : \Pi\ (n : \mathbb{N}).\ P(n) \to P(\mathsf{suc}(n))\end{array}}{\Gamma \vdash \mathsf{rec}\mathbb{N}(p_0, p_s) : \Pi\ (n : \mathbb{N}).\ P(n)}$$

$$\frac{\Gamma,\ n : \mathbb{N} \vdash P(n)\ \mathsf{type} \quad \Gamma \vdash p_0 : P(\mathsf{zero}) \quad \Gamma \vdash p_s : \Pi\ (n : \mathbb{N}).\ P(n) \to P(\mathsf{suc}(n))}{\Gamma \vdash \mathsf{rec}\mathbb{N}(p_0,p_s,\mathsf{zero}) \doteq p_0 : P(\mathsf{zero})}$$

$$\frac{\Gamma,\ n : \mathbb{N} \vdash P(n)\ \mathsf{type} \quad \Gamma \vdash p_0 : P(\mathsf{zero}) \quad \Gamma \vdash p_s : \Pi\ (n : \mathbb{N}).\ P(n) \to P(\mathsf{suc}(n))}{\Gamma \vdash \mathsf{rec}\mathbb{N}(p_0,p_s,\mathsf{suc}(n)) \doteq p_s(n,\mathsf{rec}\mathbb{N}(p_0,p_s,n)) : P(\mathsf{suc}(n))}$$

2.2 Agda

Agda is dependently typed programming language and interactive theorem prover based on MLTT. We therefore use Agda as our meta language in our research and this report.

Natural Number

To define natural number $\mathbb{N}$ as a new type in Agda:

```
{- Formation Rule -}
data ℕ : Set where
  {- Introduction Rule -}
  zero : ℕ
  suc : ℕ → ℕ
```

We need to explicitly define type constructor $\mathbb{N}$ and data constructors `zero` and `suc`, which correspond to the formation rule and introduction rule.

```
{- Elimination Rule -}
recℕ : (P : ℕ → Set)
  → P zero
  → ((n : ℕ) → P n → P (suc n))
  → (n : ℕ) → P n
recℕ P p₀ pₛ zero = p₀
recℕ P p₀ pₛ (suc n) = pₛ n (recℕ P p₀ pₛ n)

_+2 : ℕ → ℕ
_+2 = recℕ (λ _ → ℕ) (suc (suc zero)) (λ _ ssn → suc ssn)
```

```
_+2' : ℕ → ℕ
zero +2' = suc (suc zero)
suc n +2' = suc (n +2)
```

The elimination rule, also called recursor in FP, tells how to define functions or proofs out of ℕ. We can define function _+2 using recℕ, or alternatively using pattern matching, which provides equivalent definition but syntactically better.

```
{- Computation Rule -}
compℕ₀ : ∀ {P p₀ pₛ}
  → recℕ P p₀ pₛ zero ≡ p₀
compℕ₀ = refl

compℕₛ : ∀ {P p₀ pₛ n}
  → recℕ P p₀ pₛ (suc n) ≡ pₛ n (recℕ P p₀ pₛ n)
compℕₛ = refl
```

Finally, the computation rule describes how eliminations behave on terms. It is primitively implemented in Agda type system and therefore trivially hold.

2.3 Category Theory

Chapter 3

Literature review

This is literature review!

Chapter 4

Topics Studied

In this section, we will cover the topics studied, related background knowledge and illustrate the issues we encountered. First, we introduce inductive types, coinductive types and their categorical semantics. Then we give the definition and some nice properties of containers, and how containers can be used to capture the semantics of types. Finally, we demonstrate that some types can not be naively defined through traditional containers.

4.1 Types and Categorical Semantics

4.1.1 Inductive Types

To avoid getting too deep into strict type theory, we give an informal (functional programming) 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 #-}
```

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)
```

Another Example

4.1.2 Inductive Types are Initial Algebras

In the category of Type, an algebra Alg of a given endofunctor F : Type → Type is:

- An object A : Type, and
- A morphism α : F A → A

A morphism between algebras (A , α) and (B , β) is defined as:

- A morphism f : A → B, and
- A commuting diagram:

$$\begin{array}{ccc} F\ A & \xrightarrow{F\ f} & F\ B \\ \downarrow{\scriptstyle \alpha} & & \downarrow{\scriptstyle \beta} \\ A & \xrightarrow{f} & B \end{array}$$

We therefore 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 as Initial Algebra

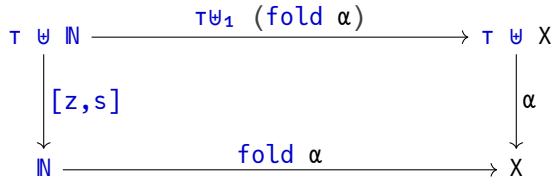
Natural number is the initial algebra of the `⊤ ⊎` functor. To prove this, we massage `ℕ` into an equivalent `Alg` form:

```
[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 check the following diagram commutes:




```
⊤⊎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
```

4.1.3 Coinductive Types

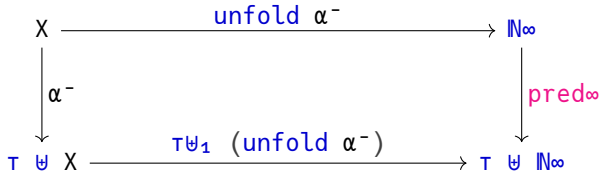
One of the greatest power of category theory is that, whenever we define something, we always get an opposite version for free. Indeed, if we inverse all the morphisms in above diagram, we will obtain a definition of conatural number `ℕ∞` and a result that `ℕ∞` is the terminal coalgebra of `⊤ ⊎`. Therefore, conatural number is defined as a coinductive type:

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

We also define the dual 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:



4.2 Containers

Chapter 5

Contribution

5.1 Higher-order Functoriality

Chapter 6

Future Work Plan

This is future work plan.

Chapter 7

Conclusions

This is conclusions.

Chapter 8

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.