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 12, 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 . . . 2

1.3 Progress to Date . . . 2

2 Notations and Settings **4**

2.1 Notations . . . 4

2.1.1 Type Theory . . . 4

2.1.2 Category Theory . . . 5

2.2 Formalization Settings . . . 6

3 Conducted Research **7**

3.1 Literature Review . . . 7

3.2 Types as Algebras . . . 7

3.2.1 Inductive Types as Initial Algebras . . . 7

3.2.2 Coinductive Types as Terminal Coalgebras . . . 8

3.3 Containers . . . 9

3.3.1 Strict Positivity . . . 9

3.3.2 Syntax . . . 10

3.3.3 Semantics . . . 12

3.3.4 W and M . . . 12

3.4 Categories with Families . . . 13

3.4.1 Simply Typed . . . 13

3.4.2 Adding Functions . . . 14

3.5 Normalization by Hereditary Substitutions . . . 15

3.6 Questions . . . 17

3.6.1 Bush . . . 17

4 Research Outcomes **18**
4.1 Higher-Order Containers 18
4.1.1 Syntax 18
4.1.2 Semantics 20
4.1.3 Simply Typed Categories with Families 20

References **21**

Chapter 1

Introduction

1.1 Background and Motivation

Martin-Löf Type Theory (MLTT) provides a constructive foundation for mathematics and programming. The essence of type theory is to represent propositions as types and their proofs as programs. Homotopy Type Theory (HoTT), a recent developed variant, extends MLTT with ideas from homotopy theory, interpreting types as spaces and equalities as paths. This richer interpretation enables reasoning about higher-dimensional structures, univalence, and equivalences between types.

Within these settings, we explore and generalize the concepts of containers, which offer a uniform way to represent a range of "well-behaved" inductive types. For many data types like trees, list, etc, container provides an alternative way to visualize and reason about their definition and data manipulating. It abstracts a data type into two components: a set of all possible shapes describing the overall form, and within each shape a position telling where data can be stored.

For example, the definition of `List X` can be visualized as a type with countable many shapes `S` and, for each concrete shape `sᵢ : S`, there are `i` many data `x : X` stored.

```
data List (X : Type) : Type where
  s₀ : List X
  s₁ : X → List X
  s₂ : X → X → List X
  s₃ : X → X → X → List X
  {- ... -}
```

Data transfer can be understood and processed in a similar way. That is to to case analysis on shapes and, for each shape, specifying the target shape and how the target positions are obtained from the source positions. For example the `tail` function:

```
tail : List X → List X
tail s₀ = s₀
tail (s₁ x) = s₀
tail (s₂ x y) = s₁ y
tail (s₃ x y z) = s₂ y z
{- ... -}
```

1.2 Aims and Objectives

Containers provide a compositional way to define and manipulate data types, making their semantics explicit and supporting generic programming, categorical analysis, and proofs about programs. In the Conducted Research section, we will present a detailed account of containers, outlining both their capabilities and their limitations. Our work generalize ordinary containers to higher-order containers, capturing and reinterpreting a broader range of concepts and structures in type theory. More specifically, we wish to:

- analyze higher(-order) data types with categorical methods
- define the meaning of higher functoriality
- define the syntax category of higher containers
- explore the categorical and algebraic properties of higher containers
- interpret the syntax to capture higher functoriality
- show that higher containers give rise to a simply typed category with families
- ...

1.3 Progress to Date

Over the first few months, I engaged in a broad exploration of type theory and category theory. I have read fundational textbooks including *Categories for the*

Lazy Functional Programmer, *The Tao of Types*, *Homotopy Type Theory - Univalent Foundations of Mathematics*. I deepened related background knowledge and developed Agda programming skills by formalizing and reinterpreting many existing papers. After the training phase, I shifted my focus toward learning and developing my current research area - containers. I met regularly with my supervisor for weekly discussions to track progress, discuss current problems, and plan next steps.

Beside my own research areas, I also learned ongoing research questions and outcomes by actively participating the *Type Theory Cafe* and *Functional Programming Lunch*, which are both internal seminars series within FP Lab. I also gave a talk about my master research - "Algebraic Effects in Haskell" on one of the FP Lunch seminars.

I had the opportunity to attend large-scale academic events. I attended *Midland Graduate School 2025* in Sheffield, where I followed the courses on coalgebras, the Curry-Howard correspondence, refinement type in Haskell. As part of MGS25, I also assisted in teaching a course on category theory by answering questions from the exercise sessions and preparing Latex solutions for the lecture notes. Before MGS25, I participated MGS24 in Leicester and MGS Christmas 24 in Sheffield, where I benefited from many courses and talks. Additionally, I also attended the *TYPES 2025* in Glasgow, a five-day international conference covering a wide range of topics in type theory.

In the spring term, I worked as a teaching assistant for several modules. I helped marking exercises and exams, as well as running weekly tutorials for the module *Languages and Computation*. I also acted as a lab tutor for *Programming Paradigms*, where I supported students on Haskell exercises during weekly lab sessions.

Chapter 2

Notations and Settings

In this section, we introduce syntax from *agda-std* and *cubical* Agda standard libraries used as notation throughout our formalizations. We assume the reader is familiar with Martin-Löf Type Theory (MLTT) and its variants, particularly Homotopy Type Theory (HoTT) and Cubical Type Theory (CTT). Additionally, we presuppose a basic understanding of category theory and do not formally introduce its foundational concepts.

2.1 Notations

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`
    - `proj₁ : Σ A B → A`
    - `proj₂ : (ab : Σ A B) → B (proj₁ ab)`

2.1.2 Category Theory

- `ℂ : Category` - Category
- `| ℂ | : Type` - Objects of category
- `ℂ [ X , Y ] : Set` - Morphisms of category
- `F : ℂ ⇛ 𝔻` or `Functor ℂ 𝔻` - Functor
- `F₀ : Type → Type` - Mapping objects part of functor
- `F₁ : (X → Y) → F₀ X → F₀ Y` - 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 Formalization Settings

We conduct theoretical research within HoTT, such as interpreting containers as endofunctors on h-level sets. This requires us to explicitly track h-level fields such as `isSet` in Cubical Agda, which can be quite bureaucratic and tedious in practice. Instead, to focus on mathematical ideas, we choose to do some post-rigorous math since we are already familiar with how to carry out such constructions rigorously. As such, our formalizations are carried out in plain Agda for intuitionistic purposes.

To be more specific, h-level definitions and checks are omitted. In this case, we are essentially working within a wild category setting. We also assume function extensionality and minimize the use of universe levels to improve both simplicity and readability.

For example, a normal container and its interpretation are defined as:

```
record Cont : Type₁ where
  constructor _◃_&_&_
  field
    S : Type
    P : S → Type
    isSetS : isSet S
    isSetP : (s : S) → isSet (P s)

⟦_⟧ : Cont → Functor (SET ℓ-zero) (SET ℓ-zero)
⟦ S ◃ P & isSetS & isSetP ⟧
  = record
  { F-ob = λ (X , isSetX) →
    (Σ[ s ∈ S ] (P s → X)) , isSetΣ isSetS (λ s → isSet→ isSetX)
  ; F-hom = λ f (s , k) → s , λ p → f (k p)
  ; F-id = λ i (s , k) → s , k
  ; F-seq = λ f g i (s , k) → s , λ p → g (f (k p))
  }
```

We would hide the field of `isSetS` and `isSetP` from `Cont`. Meanwhile we focus on the constructions and often leave `F-id` and `F-seq` implicit. `Set` means h-level set and `Type` means any type.

Chapter 3

Conducted Research

In this section, we begin by reviewing related literature. We then introduce a categorical view of types, containers, W type and M type. To facilitate later definition of higher containers in the research outcomes section, we also present related topics including categories with families, and normalization by hereditary substitutions. Finally, we outline the current challenges in this area.

3.1 Literature Review

TODO

3.2 Types as Algebras

3.2.1 Inductive Types as 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 folding:

```
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 maybe functor:
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 as 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 unfolding:

```
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:

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

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. Here is a counterexample:

Weird

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

`Weird` is not strictly positive, as 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, which is inconsistent to empty definition:

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

bad : ⊥
bad = ¬weird (foo ¬weird)
```

3.3.2 Syntax

A normal 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
```

Containers are used to describe generic type (or generic class). For example, a `List` container is `ℕ ◃ Fin`, which suggests a list should have countable many possible shapes, with each shape uniquely represents the length of list (or the number of elements).

```
data Fin : ℕ → Type where
  zero : Fin (suc zero)
  suc : ∀ {n} → Fin n → Fin (suc n)

ListC : Cont
ListC = ℕ ◃ Fin
```

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
```

such that the identity morphisms and compositions exist.

The intuition behind morphism is the process manipulating and transferring data between data structures. The first component `f` tells how to map shapes and second component `g` defines how to obtain data. Notice that mapping in `g` is backward, because data could be lost or duplicated during the transfer. For example, to define the tail function on list:

```
tailC : ContHom ListC ListC
tailC = f ◃ g
  where
  f : ℕ → ℕ
  f zero = zero
  f (suc n) = n

  g : (n : ℕ) → Fin (f n) → Fin n
  g zero ()
  g (suc n) x = suc x
```

Algebraic Structures

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

```
oneC : Cont
oneC = ⊤ ◃ const ⊥

zeroC : Cont
zeroC = ⊥ ◃ ⊥-elim

_×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 each is left- and right-annihilated by its corresponding unit. Moreover, they correspond precisely to the initial object, terminal object, product and coproduct within the category of containers.

Even better, we can generalize the binary product and coproduct to finite product and coproduct in the category of containers.

```
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
```

3.3.3 Semantics

A container should give rise to a endofunctor of `Set`. Again, we explicitly distinguish mapping objects part and mapping morphisms part of a functor:

```
record ⟦_⟧ (SP : Cont) (X : Type) : Type where
  constructor _,_
  open Cont SP
  field
    s : S
    k : P s → X

⟦_⟧₁ : (SP : Cont) → (X → Y) → ⟦ SP ⟧ X → ⟦ SP ⟧ Y
⟦ SP ⟧₁ f (s , k) = s , f ∘ k
```

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 ⟧ X → ⟦ TQ ⟧ X
⟦ f ◃ g ⟧Hom X (s , k) = f s , k ∘ g s
```

3.3.4 W and M

We now present the general form of inductive types, which is the W type:

```
data W (SP : Cont) : Type where
  sup : ⟦ SP ⟧ (W SP) → W SP
```

By definition, W is uniquely represented by an algebra specified by a container. In another word, any inductive type can be characterized by a containers. Indeed, we can retrieve an equivalent definition of natural number through the maybe 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 conatural number are defined as follow:

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

ℕ∞M : Type
ℕ∞M = M ⊤⊎Cont
```

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

$$\begin{tikzcd}
⟦ SP ⟧ (W SP) \arrow[r, "⟦ SP ⟧_1 (fold\ α)"] \arrow[d, "sup"] & ⟦ SP ⟧ X \arrow[d, "α"] \\
W SP \arrow[r, "fold\ α"] & X
\end{tikzcd}$$

$$\begin{tikzcd}
X \arrow[r, "unfold\ α^-"] \arrow[d, "α^-"] & M SP \arrow[d, "inf"] \\
⟦ SP ⟧ X \arrow[r, "⟦ SP ⟧_1 (unfold\ α^-)"] & ⟦ SP ⟧ (M SP)
\end{tikzcd}$$

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

3.4 Categories with Families

The Categories with Families (CwFs) are categorical framework for interpreting dependent type theory. They model core concepts like context formation, substitution, weakening, etc. However, CwFs are more related to an alternative type theory - a substitution calculus by Martin-Löf, in which the substitution is first class and explicitly formulated.

3.4.1 Simply Typed

We define CwFs as quotient inductive inductive type in Agda. A simply typed Category with families (SCwF) consists of the following:

- a category ℂ of contexts - `Con : Type` and context substitutions - `Tms : Con → Con → Set`

```
id  : Tms Γ Γ
_∘_ : Tms Δ Γ → Tms Θ Δ → Tms Θ Γ
idl : id ∘ γ ≡ γ
idr : γ ∘ id ≡ γ
ass : (γ ∘ δ) ∘ θ ≡ γ ∘ (δ ∘ θ)
```

- a terminal object - `• : | ℂ |` represents the empty substitution

```
•   : Con
ε   : Tms Γ •
•-η : γ ≡ ε
```

- a set of types - `Ty : Type`
- a family of presheaves - `Tm _ A : ℂᵒᵖ ⇒ Set` for each `A : Ty`, which send `Γ : Con` to `Tm Γ A : Set`, and `γ : Tms Δ Γ` to `_[ γ ] : Tm Γ A → Tm Δ A`

```
_[_] : Tm Γ A → Tms Δ Γ → Tm Δ A
[id] : t [ id ] ≡ t
[∘] : t [ γ ∘ δ ] ≡ t [ γ ] [ δ ]
```

- a context comprehension operation, which defines context extension and its constraints

```
_▹_ : Con → Ty → Con
_,_ : Tms Δ Γ → Tm Δ A → Tms Δ (Γ ▹ A)
π₁  : Tms Δ (Γ ▹ A) → Tms Δ Γ
π₂  : Tms Δ (Γ ▹ A) → Tm Δ A
π₁β : π₁ (γ , t) ≡ γ
π₂β : π₂ (γ , t) ≡ t
πη  : π₁ γ , π₂ γ ≡ γ
,∘  : (γ , t) ∘ δ ≡ γ ∘ δ , t [ δ ]
```

3.4.2 Adding Functions

The CwF on its own typically doesn't provide anything about types. We now add normal function type to the framework.

```
_⇒_ : Ty → Ty → Ty
lam : Tm (Γ ▹ A) B → Tm Γ (A ⇒ B)
app : Tm Γ (A ⇒ B) → Tm (Γ ▹ A) B
⇒β : app (lam t) ≡ t
⇒η : lam (app t) ≡ t

_↑ : Tms Δ Γ → Tms (Δ ▹ A) (Γ ▹ A)
↑≡ : γ ↑ ≡ γ ∘ π₁ {A = A} id , π₂ id
lam[] : lam t [ γ ] ≡ lam (t [ γ ↑ ])
app[] : app (t [ γ ]) ≡ app t [ γ ↑ ]
```

With this definition of function type, we can retrieve normal application in λ-calculus `_$_` from `app` and singleton extension `<_>`:

```
<_> : Tm Γ A → Tms Γ (Γ ▹ A)
< t > = id , t

_$_ : Tm Γ (A ⇒ B) → Tm Γ A → Tm Γ B
t $ u = app t [ < u > ]
```

3.5 Normalization by Hereditary Substitutions

We introduce the hereditary substitutions technique for λ-calculus normalization. To build the syntax of λ-calculus, we first define types (with a base type and function) 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 λ-terms follows the normal convention:

```
data Tm : Con → Ty → Set where
  var : Var Γ A → Tm Γ A
  lam : Tm (Γ ▹ A) B → Tm Γ (A ⇒ B)
  app : Tm Γ (A ⇒ B) → Tm Γ A → Tm Γ B
```

The normal form λ-terms are defined as either: λ-abstractions of normal forms; or neutral terms, where neutral terms are variables applied to lists of normal forms, called the spines.

```
mutual
  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
```

The normalization process is divided into three parts, handling variables, λ-abstractions, and applications separately. A variable is normalized by performing η-expansion to the fullest extent possible:

```
mutual
  nvar : Var Γ A → Nf Γ A
  nvar x = ne2nf (x , ε)

  ne2nf : Ne Γ A → Nf Γ A
  ne2nf {A = ∗} e = ne e
  ne2nf {A = A ⇒ B} (v , ns) =
    lam (ne2nf {A = B} (vs v , appSp (wkSp vz ns) (nvar vz)))
```

Term applications are much more involved, where it is the actual hereditary substitutions happening. To do this, we need to define how to substitute variables inside normal forms and spines, and how to fold a spine on a normal form. The normal form application function `napp` is defined to launch the substitution process:

```
napp : Nf Γ (A ⇒ B) → Nf Γ A → Nf Γ B
napp (lam t) u = t [ vz := u ]
```

Finally, we have the normalization function from λ-terms to normal forms:

```
nf : Tm Γ A → Nf Γ A
nf (var x) = nvar x
nf (lam x) = lam (nf x)
nf (app t u) = napp (nf t) (nf u)
```

3.6 Questions

3.6.1 Bush

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

How do we represent `Bush X` as a M type? It turns out that previews scheme is no longer applicable. To address the issue, the key observation is to lift the space from `Set` to `Set → Set`. 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)
```

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

Chapter 4

Research Outcomes

4.1 Higher-Order Containers

TODO

4.1.1 Syntax

We first observe the syntax of first-order functors and containers

```
⊤⊎_ : Type → Type
⊤⊎ X = ⊤ ⊎ X
```

The maybe functor

Through some observation, higher containers are much like a normal form of λ-calculus. We therefore reuse some of the syntax from previous sections:

```
mutual
  data Nf : Con → Ty → Type₁ where
    lam : Nf (Γ ▹ A) B → Nf Γ (A ⇒ B)
    ne : Ne Γ * → Nf Γ *

  record Ne (Γ : Con) (B : Ty) : Type₁ where
    constructor _◃_◃_
    inductive
    field
      S : Type
      P : Var Γ A → S → Type
      R : (x : Var Γ A) (s : S) → P x s → Sp Γ A B
  data Sp : Con → Ty → Ty → Type₁ where
    ε   : Sp Γ A A
    _,_ : Nf Γ A → Sp Γ B C → Sp Γ (A ⇒ B) C
```

We define higher-order containers as closed normal forms. From this point onward, we focus on normal forms in general, as higher-order containers are merely special cases of the former.

```
HCont : Ty → Type₁
HCont A = Nf • A
```

Categorical Structures

```
mutual
  data NfHom : (t u : Nf Γ A) → Type₁ where
    lam : NfHom t u → NfHom (lam t) (lam u)
    ne  : NeHom spr tql → NfHom (ne spr) (ne tql)

  record NeHom {Γ} {B} (spr tql : Ne Γ B) : Type₁ where
    constructor _◃_◃_
    inductive
    open Ne spr
    open Ne tql renaming (S to T; P to Q; R to L)
    field
      f : S → T
      g : (x : Var Γ A) (s : S) → Q x (f s) → P x s
      h : (x : Var Γ A) (s : S) (q : Q x (f s))
        → SpHom (R x s (g x s q)) (L x (f s) q)

  data SpHom : (t u : Sp Γ A B) → Type₁ where
    ε   : SpHom ts ts
    _,_ : NfHom t u → SpHom ts us → SpHom (t , ts) (u , us)
```

Algebraic Structures

4.1.2 Semantics

We need to first interpret types, contexts and variables:

```
⟦_⟧t : Ty → Type₁
⟦ * ⟧t = Type
⟦ A ⇒ B ⟧t = ⟦ A ⟧t → ⟦ B ⟧t

⟦_⟧c : Con → Type₁
⟦ • ⟧c = Lift (suc zero) ⊤
⟦ Γ ▹ A ⟧c = ⟦ Γ ⟧c × ⟦ A ⟧t

⟦_⟧v : Var Γ A → ⟦ Γ ⟧c → ⟦ A ⟧t
⟦ vz ⟧v (as , a) = a
⟦ vs x ⟧v (as , a) = ⟦ x ⟧v as
```

The semantics of normal forms:

```
⟦_⟧nf : Nf Γ A → ⟦ Γ ⟧c → ⟦ A ⟧t
⟦ lam t ⟧nf as a = ⟦ t ⟧nf (as , a)
⟦ ne spr ⟧nf as = ⟦ spr ⟧ne as

⟦_⟧ne : Ne Γ * → ⟦ Γ ⟧c → Type
⟦_⟧ne {Γ} (S ◃ P ◃ R) as =
  Σ[ s ∈ S ] ({A : Ty} (x : Var Γ A) (p : P x s)
    → ⟦ R x s p ⟧sp as (⟦ x ⟧v as))

⟦_⟧sp : Sp Γ A B → ⟦ Γ ⟧c → ⟦ A ⟧t → ⟦ B ⟧t
⟦ ε ⟧sp as a = a
⟦ t , ts ⟧sp as f = ⟦ ts ⟧sp as (f (⟦ t ⟧nf as))
```

Again, higher containers are just special case of normal forms:

```
⟦_⟧ : HCont A → ⟦ A ⟧t
⟦ t ⟧ = ⟦ t ⟧nf (lift tt)
```

4.1.3 Simply Typed Categories with Families

We can show that the categories of higher containers get rise to a SCwF with functions, where `Ty` is `Ty`, `Con` is `Con`, and `Tm` is `Nf`. We need to define the syntax for context substitutions.

```
data Nfs : Con → Con → Type₁ where
  ε : Nfs Γ •
  _,_ : Nfs Δ Γ → Nf Δ A → Nfs Δ (Γ ▹ A)
```

Similar to `Tms`, the `Nfs` is a list of normal forms.

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.