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 11, 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 related background knowledge, literature reviews, topics studied and illustrate the issues we encountered. We assume basic knowledge in logics and functional programming. First, we fix our formal languages to type theory and category theory. Then we review current literatures and introduce inductive types, coinductive types, containers and some nice properties of them. Finally, we present questions of the limitations of existing schemes by examples.

2.1 Our Languages

2.1.1 Type Theory and Agda

Martin-Löf Type Theory - MLTT is a formal language in mathematics logics...

Should I do this?

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

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

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

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

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

2.1.2 Category Theory

TODO

2.2 Literature Review

TODO

2.3 Types and Categorical Semantics

2.3.1 Inductive Types

To avoid getting too deep into strict type theory semantics, 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
```

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

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:

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

```
⊤⊎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
```

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:

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

2.4 Containers

2.4.1 Strictly Positivity

Containers are categorical and type-theoretical abstraction to describe strictly positive datatypes. Being strictly positive is important when constructing new types. Disobeying such property would normally cause bad behavior in the type system. Formally, a strictly positive type is the type where all data constructors do not include itself on the left-side of a function arrow. `ℕ` is one of such types. It can be trivially checked if we rewrite the constructors into an equivalent arrow form:

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

One typical counterexample of strictly positive types is `Λ`:

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

`Λ` is not strictly positive as in constructor `lam`, `X` itself appears on the left side of the →. Indeed, using this definition, we can easily construct a not normalizable term.

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:

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

$$\begin{tikzcd}
X \arrow[r, "unfold\ \alpha^-"] \arrow[d, "\alpha^-"] & M\ SP \arrow[d, "inf"] \\
\llbracket\ SP\ \rrbracket_0\ X \arrow[r, "\llbracket\ SP\ \rrbracket_1\ (unfold\ \alpha^-)"] & \llbracket\ SP\ \rrbracket_0\ (M\ SP)
\end{tikzcd}$$

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 ....

Even better, we can define `Π` and `Σ` for containers:

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

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

We now look at the definition of a coinductive type `Bush`. What is the container for this type? Or how to represent it as a `M` type?

It turns out that previews scheme is no longer applicable.

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.