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 27, 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 $(Type \to Type) \to Type \to Type$. 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 λ-calculus.

Contents

Chapter 1

Introduction

1.1 Background and Objectives

Martin-Löf Type Theory (MLTT)[12][11] provides a constructive foundation for mathematics and programming. The essence of type theory is to represent propositions as types and their proofs as programs[5]. Homotopy Type Theory (HoTT)[14], 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 study the concepts of containers[1][2], 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 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 monad transformers[10] are not $(Type \to Type)$ but $((Type \to Type) \to Type \to Type)$.

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

Despite the higher kinds, $MaybeT$ could still be viewed as type which is specified by shapes and positions of input parameters. Accordingly, our work aims to generalize the containers to a higher-order sense, to provide the semantics for arbitrary higher-order inductive types. More specifically, we wish to:

- define higher functorialities for higher data types
- provide a syntax for higher containers
- explore the categorical and algebraic properties of higher containers
- interpret higher containers to higher functors
- 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[4], *The Tao of Types*[3], *Homotopy Type Theory - Univalent Foundations of Mathematics*[14]. I deepened related background knowledge and developed Agda programming skills by formalizing 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 also had many opportunities 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. I also attended the *TYPES 2025* in Glasgow, which was 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 those foundational concepts.

The formalizations are carried out in Agda[13], 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 libraries *agda-std* and *cubical.* Therefore, for the readability, we employ both Agda code and some standard mathematical notation throughout this report.

The theoretical research is conducted within HoTT, such as interpreting containers as endofunctors on h-level sets. For the sake of formalization, 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 settings. For example, containers and container extension functors 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))
  }
```

In the plain Agda definition, there would be no fields `isSetS` and `isSetP`. Tto avoid confusion, the primitive Agda `Set` is renamed to `Type` to represent small types. Accordingly, bigger types are simply `Type₁`, `Type₂`, etc. We also assume function extensionality and minimize the use of universe levels.

Chapter 2

Conducted Research

In this section, we begin by reviewing related literatures. We then introduce related research topics including the categorical semantics of types, containers, W and M types, and categories with families. Finally, we outline the current challenges in this research.

2.1 Literature Review

Denotational Semantics of Inductive Types

From a categorical point of view, inductive types are understood as the initial algebras of functors, which the result was originally carried out in (Goguen, Thatcher, Wagner and Wright 1977)[8]. Later, the W-type was explored by (Dybjer 1997)[7] as an encoding of strictly positive inductive types . It has been proved that every strictly positive endofunctor on the category of sets generated by Martin-Löf's extensional type theory has an initial algebra.

Containers

(Abbott, Altenkirch and Ghani 2003)[1] presented the categories of containers and showed some nice closure properties. This work could be used to redefine W-type and therefore serve as an alternative semantics of strictly positive types. Later, in (Abbott, Altenkirch and Ghani 2005)[2], the n-ary containers were introduced to capture data types with arguments such as lists and trees. And in (Altenkirch and Ghani 2015), the indexed containers were studied to cover inductive type families such as vectors. They also showed that indexed W-type can be converted into normal W-type. Some related formalization has been done in Cubical Agda in (Damato, Altenkirch and Ljungström 2024)[6].

This work

The object of this project is to review this categorical semantics of higher-order data types. We formalize the syntax and semantics in Agda. We also "containerize" the definition to provide the semantics higher-order strict positivity.

2.2 Categorical Semantics of Types

In the type theory formalization, inductive types are primitive structures specified by a set of inference rules, including type formation rules, introduction rules, induction rules and elimination rules[15]. For instance, in Agda the type of natural numbers is define as:

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

Categorical semantics provides a deeper, structural interpretation of this idea by modeling inductive types as the initial algebras of a functor.

2.2.1 Inductive Types as Initial Algebras

Categories of Algebras

Given an endofunctor $F : \mathbf{Set} \to \mathbf{Set}$, an algebra is defined as a carrier type $A : Set$ and an evaluation function $\alpha : F(A) \to A$. The morphisms between algebras (A, α) and (B, β) are given by a function $f : A \to B$ such that the following diagram commutes:

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

ℕ (with its constructors) is the initial algebra of the maybe functor. We write this as $\mathbb{N} \cong \mu X.1 + X$. To prove this result, we first massage its constructors into a 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:

$$\begin{tikzcd}
1+\mathbb{N} \arrow[r, "1+fold(\alpha)"] \arrow[d, "{[z,s]}"] & 1+X \arrow[d, "\alpha"] \\
\mathbb{N} \arrow[r, "fold(\alpha)"] & X
\end{tikzcd}$$

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. The dual notion of natural number is call the conatural number. By duality, conatural number is the terminal coalgebra of $Maybe$ functor, we write this as $\mathbb{N}\infty \cong \nu X.1+X$. In Agda, conatural number is defined as a coinductive type:

Conatural Number

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

To complete the diagram, we also define the inverse of folding, which is called unfolding:

```
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:

$$\begin{tikzcd}
X \arrow[r, "unfold(\alpha^-)"] \arrow[d, "\alpha^-"] & \mathbb{N}\infty \arrow[d, "pred\infty"] \\
1+X \arrow[r, "1+unfold(\alpha^-)"] & 1+\mathbb{N}\infty
\end{tikzcd}$$

2.3 Containers

2.3.1 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
```

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
```

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

```
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.

```
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, they form a category **CONT**.

Extension Functor

An unary container gives rise to a functor **Set** → **Set** along the container extension functor ⟦_⟧. We denote the mapping objects of a functor with no subscript or $_0$ as it maps 0-cell, and mapping morphisms part with $_1$ as it maps 1-cell.

```
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 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
```

We show that the extension functor is fully faithful by constructing a bijection between $S \triangleleft P \to T \triangleleft Q$ and $[\![S \triangleleft P]\!] \to [\![T \triangleleft Q]\!]$

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

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
```

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

Compositions

We can also define compositions of unary containers. This composition should reflect the process of consecutively applying container functors.

```
_∘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)
```

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

2.3.4 W and M

We now present the general form of inductive types, which is the W-type. The standard notion in the HoTT book is given by $A : \mathit{Type}$ and $B : A \to \mathit{Type}$, which is written as $W_{a:A}B(a)$. It is equipped with a single constructor $sup : \prod_{a:A}(B(a) \to W_{x:A}B(x)) \to W_{x:A}B(x)$. Notice that this definition is equivalent to the following container definition:

```
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 functor 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:

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

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

As a conclusion, W and M are essentially $\mu$and ν:

$$W(S \triangleleft P) \cong \mu X.[\![S \triangleleft P]\!]X$$
$$M(S \triangleleft P) \cong \nu X.[\![S \triangleleft P]\!]X$$

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 General Definition

There are many equivalent ways to define CwFs, we only present one definition which is suitable for our formalization. A CwF is given by:

- A category **C**. Its objects is a set of contexts, denoted by Γ, Δ, etc, Its morphisms is a set of context substitutions, denoted by γ, δ, etc. We call the objects type *Con* and morphisms type *Tms* (or *Subst*).

  Categorical rules for **C**:

$$id_\Gamma \circ \gamma = \gamma$$
$$\gamma \circ id_\Gamma = \gamma$$
$$(\gamma \circ \delta) \circ \theta = \gamma \circ (\delta \circ \theta)$$

- A terminal object $\bullet : Con$ which represents the empty context, with the terminal map $\epsilon_\Gamma : Tms\ \Gamma\ \bullet$ which represents the empty substitution.

  Universal property of $\bullet$:

$$\epsilon_\Gamma \circ \gamma = \epsilon_\Gamma$$
$$id_\bullet = \epsilon_\bullet$$

- A presheaf $Ty : \mathbf{C}^{op} \to \mathbf{Set}$ which represents a set of types of a given context Γ. We use A, B, etc. to denote type. For each $\gamma : Tms(\Delta, \Gamma)$, we also define a special syntax $\_[\gamma] : Ty(\Gamma) \to Ty(\Delta)$ for the mapping morphisms part of Ty.

  Functoriality of Ty:

$$
\begin{aligned}
A[id_\Gamma] &= A \\
A[\gamma \circ \delta] &= A[\gamma][\delta]
\end{aligned}
$$

- A presheaf $Tm : (\int_{\mathbf{C}})^{op} \to \mathbf{Set}$ which represents a set of terms of a given type $Ty(\Gamma)$. We use a, b, etc. to denote terms. For each $\gamma : Tms(\Delta, \Gamma)$ and $A : Ty(\Gamma)$, we also define a special syntax $\_[\gamma] : Tm(\Gamma, A) \to Tm(\Delta, A[\gamma])$ for the mapping morphisms part of Tm.

  Functoriality of Ty:

$$
\begin{aligned}
a[id_\Gamma] &= a \\
a[\gamma \circ \delta] &= a[\gamma][\delta]
\end{aligned}
$$

- A comprehension operation, which represents the context extensions. For each $\Gamma : Con$ and $A : Ty(\Gamma)$, there is an extended context $\Gamma \triangleright A$. For each $\gamma : Tms(\Delta, \Gamma)$ and $a : Tm(\Delta, A[\gamma])$, there is an extended substitution $(\gamma, a) : Tms(\Delta, \Gamma \triangleright A)$. For each $\gamma : Tms(\Delta, \Gamma \triangleright A)$, there is a projection from γ to $Tms(\Delta, \Gamma)$ that forgets the top variable, denoted by π_1; and a projection from γ to an unique term $a : Tm(\Delta, A[\pi_1 \gamma])$ which essentially is the variable being forgotten, denoted by π_2.

  The comprehension also comes with an universal property that, for every $\gamma : Tms(\Delta, \Gamma)$ and $a : Tm(\Gamma, A[\gamma])$, there exists a unique substitution $(\gamma, a) : Tms(\Delta, \Gamma \triangleright A)$, such that:

$$\begin{aligned}\pi_1\ (\gamma, a) &= \gamma \\ \pi_2\ (\gamma, a) &= a \\ (\pi_1\ \gamma, \pi_2\ \gamma) &= \gamma \\ (\delta, a) \circ \gamma &= (\delta \circ \gamma, a)\end{aligned}$$

2.4.2 Adding Functions

The CwF on its own typically does not say anything about types. We can add dependent function types to the framework separately.

For each $\Gamma : Con$, $A : Ty(\Gamma)$ and $B : Ty(\Gamma \triangleright A)$, there exist a dependent function type $\Pi(A, B) : Ty(\Gamma)$. For each $b : Tm(\Gamma \triangleright A, B)$, there is a function $lam(b) : Tm(\Gamma, \Pi(A, B))$. Inversely, for each $f : Tm(\Gamma, \Pi(A, B))$, there is a term $app(f) : Tm(\Gamma \triangleright A, B)$.

Rules for *lam* and *app*, which are essentially β-rule and η-rule:

$$\begin{aligned}app(lam(a)) &= a \\ lam(app(f)) &= f\end{aligned}$$

In order to define the substitutions under Π, *lam* and *app*, we need to first construct context substitution lifting operation: for each $\gamma : Tms(\Delta, \Gamma)$, it can be lifted to $\gamma \uparrow : Tms(\Delta \triangleright A[\gamma], \Gamma \triangleright A)$ by arbitrary $A : Ty(\Gamma)$.

Rules for substitution lifting and functions under substitutions.

$$\begin{aligned}\gamma \uparrow &= \gamma \circ (\pi_1\ id), \pi_2\ id \\ (\Pi(A, B))[\gamma] &= \Pi(A[\gamma], B[\gamma \uparrow]) \\ (lam(a))[\gamma] &= lam(t[\gamma \uparrow]) \\ (app(f))[\gamma \uparrow] &= app(f[\gamma])\end{aligned}$$

2.4.3 Simply Typed Categories with Families

For the purposes of our research, we are primarily interested in a weaker variant of CwF, namely the simply typed CwF (*SCwF*). The essence of a SCwF is to capture precisely the model of the simply typed λ-calculus, where types are independent of context. Therefore, while context substitution is still required for terms, it is not needed for types.

The model can be derived by constraining Ty to be a constant presheaf. That is:

$$\begin{aligned} &Ty : \mathbf{C}^{op} \to \mathbf{Set} \\ &Ty(\Gamma) := STy \\ &Ty(\gamma) := id \end{aligned}$$

where STy should be a set of simple types.

2.5 Questions

2.5.1 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 is to lift the space from Set to $Set \to Set$. We can show $Bush$ on its own is the terminal coalgebra of an endofunctor H of the functor category **[Set,Set]**.

$$\begin{aligned} &H : (Set \to Set) \to Set \to Set \\ &H(F)(X) = X \times F(F(X)) \end{aligned}$$

such that the following diagram commutes:

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

Chapter 3

Research Outcomes

3.1 Higher-Order Functors

3.1.1 Higher-Order Functions

Before directly moving to higher-order functors and containers, we need to first figure out their data type semantics, which are higher-order functions. Higher-order (or -kinded) functions are just like type constructors in functional programming. We can classify them by their kind signatures. In addition, we should constrain $Type$ to Set for the strict definition.

- Set : Unit, Bool, Nat, ...
- $Set \to Set$: Maybe, List, ...
- $Set \to Set \to Set$: Either, Const ...
- $(Set \to Set) \to Set$: Fix, ...
- $(Set \to Set) \to Set \to Set$: MaybeT, Free, ...
- arbitrary kinds ...

3.1.2 Higher-Order Functorialities

These functions are typically required to satisfy certain functoriality properties. Accordingly, we inductively define a type family $isHFunc$, indexed by kinds and higher functions to indicate what kind of functorialities they need to satisfy.

In the base case Set, where functions are actually just sets, no additional requirements are imposed.

$$isHFunc(X) := ()$$

In the inductive case, a higher function H with type $A \to B$ has functorialities:

$$isHFunc(H) := \sum_{H' : \prod_{\{F:A\}} isHFunc(F) \to isHFunc(H(F))} \mathbf{HCat}(A) \to \mathbf{HCat}(B)$$

It is defined as a sigma type. The first component says for all higher functor F with type A, H' sends functorialities of F to functorialities of $H(F)$. The second component says (H, H') together is a functor between higher categories indexed by A and B.

We need to define higher categories **HCat**, again, as a type family over kinds. Given a type A, the objects are defined as:

$$|\mathbf{HCat}(A)| := \sum_{F:A} isHFunc(F)$$

Therefore in the base case Set, it is equivalent to the category of sets - **Set**

$$|\mathbf{HCat}(*)| := |\mathbf{Set}|$$
$$\mathbf{HCat}(*)(X, Y) := X \to Y$$

The inductive case $A \to B$ is much more involved. It is necessary to first establish some variable naming conventions for this definition. It is also a summary for higher functors.

- H is function that sends function $F : A$ to $H(F) : B$.
- H' is function that sends $isHFunc(F)$ to $isHFunc(H(F))$
- H'' is the proof of (H, H') being a functor at current level

$$\begin{aligned}
&|\mathbf{HCat}(A \to B)| \\
&:= \sum_{H:A\to B} isHFunc(H) \\
&:= \sum_{H:A\to B} \sum_{H' : \prod_{\{F:A\}} isHFunc(F) \to isHFunc(H(F))} (\mathbf{HCat}(A) \to \mathbf{HCat}(B))
\end{aligned}$$

$$\begin{aligned}&\mathbf{HCat}(A \to B)((H, isHFuncH), (J, isHFuncJ))\\ &:= \mathbf{HCat}(A \to B)((H, H', H''), (J, J', J''))\\ &:= \int_{(F, isHFuncF):|\mathbf{HCat}(A)|} (H(F), H'(isHFuncF), H'') \to (J(F), J'(isHFuncF), J'')\end{aligned}$$

Finally, the definition of a higher functor is straightforward:

$$HFunc(A) := \sum_{F:A} isHFunc(F)$$

We can show many types mentioned above are indeed higher-order functors. See example.

Functor Categories?

It is important to emphasize that **HCat** is not equivalent to the iterated functor categories over **Set**. By iterated functor categories, we mean the closure of **Set** under formation of functor categories. We can as well define it as a family of categories over kinds.

$$\begin{aligned}\mathbf{FuncCat}(\,*\,) &:= \mathbf{Set}\\ \mathbf{FuncCat}(A \to B) &:= [\mathbf{FuncCat}(A),\mathbf{FuncCat}(B)]\end{aligned}$$

In fact, **HCat** and **FuncCat** are equivalent under $Set \to Set$, but when it goes higher, their behavior begins to diverge. For example, we have:

$$\frac{HFunc((Set \to Set) \to Set \to Set)}{(Set \to Set) \to Set \to Set}$$

by simply taking the first component of the Σ. However, it is not true in **FuncCat**. The best we can get is:

$$\frac{|\mathbf{FuncCat}((Set \to Set) \to Set \to Set)|}{|\mathbf{FuncCat}(Set \to Set)| \to Set \to Set}$$

3.2 Higher-Order Containers

The idea of higher containers is to provide a semantics of strict positivity for the higher data types. That is, we should be able to specify a higher data types by describing the shapes and positions of arguments, and derive higher functorialities for free.

The definition of higher containers introduced in this section is strongly influenced by (Keller and Altenkirch)[9]. In that work, the authors developed a normalization function for the simply typed λ-calculus base on a "hereditary substitutions" technique. They explicitly specify a syntax for λ-terms and their normal forms, and prove that the normalization function precisely characterizes $\beta\eta$-equivalence.

We do not present the results from this work in order to avoid unnecessary duplication. Instead, we directly provide our formalization, as it can be regarded as an extension to the previous work.

3.2.1 Syntax

To establish a syntax of higher containers, we begin by observing a second-order data type H.

$$H : (Set \to Set) \to Set \to Set$$
$$H(F)(X) = X \times F(F(X))$$

Comparing to the first-order container, H is also a coproduct of sub-terms, so we need to keep the shape. The difference in the second-order case is that, we now need to keep track of the positions for both input variables X and F. Notably, the sub-term $F(F(X))$ is defined as nested term applications. It implies that, we also need to add recursive structures for each occurrence of term that is not fully saturated.

It leads to the syntax for higher containers. They are simply typed λ-terms closed under products and coproducts. To facilitate the definition, we first build a syntax for types and contexts:

Type and Context

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

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

Ty is closed under a base type and function type. We use A, B, etc. to denote a type. Con is a list of Ty. We use Γ, Δ, etc. to denote a context.

De Bruijn Indices

The De Bruijn indices are adopted to capture variables. The idea is to replace the name of variable in context with number, with the innermost (rightmost) variable is indexed by 0 and outermost (leftmost) variable is indexed by the biggest number. For example $\lambda x.\lambda y.x$ in de bruijn form is $\lambda.\lambda.1$. We use x, y, etc. to range over variables.

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

Terms

As described in previous section, higher containers are simply typed λ-terms that closed under arbitrary products and coproducts. Consequently, we can construct a naive syntax $Tm : Con \to Ty \to Set$:

```
data Tm : Con → Ty → Type₁ where
  var : Var Γ A → Tm Γ A
  lam : Tm (Γ ▹ A) B → Tm Γ (A ⇒ B)
  _$_ : Tm Γ (A ⇒ B) → Tm Γ A → Tm Γ B
  Πtm : (I : Set) → (I → Tm Γ A) → Tm Γ A
  Σtm : (I : Set) → (I → Tm Γ A) → Tm Γ A
```

Then H can be defined as a term, which is a binary product of X and F applied to F applied to X. ⚙

While this Tm definition is valid on its own, it has two disadvantages. The first is that the representation of a term is not unique. For example, the following F can have more than one equivalent forms.

$$F : \mathit{Set} \to \mathit{Set}$$
$$F(X) = X \times (X + 1) \cong X \times X + X$$

The second thing is that it is not "container" enough, as there are not shapes, positions, etc.

Normal Forms

We therefore present an alternative syntax for higher containers, which captures the unique form of simply typed λ-term after normalization. To define a syntax for normal forms, we also need to mutually define it with neutral terms and spines:

```
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 : 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
```

A normal form is either:

- A λ-abstraction *lam*.
- A neutral term *ne*: It is the core container definition, which is a coproduct (specified by shapes S) of product (specified by P) of variables applied to their spines (specified by R). A spine of a variable is a list of normal forms as the arguments applied to that variable, according to its arity.

We would use t, u, etc. to denote normal forms; spr, tql, etc. to denote neutral terms; and ts, us, etc. to denote the spines.

We can reconstruct H through this normal form definition. We first obtain the context $\Gamma = \bullet \triangleright F \triangleright A$ by lam twice. From the outermost layer, Hnf is a container with two shapes, with the first shape has a position of X and the second shape has a position of F. But F is not saturated yet, which requires to construct another normal form $FXnf$ as a singleton spine applied to F. $FXnf$ is a container with one shape and one position of F, which requires to define another normal form Xnf to apply to it as parameter. Finally, Xnf is a container with one shape and one position of X. ⚙

Higher Containers

We shall define higher containers as closed (with empty context) normal forms. We would therefore use normal forms and higher containers interchangeably for the formalization and the rest of report.

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

3.2.2 Semantics

We wish to interpret higher containers to higher functors. But we first interpret higher containers to higher functions, which is the first component of higher functors. We define the functional semantics of types, contexts and variables:

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

⟦_⟧c : Con → Type₁
⟦ • ⟧c = Lift (lsuc lzero) ⊤
⟦ 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
```

Ty represents the signature of higher functions. Con corresponds to a list of such higher functions. $Var(\Gamma, A)$ is used to retrieve a higher function of type A from context Γ.

We mutually defining the functional semantics of normal forms, neutral terms and spines:

```
⟦_⟧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))
```

- Given a normal form $t : Nf(\Gamma, A)$ and a current state of context, it pushes all variables appear on the domain of A into the context until there is only $*$ left in the codomain, which means it is ready to define a neutral term.
- The purpose of a spine $ts : Sp(\Gamma, A, B)$ is to map a higher function of type A to B. This process can be defined by recursively applying each normal form from the spine.
- The semantics of a neutral term $spr : Ne(\Gamma, *)$, as we proposed before, is the coproduct of products of variables applied to its spine.

Again, higher containers are just closed normal forms.

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

3.2.3 Categorical Structures

There is a category **HCONT**(A) for each type A. We define the morphisms between normal forms:

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

The morphism of normal forms is just to push variables into context until it is ready to define morphism of neutral terms. The morphisms of neutral terms $S \triangleleft P \triangleleft R$ and $T \triangleleft Q \triangleleft L$ is given by:

- a function that maps the shapes
- for each variable and shape, a function that maps the positions in the inverse direction
- for each variable, shape and position, a morphism of spines

The first two components f and g of the morphisms are the same as morphisms of first-order container. However, for the third component h, it remains unclear whether the morphisms of spines are well-defined or even meaningful.

But still we can construct some examples. ⚙

Algebraic Structures

We wish that the categories of higher containers to have many closure properties as the first-order case. Indeed, they are closed under arbitrary products and coproducts. We need to first build some auxiliary functions.

```
app : Nf Γ (A ⇒ B) → Nf (Γ ▹ A) B
app (lam t) = t

en : Nf Γ * → Ne Γ *
en (ne spr) = spr
```

The *app* is the inverse of *lam*, and *en* is the inverse of *ne*.

Products

```
Πnf : (I : Type) → (I → Nf Γ A) → Nf Γ A
Πnf {Γ} {A ⇒ B} I t⃗  = lam (Πnf I (app ∘ t⃗ ))
Πnf {Γ} {∗} I t⃗  = ne (S ◃ P ◃ R)
  where
  S : Type
  S = (i : I) → en (t⃗ i) .Ne.S

  P : Var Γ A → S → Type
  P x f = Σ[ i ∈ I ] en (t⃗ i) .Ne.P x (f i)

  R : (x : Var Γ A) (s : S) → P x s → Sp Γ A ∗
  R x f (i , p) = en (t⃗ i) .Ne.R x (f i) p
```

Coproducts

```
Σnf : (I : Type) → (I → Nf Γ A) → Nf Γ A
Σnf {Γ} {A ⇒ B} I t⃗  = lam (Σnf I (app ∘ t⃗ ))
Σnf {Γ} {∗} I t⃗  = ne (S ◃ P ◃ R)
  where
  S : Type
  S = Σ[ i ∈ I ] en (t⃗ i) .Ne.S

  P : Var Γ A → S → Type
  P x (i , s) = en (t⃗ i) .Ne.P x s

  R : (x : Var Γ A) (s : S) → P x s → Sp Γ A ∗
  R x (i , s) p = en (t⃗ i) .Ne.R x s p
```

3.2.4 Normalization

We now construct a normalization function $nf : Tm(\Gamma, A) \to Nf(\Gamma, A)$ by applying hereditary substitutions to our definitions. It turns out that normalizers for λ-abstraction, products and coproducts have already been obtained. It is left to construct the normalization function for variables $nvar : Var(\Gamma, A) \to Nf(\Gamma, A)$ and for applications $napp : Nf(\Gamma, A \to B) \to Nf(\Gamma, A) \to Nf(\Gamma, B)$.

Weakening

Weakening is a standard construction in type theory. It represents the process of extending a context with additional variables. To formalize this, we first define how to obtain a new context by removing variables. Although in practice our constructions only require removing the top variable, it is still possible to define a general function that removes an arbitrary index.

Then we can weaken a variable by shifting its index:

```
wkv : (x : Var Γ A) → Var (Γ - x) B → Var Γ B
wkv vz y = vs y
wkv (vs x) vz = vz
wkv (vs x) (vs y) = vs (wkv x y)
```

Variable Equality

We establish an notion of heterogeneous equality of variables. Two variables are the equal if they are the same variables. Two variables x, y are different if there exist a variable x' such that $x = wkv\ y\ x'$. The *eq* function decides the equality of variables.

```
data EqV : Var Γ A → Var Γ B → Type where
  same : EqV x x
  diff : (x : Var Γ A) (y : Var (Γ - x) B) → EqV x (wkv x y)

eq : (x : Var Γ A) (y : Var Γ B) → EqV x y
eq vz vz = same
eq vz (vs y) = diff vz y
eq (vs x) vz = diff (vs x) vz
eq (vs x) (vs y) with eq x y
eq (vs x) (vs .x) | same = same
eq (vs x) (vs .(wkv x y')) | diff .x y' = diff (vs x) (vs y')
```

Normal Forms Weakening

We can also weaken normal forms, which in turn, mutually weaken neutral terms and spines:

```
wkNf : (x : Var Γ A) → Nf (Γ - x) B → Nf Γ B
wkNf x (lam t) = lam (wkNf (vs x) t)
wkNf x (ne spr) = ne (wkNe x spr)

wkNe : (x : Var Γ A) → Ne (Γ - x) B → Ne Γ B
wkNe {Γ} {A} {C} x (S ◃ P ◃ R) = S ◃ P' ◃ R'
  where
  P' : Var Γ B → S → Type
  P' y s with eq x y
  P' .x s | same = ⊥
  P' y s | diff .x y' = P y' s

  R' : (y : Var Γ B) (s : S) → P' y s → Sp Γ B C
  R' y s p with eq x y
  R' y s p | diff .x y' = wkSp x (R y' s p)

wkSp : (x : Var Γ A) → Sp (Γ - x) B C → Sp Γ B C
wkSp x ε = ε
wkSp x (t , ts) = wkNf x t , wkSp x ts
```

$wkNf$ and $wkSp$ are defined in a straightforward way, by recursively calling weakening to their respective sub-constructions. The case of $wkNe$ is more interesting: when weakening a neutral term, its shape remains unchanged, the newly added variable should have no position, and therefore no spine is assigned to it.

η-expansion

The normalization functions for variables and neutral terms corresponds to η-expansion. For instance a variable with type $(Type \to Type) \to Type \to Type$ should be converted to a normal form by fully applying η-expansion many times:

$$H \to_\eta \lambda F.H(F) \to_\eta \lambda X.\lambda F.H(F)(X)$$

We first build an auxiliary function that append a normal form at the end of spine:

```
appSp : Sp Γ A (B ⇒ C) → Nf Γ B → Sp Γ A C
appSp ε u = u , ε
appSp (t , ts) u = t , appSp ts u
```

Then, the normalizers for variables and neutral terms are mutually defined:

```
nvar : Var Γ A → Nf Γ A
nvar {Γ} {B} x = ne2nf (⊤ ◃ P ◃ R)
```

```
  where
  P : Var Γ A → ⊤ → Type
  P y tt with eq x y
  P .x tt | same = ⊤
  P y tt | diff .x y' = ⊥

  R : (y : Var Γ A) (s : ⊤) → P y s → Sp Γ A B
  R y tt p with eq x y
  R .x tt p | same = ε
  R y tt () | diff .x y'

ne2nf : Ne Γ A → Nf Γ A
ne2nf {Γ} {∗} spr = ne spr
ne2nf {Γ} {A ⇒ C} (S ◃ P ◃ R) = lam (ne2nf (S ◃ P' ◃ R'))
  where
  P' : Var (Γ ▹ A) B → S → Type
  P' vz s = ⊥
  P' (vs x) s = P x s

  R' : (x : Var (Γ ▹ A) B) (s : S) → P' x s → Sp (Γ ▹ A) B C
  R' vz s ()
  R' (vs x) s p = appSp (wkSp vz (R x s p)) (nvar vz)
```

The function *nvar* applies an empty spine to variable and then calls *ne*2*nf*. The function *ne*2*nf* repeats η-expansion until the type reduces to $*$.

β-reduction

The normalization function for applications corresponds to the β-reduction.

```
_[_:=_] : Nf Γ B → (x : Var Γ A) → Nf (Γ - x) A → Nf (Γ - x) B
lam t [ x := u ] = lam (t [ vs x := wkNf vz u ])
ne {Γ} (S ◃ P ◃ R) [ x := u ] = ne (S ◃ P' ◃ R')
  where
  P' : Var (Γ - x) A → S → Type
  P' y s = P (wkv x y) s

  R' : (y : Var (Γ - x) A) (s : S) → P' y s → Sp (Γ - x) A ∗
  R' y s p = R (wkv x y) s p < x := u >

_<_:=_> : Sp Γ B C → (x : Var Γ A) → Nf (Γ - x) A → Sp (Γ - x) B C
```

```
ε < x := u > = ε
(t , ts) < x := u > = (t [ x := u ]) , (ts < x := u >)

_◇_ : Nf Γ A → Sp Γ A B → Nf Γ B
t ◇ ε = t
t ◇ (u , us) = napp t u ◇ us

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

The function $t[x := u]$ substitutes variable x with normal form u inside a normal form t.The function $t\langle x := u\rangle$ substitutes variable x with normal form u inside a spine ts. The function $t \diamond u$ folds a spine us to the variable t by $napp$. Finally, $napp$ launches the β-reduction.

Normalization Function

Finally, putting everything together, we obtain our normalizer:

```
nf : Tm Γ A → Nf Γ A
nf (var x) = nvar x
nf (lam x) = lam (nf x)
nf (t $ u) = napp (nf t) (nf u)
nf (Πtm I t⃗ ) = Πnf I (nf ∘ t⃗ )
nf (Σtm I t⃗ ) = Σnf I (nf ∘ t⃗ )
```

3.2.5 Simply Typed Categories with Families

We show that the higher containers is a model of SCwF, where Ty is Ty, Con is Con, and Tm is Nf. We also need to define a structure to represent context substitutions. As Tms is a list of Tm, we define the $Nfs : Con \to Con \to Set$ as a list of Nf, denoted by γ, δ, etc.

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

Notice that, although both Nfs and Sp are defined as lists of Nf, they carry different type signatures and serve distinct purposes.

We need to define, for each $\gamma : \Delta \to \Gamma$, a context substitution $\_[\gamma] : Nf(\Gamma, A) \to Nf(\Delta, A)$. But before that, we need to build some auxiliary functions, as well as prove the existence of other constructions.

We perform the same weakening trick for Nfs, and therefore obtain a definition of context substitution lifting $\_\uparrow$:

We also define auxiliary functions *subv* which retrieve a normal from from the substitutions.

```
subv : Var Γ A → Nfs Δ Γ → Nf Δ A
subv vz (γ , t) = t
subv (vs x) (γ , t) = subv x γ
```

The Context Substitutions

Finally, we construct the context substitution by mutually define the substitution for normal forms and for spines:

```
_[_]nf : Nf Γ A → Nfs Δ Γ → Nf Δ A
lam t [ γ ]nf = lam (t [ γ ↑ ]nf)
ne {Γ} (S ◃ P ◃ R) [ γ ]nf = Σnf[ s ∈ S ]
  Πnf[ A ∈ Ty ] Πnf[ x ∈ Var Γ A ] Πnf[ p ∈ P x s ]
  (subv x γ) ◇ (R x s p [ γ ]sp)

_[_]sp : Sp Γ A B → Nfs Δ Γ → Sp Δ A B
ε [ γ ]sp = ε
(t , ts) [ γ ]sp = (t [ γ ]nf) , (ts [ γ ]sp)
```

In the definition, the *lam* case is trivial, which calls itself recursively with a lifted context substitution. The context substitution for a spine is just to call substitutions for each normal forms inside that spine.

However, it is not trivial to define the context substitution for neutral terms. The reason is that there is no context substitution for variables:

$$\cancel{\_[\_]v : Var(\Gamma, A) \to Nfs(\Delta, \Gamma) \to Var(\Delta, A)}$$

Instead, the best we can achieve is *subv* as defined before, where the target is a normal form after context substitution:

$$subv : Var(\Gamma, A) \to Nfs(\Delta, \Gamma) \to Nf(\Delta, A)$$

Therefore, for each occurrence of variable in the neutral term: we first convert it into a normal form by *subv*, then applied its spines (after context substitutions) by $\_ \diamond \_$, and finally use the products $\Pi nf$and coproducts $\Sigma nf$of normal forms to glue them into a single normal form.

A partial proof of this result has been formalized.

Chapter 4

Conclusion

In this section, we explicitly state the research questions, as well as the current progress to address the questions.

Questions

Containers provide a semantics for strictly positive functors. By applying the W-type construction, they can be used to represent strictly positive inductive types. However, this framework does not capture all inductive types. For instance, those whose signatures involve higher-order kinds.

To address this limitation, we aim to develop a notion of higher-order containers, thereby providing a categorical and semantic interpretation of such higher-order inductive types.

Progress

Up to this stage, we have obtained several promising results. We have already:

- built a notion of higher functorialities for higher types
- defined syntax categories for higher containers
- showed that higher containers are closed under products, coproducts and compositions
- partly proved that higher containers give rise to a SCwF.

Nevertheless, there remain many open tasks, which we will outline in detail in the following section.

Chapter 5

Future Work Plan

Full Formalization

There are many things left to do in our current research. The primary object at this stage is to finish the constructions and proofs as we promised before. As a summary, we wish to:

- construct the functorial meaning of higher containers
- fully interpret higher containers to higher functors
- define the semantics of morphisms of higher containers
- prove the universal properties of categorical objects
- finish the proofs of higher containers as a model of simply typed λ-calculus

Moreover, current formalization is carried out in plain Agda. We wish to strictly replicate the whole formalization with all h-level in Cubical Agda.

Soundness and Completeness

Another objective is to prove the completeness and soundness properties of the normalizer nf. To do that, we first define an embedding from $Nf(\Gamma, A)$ to $Tm(\Gamma, A)$:

Embedding

```
embNf : Nf Γ A → Tm Γ A
embNf (lam t) = lam (embNf t)
embNf (ne spr) = embNe spr

embNe : Ne Γ A → Tm Γ A
embNe (S ◃ P ◃ R) = Σtm[ s ∈ S ]
  Πtm[ A ∈ Ty ] Πtm[ x ∈ Var _ A ] Πtm[ p ∈ P x s ]
  embSp (R x s p) (var x)

embSp : Sp Γ A B → Tm Γ A → Tm Γ B
embSp ε u = u
embSp (t , ts) u = embSp ts (u $ embNf t)
```

Here, the completeness means the embedding of any normalized term is $\beta\eta$-equivalent to itself:

```
completeness : (t : Tm Γ A) → embNf (nf t) βη≡ t
```

where the $\beta\eta$-equivalence should be a relation of λ-terms which reflects the convertibility under β-equivalence and η-equivalence. We need to characterize this relation as well.

The soundness says, if two terms are convertible, then they should reduce to the same canonical normal forms:

```
soundness : (t u : Tm Γ A) → t βη≡ u → nf t ≡ nf u
```

Combining two results together, we wish to show that the "convertibility of terms is decidable".

Chapter 6

Appendix

6.1 Agda Code - Higher-Order Containers Examples

Morphisms

```
idNfHom : NfHom t t
idNfHom {t = ne spr} = ne (id ◃ (λ x s → id) ◃ λ x s q → ε)
idNfHom {t = lam t} = lam (idNfHom {t = t})

_∘nfHom_ : NfHom u w → NfHom t u → NfHom t w
_∘spHom_ : SpHom us ws → SpHom ts us → SpHom ts ws

lam f ∘nfHom lam g = lam (f ∘nfHom g)
ne (f ◃ g ◃ h) ∘nfHom ne (f' ◃ g' ◃ h') = ne (
  (f ∘ f')
  ◃ (λ x s → g' x s ∘ g x (f' s))
  ◃ λ x s q → (h x (f' s) q) ∘spHom h' x s (g x (f' s) q)
  )

ε ∘spHom ε = ε
ε ∘spHom (g , gs) = g , gs
(f , fs) ∘spHom ε = f , fs
(f , fs) ∘spHom (g , gs) = (f ∘nfHom g) , (fs ∘spHom gs)

⊤nf : Nf Γ A
⊤nf {Γ} {*} = ne (⊤ ◃ (λ{ x tt → ⊥ }) ◃ λ{ x tt () })
```

```
⊤nf {Γ} {A ⇒ B} = lam ⊤nf

⊥nf : Nf Γ A
⊥nf {Γ} {*} = ne (⊥ ◃ (λ x ()) ◃ (λ x ()))
⊥nf {Γ} {A ⇒ B} = lam ⊥nf

_×nf_ : Nf Γ A → Nf Γ A → Nf Γ A
lam t ×nf lam u = lam (t ×nf u)
_×nf_ {Γ} {B} (ne (S ◃ P ◃ R)) (ne (T ◃ Q ◃ L)) = ne (S' ◃ P' ◃ R')
  where
  S' : Set
  S' = S × T

  P' : Var Γ A → S' → Set
  P' x (s , t) = P x s ⊎ Q x t

  R' : (x : Var Γ A) (s : S') → P' x s → Sp Γ A B
  R' x (s , t) (inj₁ p) = R x s p
  R' x (s , t) (inj₂ q) = L x t q

_⊎nf_ : Nf Γ A → Nf Γ A → Nf Γ A
lam t ⊎nf lam u = lam (t ⊎nf u)
_⊎nf_ {Γ} {B} (ne (S ◃ P ◃ R)) (ne (T ◃ Q ◃ L)) = ne (S' ◃ P' ◃ R')
  where
  S' : Set
  S' = S ⊎ T

  P' : Var Γ A → S' → Set
  P' x (inj₁ s) = P x s
  P' x (inj₂ t) = Q x t

  R' : (x : Var Γ A) (s : S') → P' x s → Sp Γ A B
  R' x (inj₁ s) p = R x s p
  R' x (inj₂ t) q = L x t q

!nf : (t : Nf Γ A) → NfHom t ⊤nf
!nf (lam t) = lam (!nf t)
!nf (ne (S ◃ P ◃ R)) = ne ((λ _ → tt) ◃ (λ x s ()) ◃ λ x s ())

¿nf : (t : Nf Γ A) → NfHom ⊥nf t
¿nf (lam t) = lam (¿nf t)
¿nf (ne (S ◃ P ◃ R)) = ne ((λ ()) ◃ (λ x ()) ◃ λ x ())
```

```
π₁nf : (t u : Nf Γ A) → NfHom (t ×nf u) t
π₁nf (lam t) (lam u) = lam (π₁nf t u)
π₁nf {Γ} {B} (ne (S ◃ P ◃ R)) (ne (T ◃ Q ◃ L)) = ne (f ◃ g ◃ h)
  where
  f : S × T → S
  f (s , t) = s

  g : (x : Var Γ A) (st : S × T) → P x (f st) → P x (st .proj₁) ⊎ Q x (st .proj₂)
  g x (s , t) p = inj₁ p

  h : (x : Var Γ A) (st : S × T) (q : P x (f st)) → SpHom (R x (f st) q) (R x (f st) q)
  h x (s , t) q = ε

i₁nf : (t u : Nf Γ A) → NfHom t (t ⊎nf u)
i₁nf (lam t) (lam u) = lam (i₁nf t u)
i₁nf {Γ} (ne (S ◃ P ◃ R)) (ne (T ◃ Q ◃ L)) = ne (f ◃ g ◃ h)
  where
  f : S → S ⊎ T
  f s = inj₁ s

  g : (x : Var Γ A) (s : S) → P x s → P x s
  g x s p = p

  h : (x : Var Γ A) (s : S) (q : P x s) → SpHom (R x s (g x s q)) (R x s q)
  h x s q = ε

<_,_>nf : NfHom t u → NfHom t w → NfHom t (u ×nf w)
< lam tu , lam tv >nf = lam < tu , tv >nf
<_,_>nf {Γ} {B} (ne (f₁ ◃ g₁ ◃ h₁)) (ne (f₂ ◃ g₂ ◃ h₂)) = ne (ff ◃ gg ◃ hh)
  where
  ff : _
  ff = < f₁ , f₂ >

  gg : (x : Var Γ A) (s : _) → _
  gg x s (inj₁ q₁) = g₁ x s q₁
  gg x s (inj₂ q₂) = g₂ x s q₂

  hh : (x : Var Γ A) (s : _) (q : _) → _
  hh x s (inj₁ q₁) = h₁ x s q₁
  hh x s (inj₂ q₂) = h₂ x s q₂
```

```
[_,_]nf : NfHom t w → NfHom u w → NfHom (t ⊎nf u) w
[ lam tv , lam uv ]nf = lam ([ tv , uv ]nf)
[_,_]nf {Γ} {B} (ne (f₁ ◃ g₁ ◃ h₁)) (ne (f₂ ◃ g₂ ◃ h₂)) = ne (ff ◃ gg ◃ hh)
  where
  ff : _
  ff = [ f₁ , f₂ ]

  gg : (x : Var Γ A) (s : _) → _
  gg x (inj₁ s₁) q₁ = g₁ x s₁ q₁
  gg x (inj₂ s₂) q₂ = g₂ x s₂ q₂

  hh : (x : Var Γ A) (s : _) (q : _) → _
  hh x (inj₁ s₁) q₁ = h₁ x s₁ q₁
  hh x (inj₂ s₂) q₂ = h₂ x s₂ q₂
```

SCwF

```
idNfs : Nfs Γ Γ
idNfs {•} = ε
idNfs {Γ ▹ A} = idNfs ↑

_∘nfs_ : Nfs Δ Γ → Nfs Θ Δ → Nfs Θ Γ
ε ∘nfs γ = ε
(δ , t) ∘nfs γ = (δ ∘nfs γ) , (t [ γ ]nf)

π₁ : Nfs Δ (Γ ▹ A) → Nfs Δ Γ
π₁ (γ , t) = γ

π₂ : Nfs Δ (Γ ▹ A) → Nf Δ A
π₂ (γ , t) = t

wk : Nfs (Γ ▹ A) Γ
wk = π₁ idNfs

nvz : Nf (Γ ▹ A) A
nvz = π₂ idNfs

nvs : Nf Γ A → Nf (Γ ▹ B) A
nvs t = t [ wk ]nf

<_> : Nf Γ A → Nfs Γ (Γ ▹ A)
```

```
< t > = idNfs , t

π₁β : π₁ (γ , t) ≡ γ
π₁β = refl

π₂β : π₂ (γ , t) ≡ t
π₂β = refl

πη : (π₁ γ , π₂ γ) ≡ γ
πη {γ = γ , t} = refl

,∘ : (γ , t) ∘nfs δ ≡ (γ ∘nfs δ , t [ δ ]nf)
,∘ = refl
```

Example - *Htm*

```
Htm : Tm • ((∗ ⇒ ∗) ⇒ ∗ ⇒ ∗)
Htm = lam (lam (Πtm (⊤ ⊎ ⊤) (λ
  { (inj₁ tt) → var vz
  ; (inj₂ tt) → var (vs vz) $ ((var (vs vz)) $ var vz)
  })))
```

Example - *Hnf*

```
Hnf : Nf • ((∗ ⇒ ∗) ⇒ ∗ ⇒ ∗)
Hnf = lam (lam (ne (S ◃ P ◃ R)))
  where
  Γ₀ : Con
  Γ₀ = (• ▹ ∗ ⇒ ∗ ▹ ∗)

  S : Type
  S = ⊤ ⊎ ⊤

  P : Var Γ₀ A → S → Type
  P vz (inj₁ tt) = ⊤
  P vz (inj₂ tt) = ⊥
  P (vs vz) (inj₁ tt) = ⊥
  P (vs vz) (inj₂ tt) = ⊤
```

```
R : (x : Var Γ₀ A) (s : S) → P x s → Sp Γ₀ A ∗
R vz (inj₁ tt) tt = ε
R (vs vz) (inj₂ tt) tt = FX , ε
  where
  FX : Nf Γ₀ ∗
  FX = ne (FX-S ◃ FX-P ◃ FX-R)
    where
    FX-S : Type
    FX-S = ⊤

    FX-P : Var Γ₀ A → FX-S → Type
    FX-P vz tt = ⊥
    FX-P (vs vz) tt = ⊤

    FX-R : (x : Var Γ₀ A) (s : FX-S) → FX-P x s → Sp Γ₀ A ∗
    FX-R (vs vz) tt tt = X , ε
      where
      X : Nf Γ₀ ∗
      X = ne (X-S ◃ X-P ◃ X-R)
        where
        X-S : Type
        X-S = ⊤

        X-P : Var Γ₀ A → X-S → Type
        X-P vz tt = ⊤
        X-P (vs vz) tt = ⊥

        X-R : (x : Var Γ₀ A) (s : X-S) → X-P x s → Sp Γ₀ A ∗
        X-R vz tt tt = ε
        X-R (vs vz) tt ()
```

Bibliography

[1] Abbott, M., Altenkirch, T., and Ghani, N. Categories of containers. In *International Conference on Foundations of Software Science and Computation Structures* (2003), Springer, pp. 23–38.

[2] Abbott, M., Altenkirch, T., and Ghani, N. Containers: Constructing strictly positive types. *Theoretical Computer Science 342*, 1 (2005), 3–27.

[3] Altenkirch, T. The tao of types, 2021.

[4] Altenkirch, T. Categories for the lazy functional programmer, 2024.

[5] Curry, H. B. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences 20*, 11 (1934), 584–590.

[6] Damato, S., Altenkirch, T., and Ljungström, A. Formalising inductive and coinductive containers. *arXiv preprint arXiv:2409.02603* (2024).

[7] Dybjer, P. Representing inductively defined sets by wellorderings in martin-löf's type theory. *Theoretical computer science 176*, 1-2 (1997), 329–335.

[8] Goguen, J. A., Thatcher, J. W., Wagner, E. G., and Wright, J. B. Initial algebra semantics and continuous algebras. *Journal of the ACM (JACM) 24*, 1 (1977), 68–95.

[9] Keller, C., and Altenkirch, T. Normalization by hereditary substitutions. *proceedings of Mathematical Structured Functional Programming* (2010).

[10] Liang, S., Hudak, P., and Jones, M. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (1995), pp. 333–343.

[11] Martin-Löf, P. An intuitionistic theory of types. *Twenty-five years of constructive type theory 36* (1998), 127–172.

[12] Martin-Löf, P., and Sambin, G. *Intuitionistic type theory*, vol. 9. Bibliopolis Naples, 1984.

[13] Norell, U. *Towards a practical programming language based on dependent type theory*, vol. 32. Chalmers University of Technology, 2007.

[14] Program, T. U. F. Homotopy type theory: Univalent foundations of mathematics. *arXiv preprint arXiv:1308.0729* (2013).

[15] Rijke, E. Introduction to homotopy type theory. *arXiv preprint arXiv:2212.11082* (2022).