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

August 18, 2025

Abstract

Containers are semantic and categorical way to talk about strictly positive types. It is shown in previous work that the category of containers is closed under compositions, products, coproducts and exponentials. More extensions and variants, such as indexed containers, generalized containers are studied to capture a boarder class. In this project, we aim to develop the syntax and semantics of the higher-order containers and do formalization in Agda. The idea of higher containers is to provide a semantics for higher strictly positive types, such as monad transformers with kinds `(Type → Type) → Type → Type`. We define a family of categories of containers indexed by their kinds and explore the properties. We also show that the higher containers give rise to a model of simply typed -calculus.

Contents

Chapter 1

Introduction

1.1 Background and Objectives

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 the language of type theory, we explore 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 shapes describing the overall form, and within each shape a set of positions telling where data can be stored.

For example, the definition of a list can be visualized as a structure with all finite shapes and, for each shape of length `n`, there are `n` many data 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 componentwisely specify the mapping on shapes and positions. For example the tail function on list:

```
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
{- ... -}
```

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. The unary containers, however, are not capable of capturing data types with higher kinds. For example, the type of the *Maybe* monad transformer is not `Type → Type` but `(Type → Type) → Type → Type`.

```
MaybeT : (Type → Type) → Type → Type
MaybeT M X = M (⊤ ⊎ X)
```

Despite the higher kinds, from a container point of view, `MaybeT` could still be viewed as a compound of shapes, positions and applications. Our work therefore aims to generalize the containers to a higher-order sense, to capture and reinterpret a broader range of concepts and structures. More specifically, we wish to:

- provide a syntax for higher containers
- explore the categorical and algebraic properties of higher containers
- provide a categorical semantics of higher types
- show that higher containers give rise to a simply typed category with families
- ...

1.2 Progress to Date

Over the first few months, I engaged in a broad exploration of type theory and category theory. I worked on 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.

1.3 Settings

We assume the reader is familiar with MLTT and HoTT. Additionally, we presuppose a basic understanding of category theory and do not formally introduce its foundational concepts.

The formalizations are carried out in Agda, which is a dependently typed functional programming language and proof assistant. It enforces features such as strict type checking, termination checking, positivity checking and so on. We also depends on Agda standard library *agda-std* and *cubical* and borrow their notations for this report.

We conduct theoretical research within HoTT, such as interpreting contain-

ers 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 intuition, 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 container and container extension functor are defined in Cubical Agda 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 fields `isSetS` and `isSetP`. Meanwhile we focus on the constructions and often leave `F-id` and `F-seq` implicit.

Chapter 2

Conducted Research

In this section, we begin by reviewing related literature. We then introduce a categorical semantics of types, containers, W and M types. To facilitate later definition and result 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.

2.1 Literature Review

TODO

2.2 Types as Algebras

2.2.1 Inductive Types as Initial Algebras

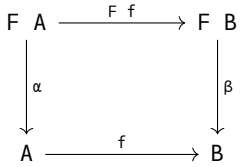
Natural Number

Natural number `ℕ = {0, 1, 2, 3, ...}` can be defined inductively:

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

Categories of Algebras

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. We write this as

```
ℕ ≅ μ X. ⊤ ⊎ X
```

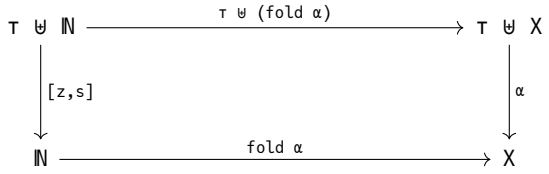
To prove this, we first massage its constructors into an equivalent function form:

```
[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 function:

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

such that the following diagram commutes:



2.2.2 Coinductive Types as Terminal Coalgebras

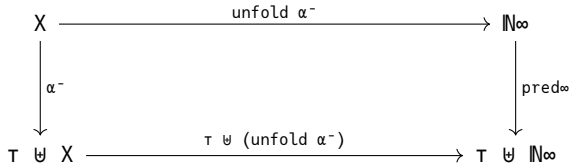
One of the greatest power of category theory is that the opposite version of a theorem can always be derived for free. We dualize above diagram, defining

conatural number as coinductive type and the inverse of folding function, the unfolding:

Conatural Number

```
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:



Therefore conatural number is the terminal coalgebra of the same signature functor.

$$\mathbb{N}\infty \cong \nu\ X.\ \top \uplus X$$

2.3 Containers

2.3.1 Strict Positivity

Containers[1] 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
```

The type `Weird` is not strictly positive, as itself appears on the left-side of `→` in `foo`. Violating strict positivity can lead to issues such as non-normalizability, non-termination, inconsistency and so on. In this weird case, it becomes possible to construct a empty type term, which contradicts to the empty definition:

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

empty : ⊥
empty = ¬weird (foo ¬weird)
```

2.3.2 Syntax and Semantics

An unary container is given by a type of shapes `S` and a type family indexed by `S`,called positions `P`:

```
record Cont : Type₁ where
  constructor _◃_
  field
    S : Type
    P : S → Type
```

Containers could be used to describe generic type (or generic class). For example, the container form of list:

```
ListC : Cont
ListC = ℕ ◃ Fin
```

The morphisms of unary containers are defined as a function which maps shapes and a family of (backward) functions that map positions indexed by `S`.

```
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 `idContHom` and compositions `_∘ContHom_` exist.

```
idContHom : ContHom SP SP
idContHom = id ◃ λ s → id

_∘ContHom_ : ContHom TQ CV → ContHom SP TQ → ContHom SP CV
(f ◃ g) ∘ContHom (h ◃ k) = (f ∘ h) ◃ λ s → k s ∘ g (h s)
```

Therefore, the containers form a category **Cont**.

```
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
```

Extension Functor

An unary container gives rise to a endofunctor of `Set` along the container extension functor ⟦_⟧. We explicitly distinguish mapping objects 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, naturally, the morphisms of containers 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
```

We show that the extension functor is fully faithful by constructing a bijection between `ContHom (S ◃ P) (T ◃ Q)` and `NatTrans ⟦ S ◃ P ⟧ ⟦ T ◃ Q ⟧`

$$
\begin{aligned}
& S \triangleleft P \to_{Cont} T \triangleleft Q \\
&= \sum_{f:S\to T} \prod_{s:S} Q\ (f\ s) \to P\ s && \text{definition of } \to_{Cont} \\
&\cong \prod_{s:S} \sum_{t:T} Q\ t \to P\ s && \text{type theoretical choice} \\
&= \prod_{s:S} [\![T \triangleleft Q]\!](P\ s) && \text{definition of } [\![\_]\!] \\
&\cong \prod_{s:S} \int_{X:Set} ((P\ s \to X) \to [\![T \triangleleft Q]\!]X) && \text{covariant Yoneda lemma} \\
&\cong \int_{X:Set} \prod_{s:S} ((P\ s \to X) \to [\![T \triangleleft Q]\!]X) && \text{commutative of } \textstyle\int \text{ and } \prod \\
&\cong \int_{X:Set} (\sum_{s:S} (P\ s \to X) \to [\![T \triangleleft Q]\!]X) && \text{uncurry} \\
&= \int_{X:Set} [\![S \triangleleft P]\!]X \to [\![T \triangleleft Q]\!]X && \text{definition of } [\![\_]\!]
\end{aligned}
$$

2.3.3 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:

```
⊤C : Cont
⊤C = ⊤ ◃ const ⊥

⊥C : Cont
⊥C = ⊥ ◃ ⊥-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. Both multiplication and addition are commutative, associative, and each is left- and right-annihilated by its corresponding unit. We use `⊥`, `⊤`, `×` and `⊎` as they correspond precisely to the initial object, terminal object, product and coproduct within the category of containers. In fact, the category of containers has all finite products and coproducts. We give the definitions and show that they are both preserved by the extension functor.

Products

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

$$
\begin{aligned}
&\prod_{i:I} (\llbracket S_i \triangleleft P_i \rrbracket X) \\
&= \prod_{i:I} \sum_{s:S_i} (P_i\ s \to X) && \text{definition of } \llbracket \_ \rrbracket \\
&\cong \sum_{f:\prod_{i:I} S_i} \prod_{i:I} (P_i\ (f\ i) \to X) && \text{type theoretical choice} \\
&\cong \sum_{f:\prod_{i:I} S_i} \left(\sum_{i:I} P_i\ (f\ i) \to X \right) && \text{uncurry} \\
&= \left\llbracket f : \prod_{i:I} S_i \triangleleft \sum_{i:I} P_i\ (f\ i) \right\rrbracket X && \text{definition of } \llbracket \_ \rrbracket
\end{aligned}
$$

Coproducts

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

$$
\begin{aligned}
&\sum_{i:I} (\llbracket S_i \triangleleft P_i \rrbracket X) \\
&= \sum_{i:I} \sum_{s:S_i} (P_i\ s \to X) && \text{definition of } \llbracket \_ \rrbracket \\
&\cong \sum_{(i,s_i):\sum_{i:I} S_i} (P_i\ s_i \to X) && \text{associative of } \textstyle\sum \\
&= \left\llbracket (i, s_i) : \sum_{i:I} S_i \triangleleft P_i\ s_i \right\rrbracket X && \text{definition of } \llbracket \_ \rrbracket
\end{aligned}
$$

Composition

We can also define composition of unary containers.

```
_∘C_ : Cont → Cont → Cont
(S ◃ P) ∘C (T ◃ Q) = (Σ[ s ∈ S ] (P s → T)) ◃ λ (s , f) → Σ[ p ∈ P s ] Q (f p)
```

$$
\begin{aligned}
&\llbracket S \triangleleft P \rrbracket (\llbracket T \triangleleft Q \rrbracket X) \\
&= \sum_{s:S} \left(P\ s \to \sum_{t:T} (Q\ t \to X) \right) && \text{definition of } \llbracket \_ \rrbracket \\
&\cong \sum_{s:S} \sum_{f:P\ s \to T} \prod_{p:P\ s} (Q\ (f\ p) \to X) && \text{type theoretical choice} \\
&\cong \sum_{(s,f):\sum_{s:S} (P\ s \to T)} \prod_{p:P\ s} (Q\ (f\ p) \to X) && \text{associative of } \textstyle\sum \\
&\cong \sum_{(s,f):\sum_{s:S} (P\ s \to T)} \left(\sum_{p:P\ s} Q\ (f\ p) \to X \right) && \text{uncurry} \\
&= \left\llbracket \sum_{(s,f):\sum_{s:S} (P\ s \to T)} \triangleleft \sum_{p:P\ s} Q\ (f\ p) \right\rrbracket X && \text{definition of } \llbracket \_ \rrbracket
\end{aligned}
$$

2.3.4 2-Categorical Structures

Cont has 2-categorical structures as the extension functor leads to the functor category. The vertical composition is just `_∘ContHom_`. We define the horizontal

compositions (tensor products) of morphisms, which also corresponds to the functoriality of `_∘C_`

Horizontal Compositions

```
_⊗ContHom_ : ContHom SP TQ → ContHom SP' TQ' → ContHom (SP ∘C SP') (TQ ∘C TQ')
(f ◃ g) ⊗ContHom (f' ◃ g')
  = (λ{ (s , k) → f s , f' ∘ k ∘ g s })
  ◃ λ{ (s , k) (q' , h) → g s q' , g' (k (g s q')) h }
```

such that the identity and compositions (interchanged law) are preserved. The proof is immediate by checking cases.

TODO : I need a diagram here

2.3.5 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
```

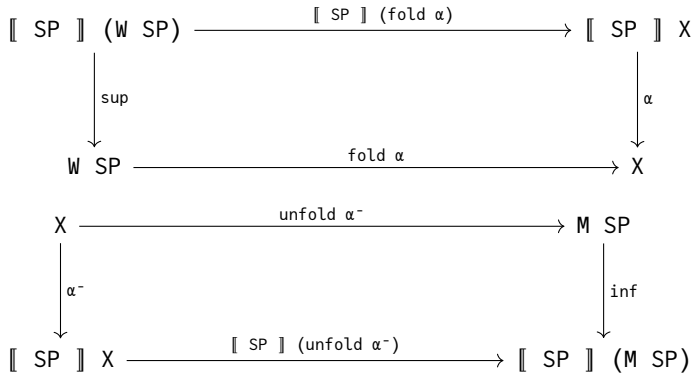
This definition implies that any inductive type can be characterized by a container algebra. Indeed, we can retrieve an equivalent definition of natural number through the maybe container.

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

ℕ' : Type
ℕ' = 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∞' : Type
ℕ∞' = M ⊤⊎Cont
```

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



2.4 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.

2.4.1 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 $\mathbb{C}$ 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 ]
```

2.4.2 Adding Functions

The CwF on its own typically doesn't provide anything about types. We now add function types 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 > ]
```

2.5 Normalization by Hereditary Substitutions

We introduce the hereditary substitutions technique for λ-calculus normalization. To build the syntax of λ-calculus, we first define a simple syntax of types and contexts. We use `A B C` to range over types and `Γ Δ Θ` to range over contexts.

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

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

De Bruijn Indices

We adopt the De Bruijn indices to capture variables. The idea is to replace the name of variable in context with number, with the innermost (rightmost) variable gets index 0 and outermost (leftmost) variable gets the biggest index. For example `λx.λy.x` in de bruijn form is `λ.λ.1`. We use `x y` to range over variables.

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

λ-Terms

The λ-terms follows the normal convention, which is either a variable, or a λ-abstraction, or a term application.

```
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
```

Normal Forms

We also need to define the syntax of normal forms. Normalized 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. The `Nf`, `Ne` and `Sp` are mutually defined in Agda:

```
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
```

In TODO, it builds a canonical normalizer for λ-calculi - `nf : Tm Γ A → Nf Γ A`. We shall not go through the full normalization process in details, but just illustrating the key components for each case.

A variable is normalized by performing η-expansion to the fullest extent possible, defined by `nvar`. For example, a variable with type `(* ⇒ *) ⇒ * ⇒ *`:

$$\mathtt{H} \to \lambda\mathtt{F.H\ F} \to \lambda\mathtt{X.}\lambda\mathtt{F.H\ F\ X}$$

```
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)))
```

The application case is more involved, where it is the actual hereditary substitutions happening. The substitutions under normal forms are recursively

defined together with substitutions under neutral terms and spines. Finally, the napp launches the whole process by substitutes the top variable of first term by the second term.

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

Finally, we obtain the normalization function that β-reduces and η-expands simply typed terms.

```
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)
```

In TODO, it shows the completeness and soundness of the normalizer. But we omit the proof for simplicity purpose.

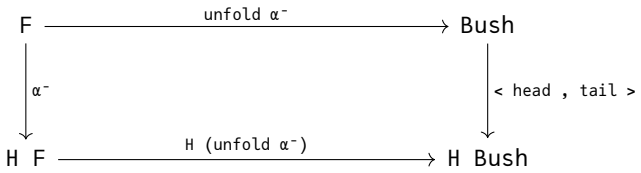
2.6 Questions

2.6.1 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 "higher" terminal coalgebra of a "higher" endofunctor:

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



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.