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

August 1, 2025

Abstract

Giving a short overview of the work in your project.[1]

Contents

Chapter 1

Introduction

1.1 Background and Motivation

TODO

1.2 Aims and Objectives

TODO

1.3 Progress to Date

TODO : progress and achievements during this stage, training courses, seminars and presentations.

Chapter 2

Formalization Settings

2.1 Notations

We assume the reader has basic knowledge in type theory and category theory. For simplicity and readability, we only present some Agda syntax, rather than formally introducing all underlying concepts.

2.1.1 Type Theory

- `A → B` - function type
- `⊤` - unit type
  - `tt : ⊤`
- `⊥` - empty type
- `A × B` - product type
  - `_,_ : A → B → A × B`
  - `proj₁ : A × B → A`
  - `proj₂ : A × B → B`
- `A ⊎ B` - coproduct type
  - `inj₁ : A → A ⊎ B`
  - `inj₂ : B → A ⊎ B`
- `Π A B` or `(a : A) → B a` - dependent function type

- `Σ A B` or `Σ[ a ∈ A ] B a` - dependent product type
    - `_,_ : (a : A) → B a → Σ A B`
    - $\texttt{proj}_1$ `: Σ A B → A`
    - $\texttt{proj}_2$ `: (ab : Σ A B) → B (`$\texttt{proj}_1$ `ab)`

2.1.2 Category Theory

- `ℂ : Cat` - Category
- `∣ ℂ ∣` - Objects of category
- `ℂ [ X , Y ]` - Morphisms of category
- `F : ℂ ⇒ 𝔻` or `Func ℂ 𝔻` - Functor
- $\texttt{F}\ {}_0$ - Mapping objects part of functor
- $\texttt{F}\ {}_1$ - Mapping morphisms part of functor
- `α : ∫ F ⇒ G` or `NatTrans F G` - Natural transformation.
- `[ f , g ]` - The unique morphism to the product

$$
\begin{tikzcd}
 & X \arrow[dl, "f"'] \arrow[d, dashed, "{[f,g]}"] \arrow[dr, "g"] & \\
A & A \times B \arrow[l, "\mathrm{proj}_1"] \arrow[r, "\mathrm{proj}_2"'] & B
\end{tikzcd}
$$

- `< f , g >` - The unique morphism from the coproduct

$$
\begin{tikzcd}
A \arrow[r, "\mathrm{inj}_1"] \arrow[dr, "f"'] & A + B \arrow[d, dashed, "{<f,g>}"] & B \arrow[l, "\mathrm{inj}_2"'] \arrow[dl, "g"] \\
 & X &
\end{tikzcd}
$$

2.2 Settings

TODO : our settings

Chapter 3

Conducted Research

In this section, we begin by reviewing related literature. We then introduce a categorical view of types, containers, W-types and M-types. We also introduce category with families - a model of type theory, and hereditary substitutions - a normalization technic for λ-calculus. Finally, we outline the current challenges.

3.1 Literature Review

TODO

3.2 Types as Algebras

3.2.1 Inductive Types are Initial Algebras

We use natural number as our example through this sections. Natural number `ℕ = {0, 1, 2, 3, ...}` can be defined as inductive type:

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

Given an endofunctor `F : Type ⇒ Type`, an algebra is defined as a carrier type `A : Type` and an evaluation function `α : F A → A`. The morphisms between algebras `(A , α)` and `(B , β)` are given by a function `f : A → B` such that the following diagram commutes:

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

Natural number (with its constructors) is the initial algebra of the `⊤⊎_` (**Maybe**) functor. To prove this, we first its constructors into an equivalent function form:

```
[z,s] : ⊤ ⊎ ℕ → ℕ
[z,s] (inj₁ tt) = zero
[z,s] (inj₂ n) = suc n
```

For the initiality, we show for any algebra there exists a unique morphism form algebra ℕ. That is to first undefine `fold`:

```
fold : (⊤ ⊎ X → X) → ℕ → X
fold α zero = α (inj₁ tt)
fold α (suc n) = α (inj₂ (fold α n))
```

then construct the mapping morphism part of `⊤⊎_`:
such that the following diagram commutes:

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

3.2.2 Coinductive Types are Terminal Coalgebras

One of the greatest power of category theory is that the opposite version of a theorem can always be derived for free. Now we inverse all the morphisms in above diagram and therefore dualize all concepts. We define conatural number as coinductive type and reverse `fold` to `unfold`:

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

```
unfold : (X → ⊤ ⊎ X) → X → ℕ∞
pred∞ (unfold α⁻ x) with α⁻ x
... | inj₁ tt = inj₁ tt
... | inj₂ x' = inj₂ (unfold α⁻ x')
```

such that the following diagram commutes:

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

Therefore we get the result of coinductive types are terminal coalgebras for free.

3.3 Containers

3.3.1 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. One typical counterexample is `Weird`:

```
{-# NO_POSITIVITY_CHECK #-}
data Weird : Type where
  foo : (Weird → ⊥) → Weird
```

`Weird` is not strictly positive, as `Weird` itself appears on the left-side of → in `foo`. Losing strict positivity can cause issues like non-normalizable, non-terminating, inconsistency, etc. In this case, a empty type term can be constructed using `¬weird` which is inconsistent to empty definition:

```
¬weird : Weird → ⊥
¬weird (foo x) = x (foo x)

bad : ⊥
bad = ¬weird (foo ¬weird)
```

3.3.2 Syntax and Semantics

A container is given by a shape `S` and with each shape a position `P`:

```
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 a functor:

```
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
```

3.3.3 Categorical Structures

Containers and their morphisms form a category. The morphisms are defined as follow:

```
record ContHom (SP TQ : Cont) : Type where
  constructor _◃_
  open Cont SP
  open Cont TQ renaming (S to T; P to Q)
  field
    f : S → T
    g : (s : S) → Q (f s) → P s
```

As containers give rise to functors, the morphisms of containers should naturally give rise to the morphisms of functors - the natural transformations:

```
⟦_⟧Hom : ContHom SP TQ → (X : Type) → ⟦ SP ⟧₀ X → ⟦ TQ ⟧₀ X
⟦ f ◃ g ⟧Hom X (s , k) = f s , k ∘ g s
```

3.3.4 Algebraic Structures

Containers are also known as **polynomial functors** as they forms a semiring structure. Namely, we can define one, zero, multiplication and addition for containers:

```
oneC : Cont
oneC = ⊤ ◃ λ x → ⊥

zeroC : Cont
zeroC = ⊥ ◃ λ ()

_×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 the semiring laws should hold. In fact, both multiplication and addition are commutative, associative and left- and right-annihilated by their units. It turns out they also exactly correspond to the initial, terminal, product and coproduct of the containers.

Even better, we can generalize `×` and `⊎` to `Π` and `Σ` for containers, which are finitary product and coproduct objects.

```
ΠC : (I → Cont) → Cont
ΠC {I} SPs = ((i : I) → SPs i .S) ◃ λ f → Σ[ i ∈ I ] SPs i .P (f i)

ΣC : (I → Cont) → Cont
ΣC {I} SPs = (Σ[ i ∈ I ] SPs i .S) ◃ λ (i , s) → SPs i .P s
```

3.3.5 W and M

With containers, 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 `ℕ` through the `⊤ ⊎` container:

```
⊤⊎Cont : Cont
⊤⊎Cont = (⊤ ⊎ ⊤) ◃ λ{ (inj₁ tt) → ⊥ ; (inj₂ y) → ⊤ }

ℕW : Type
ℕW = W ⊤⊎Cont
```

Dually, the general form of coinductive types - `M` type is just the terminal coalgebra of containers. `W` and `ℕ∞` are defined as follow:

```
record M (SP : Cont) : Type where
  coinductive
  field
    inf : ⟦ SP ⟧₀ (M SP)

ℕ∞M : Type
ℕ∞M = M ⊤⊎Cont
```

Finally, we have the following commutative diagrams for `W` and `M`:

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

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

We have now showed inductive types are initial algebras and coinductive types are terminal coalgebras.

3.4 Simply-typed Categories with Families

TODO : What is CwFs?

3.5 Normalization by Hereditary Substitutions

We introduce the hereditary substitutions for λ-calculus normalization. To build the syntax of λ-calculus, we first define types and contexts:

```
data Ty : Type where
  * : Ty
  _⇒_ : Ty → Ty → Ty

data Con : Type where
  • : Con
  _▹_ : Con → Ty → Con
```

We adopt the De Bruijn indices to represent bound variables without explicit naming.

```
data Var : Con → Ty → Type where
  vz : Var (Γ ▹ A) A
  vs : Var Γ A → Var (Γ ▹ B) A
```

The normal form λ-terms `Nf` are defined as either:

- λ-abstractions of `Nf`;
- Or neutral terms `Ne`, where `Ne` are variables `Var` applied to lists of `Nf`, called spine `Sp`.

```
data Nf : Con → Ty → Type where
  lam : Nf (Γ ▹ A) B → Nf Γ (A ⇒ B)
  ne : Ne Γ * → Nf Γ *

data Ne : Con → Ty → Type where
  _,_ : Var Γ A → Sp Γ A B → Ne Γ B

data Sp : Con → Ty → Ty → Type where
  ε  : Sp Γ A A
  _,_ : Nf Γ A → Sp Γ B C → Sp Γ (A ⇒ B) C
```

3.6 Questions

TODO

3.6.1 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)
```

How do we represent `Bush X` as a `M` type? It turns out that previews scheme is no longer applicable.

The key observation is to lift the space from `Type` to `Type → Type`. We can show `Bush` is the terminal coalgebra of a "higher" endofunctor. We need to also define a "higher" algebra:

```
H : (Type → Type) → Type → Type
H F X = X × F (F X)
```

$$
\begin{tikzcd}
\mathtt{F} \arrow[r, "\mathtt{unfold}\ \alpha^-"] \arrow[d, "\alpha^-"] & \mathtt{Bush_0} \arrow[d, "\mathtt{<\ head\ ,\ tail\ >}"] \\
\mathtt{H\ F} \arrow[r, "\mathtt{H_1\ (unfold}\ \alpha^-\mathtt{)}"] & \mathtt{H\ Bush_0}
\end{tikzcd}
$$

Chapter 4

Research Outcomes

TODO

4.1 Higher-order Containers

4.1.1 Syntax and Semantics

4.1.2 Categorical Structures

4.1.3 Algebraic Structures

4.1.4 Simply-typed Categories with Families

Chapter 5

Conclusions

This is conclusions.

Chapter 6

Future Work Plan

This is future work plan.

Chapter 7

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.