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

May 29, 2025

Abstract

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

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 -term `(λx.x x)(λx.x x)` which reduces to itself infinitely. It is valid in untyped -calculus but not typable in simply-typed -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:

- Section 2 - Literature Review: ...
- Section 3 - Conducted Research: Literature review, and topics studied. We introduce inductive types, containers, category with families, hereditary substitution.

- Section 4 - Future Work Plan: Our future plan!

Chapter 2

Literature review

This is literature review!

Chapter 3

Conducted Research

In this section, we introduce basic concepts of type theory and category theory, as well as background knowledge of relevant fields. Then we give the definition of containers and demonstrate their properties. Finally, we introduce some contributions to the container model, such as higher functoriality, higher containers and their properties. We will use Agda.

3.1 Type Theory

The **Martin-Löf Type Theory - MLTT** is a formal language in mathematics logics. The idea of type theory is very close to the type system of functional programmings, which describes all objects and functions as types. Additionally, it introduces concepts like dependent types, universe size, strong normalization, etc. avoiding paradoxes and being used as foundations of mathematics and programmings.

Each types should be generated within some constraints, namely the corresponding *formation rule*, *introduction rule*, *elimination rule* and *computation rule*. For example when creating a type in Agda:

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

{- 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)

{- Computation Rule -}
recℕ₀ : ∀ {P p₀ pₛ} → recℕ P p₀ pₛ zero ≡ p₀
recℕ₀ = refl

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

3.1.1 Intensional and Extensional Type Theory

We need to be able to talk about equality of types. However, there are different notions of equality depends on whether you are looking at types from inside or outside. The definitional equality says two terms are equal if they are constructed in the same way. That is

3.2 Inductive Types

We now give a definition of inductive types. An inductive type `T` is given by a finite number of constructors `cᵢ`:

```
data T : Set where
  c₀ : A₀ → A₁ → ... → T
  c₁ : A₀ → A₁ → ... → T
```

such that there should also be corresponding elimination rule and computation rule.

It means `Bool`, `ℕ`, `List`, `BTree` are all inductive types.

Chapter 4

Future Work Plan

This is future work plan.

Chapter 5

Conclusions

This is conclusions.

Chapter 6

Appendix

This is appendix.