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 28, 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 algorthm, which converts binary tree to list is a typical example such polymorphic functions, cause there is no special constraints upon the internal type X.

We use **Agda**, a dependently typed functional programming language, in the rest of the report. Here is a piece of agda code that illustrates all the concepts - simple types, higher-kinded types, and polymorphic functions as discussed before:

```
data Bool : Set where
  false true : Bool
```

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

data List (X : Set) : Set where
  [] : List X
  _∷_ : X → List X → List X

_++_ : {X : Set} → List X → List X → List X
[] ++ ys = ys
(x ∷ xs) ++ ys = x ∷ (xs ++ ys)

data BTree (X : Set) : Set where
  leaf : BTree X
  node : BTree X → X → BTree X → BTree X

flatten : {X : Set} → 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:

- Section 2 - Prerequisites: General background knowleagde of this report. We introduce type theory, category theory.
- 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

Prerequisites

2.1 Type Theory

2.2 Category Theory

We will use the semantics of type theory to introduce basic category theory concepts. Categories, functors and natural transformations:

```
record Cat (Obj : Type1) : Type2 where
  infixr 9 _∘_
  field
    Hom : Obj → Obj → Type1
    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 : ∀ {W X Y Z} (f : Hom X W) (g : Hom Y X) (h : Hom Z Y)
          → (f ∘ g) ∘ h ≡ f ∘ (g ∘ h)

record Func {A B : Type1} (ℂ : Cat A) (𝔻 : Cat B) (F : A → B) : Type1 where
  open Cat
  field
    F1 : ∀ {X Y} → Hom ℂ X Y → Hom 𝔻 (F X) (F Y)
    F-id : ∀ {X} → F1 {X} (ℂ .id) ≡ 𝔻 .id
    F-∘ : ∀ {X Y Z} (f : Hom ℂ Y Z) (g : Hom ℂ X Y)
          → F1 (ℂ ._∘_ f g ) ≡ 𝔻 ._∘_ (F1 f) (F1 g)
```

```
record Nat {A B : Type1} (ℂ : Cat A) (𝔻 : Cat B)
  (F G : A → B) (FF : Func ℂ 𝔻 F) (GG : Func ℂ 𝔻 G) : Type1 where
  open Cat
  open Func
  field
    η : ∀ X → Hom 𝔻 (F X) (G X)
    nat : ∀ {X Y} (f : Hom ℂ X Y)
      → 𝔻 ._∘_ (GG .F1 f) (η X) ≡ 𝔻 ._∘_ (η Y) (FF .F1 f)
```

Chapter 3

Conducted Research

3.1 Literature review

3.2 Topics Studied

3.3 Questions

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.