University of Nottingham
UK | CHINA | MALAYSIA

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

July 30, 2025

Abstract

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

Contents

1 Introduction **1**
- 1.1 Background and Motivation . . . 1
- 1.2 Aims and Objectives . . . 1
- 1.3 Progress to Date . . . 1

2 Formalization Settings **2**
- 2.1 Type Theory . . . 2
- 2.2 Category theory . . . 3
- 2.3 Non-Cubical Settings . . . 3

3 Conducted Research **5**
- 3.1 Literature Review . . . 5
- 3.2 Types as Algebras . . . 5
  - 3.2.1 Inductive Types are Initial Algebras . . . 5
  - 3.2.2 Coinductive Types are Terminal Coalgebras . . . 6
- 3.3 Containers . . . 7
  - 3.3.1 Strict Positivity . . . 7
  - 3.3.2 Syntax and Semantics . . . 7
  - 3.3.3 Categorical Structure . . . 8
  - 3.3.4 Semiring Structure . . . 9
  - 3.3.5 W and M . . . 9
- 3.4 Simply-Typed Category with Families . . . 10
- 3.5 Hereditary Substitutions . . . 10
- 3.6 Questions . . . 11
  - 3.6.1 Bush . . . 11

4 Research Outcomes **13**
- 4.1 2nd-order Containers . . . 13

4.1.1 Syntax and Semantics 13
4.2 Higher-order Containers 14
4.2.1 Syntax and Semantics 14
4.2.2 Categorical Structure 14
4.2.3 Algebraic Structure 14
4.2.4 Simply-Typed Lambda Calculus 14

5 Conclusions **15**

6 Future Work Plan **16**

7 Appendix **17**

References **17**

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 Type Theory

We assume the reader has basic knowledge in type theory. For simplicity and readability, we only introduce Agda syntax to fix terminology, rather than formally introducing all underlying concepts.

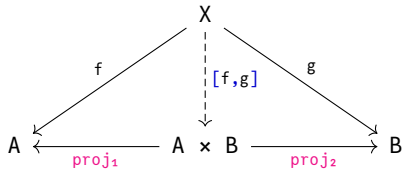
Types

- `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

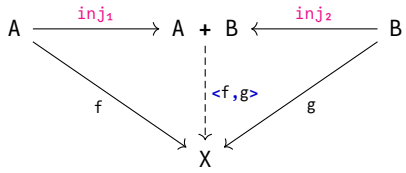
- `Σ A B` or `Σ[ a ∈ A ] B a` - dependent product type
  - `_,_ : (a : A) → B a → Σ A B`
  - `proj₁ : Σ A B → A`
  - `proj₂ : (ab : Σ A B) → B (proj₁ ab)`

2.2 Category theory

- `ℂ : Cat` - Category
- `| ℂ |` - Objects of category
- `ℂ [ X , Y ]` - Morphisms of category
- `F : ℂ ⇒ 𝔻` or `Func ℂ 𝔻` - Functor
- `F ₀` - Mapping objects part of functor
- `F ₁` - Mapping morphisms part of functor
- `α : ∫ F ⇒ G` or `NatTrans F G` - Natural transformation.
- `[ f , g ]` - The unique morphism to the product



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



2.3 Non-Cubical Settings

Terence Tao once describes three stages in mathematical learning and practice: pre-rigorous, rigorous, and post-rigorous. The idea of post-rigorous is that,

when one already know how to do thins rigorously, then he can move fluently between intuition, informal reasoning, and formal rigor as needed.

In our context, we conduct theoretical research within HoTT, exploring concepts such as the interpretation of containers as endofunctors on h-level sets. This requires us to explicitly track h-level fields such as `isSet` in Cubical Agda. We did formalizations in both MLTT and CTT and decided to continue and present our work using vanilla Agda.

We now explicitly state our working assumptions. H-level checking is hidden, therefore equalities between equalities within set-level structures are ignored. We also assume function extensionality and minimize the use of universe levels for simplicity.

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

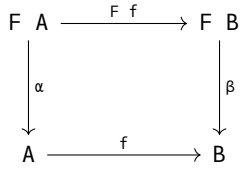
3.2 Types as Algebras

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:



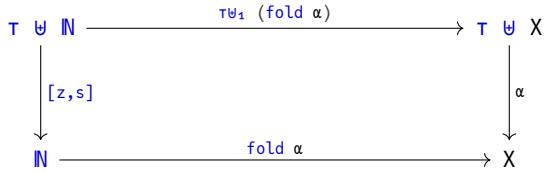
Natural number (with its constructors) is the initial algebra of the `⊤⊎_` (**Maybe**) functor. To prove this, we first its constructors into an equivalent function form:

```
[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 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:



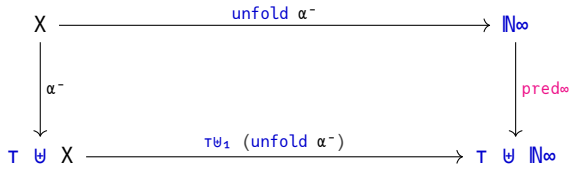
3.2.2 Coinductive Types are Terminal Coalgebras

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 dualize all the concepts. We define conatural number as a coinductive type and inverse of `fold`, which is `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:



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 in the domain. One typical counterexample is `Weird`:

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

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

bad : ⊥
bad = ¬weird (foo ¬weird)
```

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

3.3.2 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
```

3.3.3 Categorical Structure

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 Semiring Structure

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 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

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
```

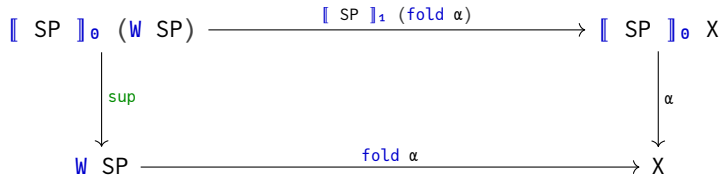
From the definition, `W` is an inductive type that specified by a container. In another word, any inductive type can be specified by a strictly positive functor. We can retrieve the definition of `ℕ` through the `⊤ ⊎` functor:

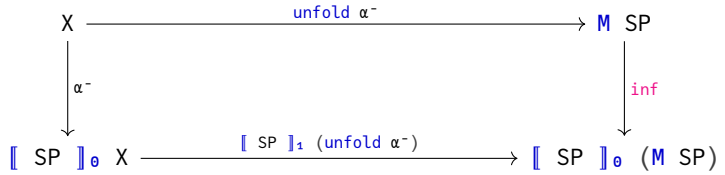
```
⊤⊎Cont : Cont
⊤⊎Cont = (⊤ ⊎ ⊤) ◃ λ{ (inj₁ tt) → ⊥ ; (inj₂ y) → ⊤ }

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`:





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

3.4 Simply-Typed Category with Families

3.5 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 Ctx : Type where
  • : Ctx
  _▷_ : Ctx → Ty → Ctx
```

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

```
data Var : Ctx → 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 : Ctx → Ty → Type where
  lam : Nf (Γ ▷ A) B → Nf Γ (A ⇒ B)
  ne : Ne Γ ∗ → Nf Γ ∗

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

data Sp : Ctx → 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

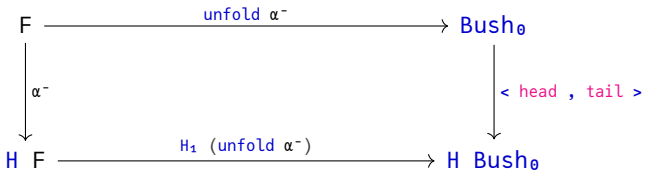
```
record Bush₀ (X : Type) : Type where
  coinductive
```

```
  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)
```



Chapter 4

Research Outcomes

TODO

4.1 2nd-order Containers

4.1.1 Syntax and Semantics

We first define the 2nd-order containers before going straight to general higher order containers

```
record 2Cont : Type₁ where
  constructor _◃_◃_◃_
  inductive
  field
    S : Type
    PX : S → Type
    PF : S → Type
    RF : (s : S) → PF s → 2Cont

record 2ContHom (SPR TQL : 2Cont) : Type where
  constructor _◃_◃_◃_
  inductive
  eta-equality
  open 2Cont SPR
  open 2Cont TQL renaming (S to T; PX to QX; PF to QF; RF to LF)
  field
    f : S → T
```

```
    gx : (s : S) → QX (f s) → PX s
    gf : (s : S) → QF (f s) → PF s
    hf : (s : S) (qf : QF (f s)) → 2ContHom (RF s (gf s qf)) (LF (f s) qf)

record 2⟦_⟧₀ (SPR : 2Cont) (F : Cont) (X : Type) : Type where
  constructor _,_,_
  inductive
  eta-equality
  open 2Cont SPR
  field
    s : S
    kx : PX s → X
    kf : (p : PF s) → ⟦ F ⟧₀ (2⟦ RF s p ⟧₀ F X)

{-# TERMINATING #-}
2⟦_⟧₁ : (SPR : 2Cont)
  → {F G : Cont} → ContHom F G
  → {X Y : Type} → (X → Y)
  → 2⟦ SPR ⟧₀ F X → 2⟦ SPR ⟧₀ G Y
2⟦ SPR ⟧₁ {F} {G} α {X} {Y} f (s , kx , kf) = s , (f ∘ kx)
   , (λ pf → ⟦ α ⟧Hom (2⟦ RF s pf ⟧₀ G Y) (⟦ F ⟧₁ (2⟦ RF s pf ⟧₁ α f) (kf pf)))
  where open 2Cont SPR

{-# TERMINATING #-}
2⟦_⟧Hom : {H J : 2Cont} → 2ContHom H J
  → (F : Cont) (X : Type)
  → 2⟦ H ⟧₀ F X → 2⟦ J ⟧₀ F X
2⟦ f ◃ gx ◃ gf ◃ hf ⟧Hom F X (s , kx , kf) = f s , (kx ∘ gx s)
  , (λ pf → ⟦ F ⟧₁ (2⟦ hf s pf ⟧Hom F X) (kf (gf s pf)))
```

4.2 Higher-order Containers

4.2.1 Syntax and Semantics

4.2.2 Categorical Structure

4.2.3 Algebraic Structure

4.2.4 Simply-Typed Lambda Calculus

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.