Investigating Quotient Inductive-Inductive Types

First year review report

Stefania Damato

supervised by Prof. Thorsten Altenkirch
& Dr Nicolai Kraus

School of Computer Science
Functional Programming Lab

University of Nottingham

July 2022

Abstract

Martin-Löf's dependent type theory is a formal language for programming and constructive mathematics. Inductive types are a central notion in this language, and allow us to define types by specifying their constructors. Homotopy type theory is a more recent development which interprets type theory in a homotopy theoretic way, whereby types are seen as spaces and the identity type is seen as the path space between two points. In this context, inductive types are not only defined in terms of their elements (points), but also in terms of their equalities (paths). These kinds of types are called higher inductive types (HITs). My thesis will provide a starting point for obtaining a general semantics for HITs. In particular, we look at quotiented types allowing induction-induction, known as quotient inductive-inductive types (QIITs).

Contents

1 Introduction **1**
- 1.1 Research Topic . . . 1
- 1.2 Progress to Date . . . 2
- 1.3 Overview of the Report . . . 3

2 Prerequisites **4**
- 2.1 Martin-Löf Type Theory . . . 4
- 2.2 Equality and HoTT . . . 5
- 2.3 Overview of Inductive Types . . . 6
- 2.4 Category Theory . . . 10
  - 2.4.1 Category of families . . . 10
  - 2.4.2 Category of elements . . . 10
  - 2.4.3 Ends . . . 10
  - 2.4.4 The Yoneda Lemma . . . 13
  - 2.4.5 Initial Algebra Semantics of Inductive Types . . . 14

3 Literature Review **17**

4 Topics Studied **20**
- 4.1 Containers . . . 20
  - 4.1.1 Inductive Types . . . 20
  - 4.1.2 Defining Containers . . . 22
  - 4.1.3 Categories of Containers . . . 23
  - 4.1.4 Initial Algebras and Terminal Coalgebras . . . 36
  - 4.1.5 Generalisations . . . 42
- 4.2 Models of type theory . . . 43

5 Future Work Plan **47**
- 5.1 Container Model of Type Theory . . . 47
- 5.2 Containerification of Semantics for (Q)IITs . . . 49
- 5.3 Syntax for (Q)IITs . . . 52

Appendix Cubical Agda Code **54**

1

Introduction

1.1 Research Topic

Martin-Löf type theory (MLTT) is a formal language developed to be used as a foundation for constructive mathematics as well as a theory for program construction via the propositions as types paradigm. One of the central notions of this theory is defining types inductively, which allows us to define types by simply specifying their constructors in a strictly positive way. Recent developments in homotopy theoretic interpretations of type theory have lead to homotopy type theory (HoTT), which alters our definition of an inductive type by not only viewing an inductive type as specified by its elements (or points), but also by its equalities (or paths). Inductive types in this context are called higher inductive types (HITs).

The main aim of my thesis will be to contribute towards the long-term goal of providing semantics for HITs. Specifically, I will be focussing on degenerate types of HITs called quotient inductive types (QITs), which allow path constructors up to first-order equality. I will also look at inductive-inductive types (IITs), which allow a higher degree of dependency between sorts of a given type. Combining these two together, we get quotient inductive-inductive types (QIITs). Our approach will involve using containers to restrict existing work to strictly positive types.

HITs have been used in the literature to define computational notions such as permutable trees and the syntax of type theory, as well as mathematical notions such as the Cauchy real numbers, surreal numbers, and homotopical structures like circles, spheres, and tori. Despite several uses of concrete HITs in the literature, there is still some difficulty around giving a fully general semantics and theoretical foundation for HITs, and particularly HITs

that allow induction-induction, or higher inductive-inductive types (HIITs). My project aims to act as a starting point to fill this gap in the literature.

1.2 Progress to Date

The main focus of the first 10 months of my PhD was learning about containers, reinterpreting the existing literature on containers in a more type theoretic way, and formalising properties of containers in Agda. I aim to combine this work into a modern, introductory paper on containers in the next few months. Details of my study into this area can be found in section 4.1. A formalisation in cubical Agda of a proof that the container extension functor $[\![_]\!]$ is full and faithful can be found in the appendix. I am currently working on formalising another proof of this, the more type theoretic version using the Yoneda lemma which is presented in section 4.1.3.

In order to understand the literature on HITs, containers, and models of type theory, I learned a lot more about category theory and homotopy type theory, which I only had a basic knowledge of prior to my PhD. Although I have programmed in Agda before and used it for my master's dissertation, I had never used cubical Agda or path types. The fact we wanted to use the cubical mode of Agda became clear early on since we want to study higher inductive types, and we need some extensionality results that are provable in cubical but would require postulating in vanilla Agda. Throughout this year, I became familiar with this new way of viewing equality and using the cubical mode of Agda, and am now able to use cubical Agda productively to formulate and prove results.

I attended the Midlands Graduate School 2022 held in Nottingham, where I followed courses on realizability, coalgebras, and HoTT with univalent foundations, and also organised and chaired a participants' talks session. I also attended TYPES 2022 held in Nantes, where I listened to numerous talks throughout the four-day conference, and after which I also attended the first Dedukti school.

I have helped teach a number of undergraduate modules. In the first semester, I helped out in weekly lab sessions and prepared and run weekly tutorials for 'COMP2009 Algorithms, Correctness, and Efficiency'. In the second semester, I helped out in weekly lab sessions for the functional pro-

gramming part of 'COMP1009 Programming Paradigms'. I graded coursework for both modules as well as for 'COMP2012 Languages and Computation'.

I attended weekly Type Theory Cafe seminars where I learned about ongoing research by my colleagues in the Functional Programming Lab and other topics related to type theory. I contributed to one of the seminars by talking about my studies on containers. I also attended weekly FP lunch sessions where we usually listened to a short talk by someone from the lab, or tried to solve a computer science related puzzle.

1.3 Overview of the Report

Chapter 2 presents an introduction to the general background of this report. Namely, we introduce MLTT, equality and identity types, HoTT, classes of inductive types, and some category theory background.

Chapter 3 reviews the related literature, and further contextualises and motivates our project.

Chapter 4 goes into detail about the topics studied over the past 10 months. We cover containers and give a brief description of models of type theory.

Chapter 5 contains our plans for the rest of the PhD, and sets out clear objectives and some explanation of how they can be achieved.

2

Prerequisites

This chapter gives an overview of the background material required for the rest of this report. Throughout this report, we use $\underline{\mathbf{C}}$ to denote a category and $|\underline{\mathbf{C}}|$ to denote the type of its objects. Whenever we leave out composition, associativity, and identity from the definition of a category, or preservation of identity morphisms and composition from the definition of a functor, it is because they follow in the obvious way. $\underline{\mathbf{Set}}$ refers to the category whose objects are (small) sets (having uniqueness of identity proofs (UIP)) of type Set and whose morphisms are functions. We use `Set` to mean Agda's universe of types with UIP. $\top$ or $\mathbf{1}$ or $\{*\}$ denote the unit type and $\bot$ or $\mathbf{0}$ denote the empty type. In cubical Agda code, we use `Type` to mean cubical Agda's universe of types. We use the symbol $:=$ when we are first defining something.

2.1 Martin-Löf Type Theory

Per Martin-Löf's type theory was designed as a foundational language for constructive mathematics and programming, based on the Brouwer-Heyting-Kolmogorov interpretation of logic. Being a type theory, this language identifies propositions with types, and the proofs of a proposition with the elements of the type corresponding to the proposition. It differs from other constructive type theories by introducing dependent types and offering a novel approach to equality.

Martin-Löf's formulations of type theory (Martin-Löf, 1971, 1972, 1982; Martin-Löf and Sambin, 1984) include inductive definitions of, among others, the set of natural numbers $\mathbb{N}$, finite sets $\mathrm{Fin}(n)$, and the disjoint union of two types $A + B$. Inductive definitions are fundamental to modern type theory and functional programming languages, and have been studied and

generalised extensively. An *inductive type* X is one which can be defined by providing a list of constructors, each of which is a function (possibly having zero arguments) with codomain X, specifying how to form elements of this type. Our project focusses on the semantics of such inductive types, so we give a summary of the different classes of inductive types in section 2.3.

2.2 Equality and HoTT

Martin-Löf type theory has two different kinds of equality. *Definitional equality* is the stronger kind of equality, and is not a type per se but a judgment in the metatheory of the language. In Agda code this is denoted by =. On the other hand, *propositional equality* is a particular type constructor `Id : {A : Type} -> A -> A -> Type` which relates two terms of the same type. In Agda code this is denoted by ≡. Unlike definitional equality, propositional equality can be treated like any other type in type theory.

To clarify this distinction, we provide some examples. When defining a function $f : \mathbb{N} \to \mathbb{N}$ by $f(x) = x{+}5$, that $f(2)$ and 7 are equal is definitional—it is simply a matter of expanding out a definition. It would not make sense to reason about this equality as a proposition. On the other hand, that $x + 5$ is equal to $5 + x$ is a proposition that can be proved, and is hence a propositional equality.

An intensional type theory treats the two equalities differently, but has some undesirable qualities like making functional extensionality unprovable. One workaround is to introduce it as an axiom, but this then introduces other potential problems like breaking canonicity. An extensional type theory extends intensional type theory with a reflection rule that essentially forces propositional equality back into definitional equality:

$$\frac{\Gamma \vdash p : \mathtt{Id}\, x\, y}{\Gamma \vdash x \equiv y}$$

This however makes Martin-Löf's type theory undecidable so it is also undesirable.

Homotopy type theory (HoTT) is a relatively new field of study which proposes a different view of equality. In this view, types are regarded as topological spaces and the identity type `Id` on two terms becomes the type of paths between two objects in the space. Inductive types in HoTT are called

higher inductive types (HITs) as they not only allow the constructors to produce points of the type being defined, but also elements of its identity types, i.e. equalities.

2.3 Overview of Inductive Types

Simple types

Types with zero or more constructors, each of which can be non-recursive or recursive.

Example 1: The unit type having one (non-recursive) constructor `tt`.

```
data ⊤ : Type where
  tt : ⊤
```

Example 2: The type of natural numbers à la Peano, specifying that 0 is a natural number, and that if n is a natural number, then so is its successor `suc` n.

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

Example 3: The type of binary trees storing data in the leaves. Note that this type has a parameter A (placed before the :) of the type of data to be stored in the tree.

```
data Tree (A : Type) : Type where
  leaf : A → Tree A
  node : Tree A → Tree A → Tree A
```

The W-type is a canonical form for simple inductive types. It is the type of well-ordered trees and is formed by providing a type `A` and a type `B : A -> Type` indexed by `A`. To construct elements of $W_{x:\,A}B(x)$, we use the constructor

$$\mathrm{sup}\colon (a : A) \to (B(a) \to W_{x:\,A}B(x)) \to W_{x:\,A}B(x).$$

Similarly, the M-type is a canonical form for simple coinductive types, but we will not be going into coinductive types here.

Dependent types (a.k.a. Inductive families a.k.a. Indexed inductive types)

Types that depend on a value from an input.

Example 1: The type of vectors of length n having elements of type `A`. Here, A is a parameter whereas $\mathbb{N}$ is an index (placed after the :). `Vec A` defines a collection of types, as it encapsulates the definitions of the types `Vec A zero`, `Vec A (suc zero)`, `Vec A (suc (suc zero))`, and so on, as opposed to a single type. Note how the target type of the constructors is different.

```
data Vec (A : Type) : ℕ → Type where
  [] : Vec A zero
  _::_ : {n : ℕ} → A → Vec A n → Vec A (suc n)
```

A use case for such a type is being able to express finer requirements for our code. Appending two vectors of length i and j, should result in a vector of length $i + j$. This constraint can be expressed in the type of the append function for vectors, but we cannot do the same for lists.

```
_++v_ : {A : Type}{i j : ℕ} → Vec A i → Vec A j → Vec A (i + j)
[] ++v y = y
(x :: xs) ++v y = x :: (xs ++v y)
```

Example 2: The type of finite sets of length n. For an $n > 0$, `Finite n` is isomorphic to the type having elements $\{0, 1, \cdots, n-1\}$.

```
data Finite : ℕ → Type where
  f-z : {n : ℕ} → Finite (suc n)
  f-s : {n : ℕ} → Finite n → Finite (suc n)
```

Mutual inductive types

Types with more than one sort, where the constructors of one sort can make use of the other sort. Mutual inductive types can be rewritten as simple types.

Example: The mutual definition of evens and odds. The two sorts are `even` and `odd`. Note how `odd` appears in the definition of a constructor of `even`.

```
data even : Type
data odd : Type

data even where
  e-zero : even
  e-suc : odd → even

data odd where
  e-suc : even → odd
```

Inductive-inductive types (IITs)

Types with more than one sort, of type `A : Type`, `B : A -> Type`, where the constructors of `B` may refer to those of `A` and crucially, the constructors of `A` may refer to those of `B`. (In general, IITs can have more than two sorts, but these have been shown to be reducible to IITs with only two sorts.)

Example: The type of contexts and types defined within that context. This is part of the syntax of type theory.

```
data Con : Type
data Ty : Con → Type

data Con where
  ◇ : Con
  _,_ : (Γ : Con) → Ty Γ → Con

data Ty where
  ι : (Γ : Con) → Ty Γ
  σ : (Γ : Con) (A : Ty Γ) (B : Ty (Γ , A)) → Ty Γ
```

Quotient Inductive types (QITs)

Types that have not only point constructors as we have seen so far, but also path constructors of equalities on the type. These path constructors are only allowed to be first order equalities (and not higher order), so that the inductive types defined are h-sets.

Example: The type of permutable trees, which are $\mathbb{N}$-branching trees modulo permutations of subtrees. (Note that we do not have a set-truncation constructor, but a set-truncation is implicitly included in the statement `PermTree : Set`, ensuring that `PermTree` is an h-set.)

```
data PermTree : Set where
  leaf : PermTree
  node : (ℕ → PermTree) → PermTree
  perm : (f : ℕ → PermTree) (g : ℕ → ℕ) → isIso g →
         (node f) ≡ node (f ∘ g)
```

Quotient Inductive-Inductive Types (QIITs)

Types that combine the last two classes together, i.e. types that allow induction-induction as well as first order equalities.

Example: The same `Con Ty` example given for IITs, except we rewrite `Con` as shown below. Now `Con= Ty` uses both induction-induction and first order path constructors.

```
data Con= where
  ◇ : Con=
  _,_ : (Γ : Con=) → Ty Γ → Con=
  eq : (Γ : Con=) (A : Ty Γ) (B : Ty (Γ , A)) → ((Γ , A) , B) ≡ (Γ , σ A B)
```

Higher inductive types (HITs)

Types that allow point constructors and path constructors up to any level of equality, i.e. allowing higher order equalities. QITs can be considered degenerate HITs.

Example: The type of the circle defined as a HIT. Note how the last constructor constructs an equality (or path). The points on the circle are represented by the different proofs of `base ≡ base`, that is, `loop`, `loop ∘ loop`, `loop`$^{-1}$`∘ loop ∘ loop`, etc.

```
data S¹ : Set where
  base : S¹
  loop : base ≡ base
```

2.4 Category Theory

This section aims to introduce several category theory notions to be used later on in the report. We presuppose some very basic category theory definitions such as categories, functors, natural transformations, isomorphisms, and terminal objects—a good starting point to these from a mathematical perspective can be found in Leinster (2016), while Milewski (2018) and Altenkirch (2019) provide a more computer scientific perspective. For a category $\underline{\mathbf{C}}$ we denote its type of objects by $|\underline{\mathbf{C}}|$ and its type of morphisms between objects A and B by $\underline{\mathbf{C}}(A, B)$, or sometimes $f \colon A \to B$.

2.4.1 Category of families

The *category of families of sets*, denoted by $\underline{\mathbf{Fam}}$, is defined as follows.

- Objects are pairs $(A, (B_a)_{a:A})$, where A is a set and B_a is an A-indexed family of sets $B \colon A \to \text{Set}$.
- Morphisms $(A, (B_a)_{a:A}) \to (A', (B'_{a'})_{a':A'})$ are pairs (f, g) where $f \colon A \to A'$ and $g \colon (a : A) \to B_a \to B'_{f\,a}$.

2.4.2 Category of elements

Given a functor $F \colon \underline{\mathbf{C}} \to \underline{\mathbf{Set}}$, the *category of elements* denoted by $\int F$, is defined as follows.

- Objects are pairs of type $((c : |\underline{\mathbf{C}}|), F\,c)$.
- Morphisms $(c, F_c) \to (d, F_d)$ are pairs (u, f) where $u \colon c \to d$ and $f \colon (F\,u)\,F_c = F_d$.

2.4.3 Ends

In order to make precise some notation used in the next section and in later chapters, we need to define ends. We start off by defining profunctors, which can be thought of as bifunctors that are contravariant on the first argument and covariant on the second.

Definition 2.1. Given a category $\underline{\mathbf{C}}$, a *profunctor*[1] $F \colon \underline{\mathbf{C}}^{\text{op}} \times \underline{\mathbf{C}} \to \underline{\mathbf{Set}}$ is

[1] We define a special case of profunctors here, namely the ones in terms of just one category as opposed to two.

defined as follows:

- Given objects c, d in $\underline{\mathbf{C}}$, F maps them to an object $F(c, d)$ in $\underline{\mathbf{Set}}$.
- Given morphisms $f : c \to c'$ and $g : d \to d'$ in $\underline{\mathbf{C}}$, F maps them to a morphism $F(f, g) : F(c', d) \to F(c, d')$ in $\underline{\mathbf{Set}}$.

The conditions for identity and composition are given by $F(\mathrm{id}_c, \mathrm{id}_d) = \mathrm{id}_{F(c,d)}$, and for $c_1 \xrightarrow{g'} c_2 \xrightarrow{g} c_3$ and $d_1 \xrightarrow{f'} d_2 \xrightarrow{f} d_3$, $F((g \circ g'), (f \circ f')) = F(g, f') \circ F(g', f)$ respectively. ■

An end is a generalisation of a limit of a functor. Similarly to how a limit of a functor is a universal cone, an end of a profunctor is a universal wedge.

Definition 2.2. Given a profunctor $F : \underline{\mathbf{C}}^{\mathrm{op}} \times \underline{\mathbf{C}} \to \underline{\mathbf{Set}}$, a *wedge* on F is an object X : Set and a family of morphisms $\chi_c : X \to F(c, c)$ for each c in $\underline{\mathbf{C}}$, such that for any morphism $f : c \to c'$, the below diagram commutes.

$$\begin{array}{ccccc} & & X & & \\ & \swarrow^{\chi_c} & & \searrow^{\chi_{c'}} & \\ F(c,c) & & & & F(c',c') \\ & \searrow_{F\,\mathrm{id}_c\,f} & & \swarrow_{F\,f\,\mathrm{id}_{c'}} & \\ & & F(c,c') & & \end{array}$$

■

Definition 2.3. Given a profunctor $F : \underline{\mathbf{C}}^{\mathrm{op}} \times \underline{\mathbf{C}} \to \underline{\mathbf{Set}}$, an *end* of F is a universal wedge on F, i.e. an object E and a family of morphisms $\epsilon_c : E \to F(c, c)$, such that any other wedge X and $\chi_c : X \to F(c, c)$ factors through E via a unique map h as shown below.

$$\begin{array}{ccccc} & & X & & \\ & \swarrow^{\chi_c} & \downarrow h & \searrow^{\chi_{c'}} & \\ & & E & & \\ & \swarrow^{\epsilon_c} & & \searrow^{\epsilon_{c'}} & \\ F(c,c) & & & & F(c',c') \\ & \searrow_{F\,\mathrm{id}_c\,f} & & \swarrow_{F\,f\,\mathrm{id}_{c'}} & \\ & & F(c,c') & & \end{array}$$

■

Example 2.4. If $\underline{\mathbf{C}}$ is a locally small category (i.e. all of its homsets are small sets), then we can define the profunctor

$$\begin{aligned}
\mathrm{Hom}_{\underline{\mathbf{C}}}(-,-)\colon \underline{\mathbf{C}}^{\mathrm{op}} \times \underline{\mathbf{C}} \to \underline{\mathbf{Set}} \\
\mathrm{Hom}_{\underline{\mathbf{C}}}(c,c') : \mathrm{Set} \\
\mathrm{Hom}_{\underline{\mathbf{C}}}(c,c') \coloneqq \underline{\mathbf{C}}(c,c')
\end{aligned}$$

$$\begin{aligned}
\mathrm{Hom}_{\underline{\mathbf{C}}}(c \xrightarrow{f} c', d \xrightarrow{g} d')\colon \mathrm{Hom}_{\underline{\mathbf{C}}}(c',d) \to \mathrm{Hom}_{\underline{\mathbf{C}}}(c,d') \\
\mathrm{Hom}_{\underline{\mathbf{C}}}(c \xrightarrow{f} c', d \xrightarrow{g} d')\, h \coloneqq g \circ h \circ f
\end{aligned}$$

Now consider functors $F, G\colon \underline{\mathbf{C}} \to \underline{\mathbf{D}}$. We can define the functor $\mathrm{Nat}_{F,G}$.

$$\begin{aligned}
\mathrm{Nat}_{F,G}\colon \underline{\mathbf{C}}^{\mathrm{op}} \times \underline{\mathbf{C}} \to \underline{\mathbf{Set}} \\
\mathrm{Nat}_{F,G}(c,c') : \mathrm{Set} \\
\mathrm{Nat}_{F,G}(c,c') \coloneqq \underline{\mathbf{D}}(F\,c, G\,c')
\end{aligned}$$

$$\begin{aligned}
\mathrm{Nat}_{F,G}(c \xrightarrow{f} c', d \xrightarrow{g} d')\colon \underline{\mathbf{D}}(F\,c', G\,d) \to \underline{\mathbf{D}}(F\,c, G\,d') \\
\mathrm{Nat}_{F,G}(c \xrightarrow{f} c', d \xrightarrow{g} d')\, h \coloneqq G\,g \circ h \circ F\,f
\end{aligned}$$

Then note that a wedge on $\mathrm{Nat}_{F,G}$ is an object W : Set and a family of morphisms $\chi_c\colon W \to \mathrm{Nat}_{F,G}(c,c)$, such that we have

$$\begin{array}{ccccc}
 & & W & & \\
 & \swarrow^{\chi_c} & & \searrow^{\chi_{c'}} & \\
\mathrm{Nat}_{F,G}(c,c) = \underline{\mathbf{D}}(F\,c, G\,c) & & & & \mathrm{Nat}_{F,G}(c',c') = \underline{\mathbf{D}}(F\,c', G\,c') \\
 & \searrow_{\mathrm{Nat}_{F,G}(\mathrm{id}_c, f)} & & \swarrow_{\mathrm{Nat}_{F,G}(f, \mathrm{id}_{c'})} & \\
 & & \mathrm{Nat}_{F,G}(c,c') & &
\end{array}$$

Now note that for $h_1 : \underline{\mathbf{D}}(F\,c, G\,c)$ and $h_2 : \underline{\mathbf{D}}(F\,c', G\,c')$,

$$\begin{aligned}
\mathrm{Nat}_{F,G}(\mathrm{id}_c, f)\, h_1 &= G\,f \circ h_1 \circ F\,\mathrm{id}_c \\
&= G\,f \circ h_1 && \text{functoriality of } F \\
\mathrm{Nat}_{F,G}(f, \mathrm{id}_{c'})\, h_2 &= G\,\mathrm{id}_{c'} \circ h_2 \circ F\,f \\
&= h_2 \circ F\,f && \text{functoriality of } G
\end{aligned}$$

and by substituting χ_c for h_1 and $\chi_{c'}$ for h_2, the commutative diagram above gives us that

$$G\,f \circ \chi_c = \chi_{c'} \circ F\,f.$$

This is precisely the naturality condition required for a natural transformation between F and G. So the family of morphisms χ_c represents a natural transformation, and its commutative diagram is the naturality condition required.

Since a wedge of $\mathrm{Nat}_{F,G}$ represents a natural transformation from F to G, an end of $\mathrm{Nat}_{F,G}$ represents the set of all natural transformations from F to G.

For a profunctor F, we use the notation $\mathrm{End}\,F = \int_c F(c,c)$. Hence, we can represent the set of natural transformations from F to G, denoted by $[\underline{\mathbf{C}}, \underline{\mathbf{D}}](F,G)$, as

$$[\underline{\mathbf{C}}, \underline{\mathbf{D}}](F,G) = \mathrm{End}\,\mathrm{Nat}_{F,G} = \int_c \mathrm{Nat}_{F,G}(c,c) = \int_c \underline{\mathbf{D}}(F\,c, G\,c).$$

This integral notation will be used in the next subsection as well as in section 4.1.

2.4.4 The Yoneda Lemma

A very important category theory result that will also be used extensively in section 4.1 is the Yoneda lemma. Although the standard presentation of the Yoneda lemma is its contravariant version, here we present the covariant version as it is the one we predominantly use.

The Yoneda lemma is a statement about representable functors, so we start off by defining them.

Definition 2.5. Let $\underline{\mathbf{C}}$ be a locally small category and A an object in $\underline{\mathbf{C}}$. The *covariant hom functor* H^A of type

$$H^A := \underline{\mathbf{C}}(A, -) : \underline{\mathbf{C}} \to \underline{\mathbf{Set}}$$

is defined as follows:

- for $B : |\underline{\mathbf{C}}|, H^A(B) = \underline{\mathbf{C}}(A, B)$,

- for map $B \xrightarrow{g} B'$ in $\underline{\mathbf{C}}$, define

$$H^A(g) = \underline{\mathbf{C}}(A, g) : \underline{\mathbf{C}}(A, B) \to \underline{\mathbf{C}}(A, B')$$
$$H^A(g)\, p \coloneqq g \circ p$$

for $p : \underline{\mathbf{C}}(A, B)$.

Definition 2.6. A functor $X : \underline{\mathbf{C}} \to \underline{\mathbf{Set}}$ is called *representable* if $X \cong H^A$ for some A in $\underline{\mathbf{C}}$.

Theorem 2.7. (Yoneda lemma) Consider a locally small category $\underline{\mathbf{C}}$. Take an object A in $\underline{\mathbf{C}}$ and a functor $F : \underline{\mathbf{C}} \to \underline{\mathbf{Set}}$. Then the set of natural transformations from $H^A = \underline{\mathbf{C}}(A, -)$ to F is isomorphic to the set of $F(A)$s:

$$[\underline{\mathbf{C}}, \underline{\mathbf{Set}}]\,(H^A, F) \cong F(A)$$

naturally in A and F.

Throughout the rest of the report, the notation we will use for the Yoneda lemma is

$$\int_{X:\mathrm{Set}} ((A \to X) \to F\, X) \cong F A,$$

with the integral notation introduced in section 2.4.3, as this is closer to how we think about it type theoretically.

Intuitively, the Yoneda lemma states that from a representable functor, you can get to any other $\underline{\mathbf{Set}}$-valued functor F via natural transformations, and that this set of natural transformations (which could potentially be very big) is in fact very limited, and can be obtained by evaluating F at the 'representer' A.

2.4.5 Initial Algebra Semantics of Inductive Types

This subsection outlines how in the categorical semantics of type theory, an inductive type is interpreted as the initial algebra of an endofunctor.

First, we illustrate how to obtain the endofunctor associated to an inductive type. As an example we look at the type of natural numbers $\mathbb{N}$.

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

This type is entirely described by its two constructors `zero` : ℕ and `suc` : ℕ → ℕ. In order to represent the two constructors as functions `z` and `s` respectively, we can rewrite `zero` as a function from the unit type $\{*\}$, so we obtain

$$\begin{aligned} \texttt{z} &: \{*\} \to \mathbb{N} \\ \texttt{s} &: \mathbb{N} \to \mathbb{N}. \end{aligned}$$

We can now view `z` and `s` as morphisms in the category **Set**, where $\{*\}$ is a terminal object and will therefore be represented henceforth as $\mathbf{1}$. This being a Cartesian closed category allows us to rewrite the morphisms using exponentials

$$\begin{aligned} \texttt{z} &: \mathbb{N}^{\mathbf{1}} \\ \texttt{s} &: \mathbb{N}^{\mathbb{N}} \end{aligned}$$

followed by products to describe them as a single pair

$$\texttt{z} \times \texttt{s} : \mathbb{N}^{\mathbf{1}} \times \mathbb{N}^{\mathbb{N}}$$

followed by some basic algebra which holds in any Cartesian closed category, along with going back to morphism notation

$$\begin{aligned} \texttt{z} \times \texttt{s} &: \mathbb{N}^{\mathbf{1}+\mathbb{N}} \\ \texttt{z} \times \texttt{s} &: \mathbf{1} + \mathbb{N} \to \mathbb{N}. \end{aligned}$$

The left hand side of the resultant type corresponds to an endofunctor **Set** → **Set**, which takes a set X to the sum $\mathbf{1} + X$:

$$\begin{aligned} F &: \underline{\mathbf{Set}} \to \underline{\mathbf{Set}} \\ F\,X &= \mathbf{1} + X. \end{aligned}$$

We have therefore obtained the endofunctor F over $\underline{\textbf{Set}}$ associated to the inductive type $\mathbb{N}$. Given F, we can now consider F-algebras (A, α) where $A : \underline{\textbf{Set}}$ is the carrier set and $\alpha \colon F\,A \to A$ represents constructors of the carrier. In this case, $\alpha \colon \mathbf{1} + A \to A$. We can form the category of F-algebras, where one such algebra would be the carrier set **Bool** with constructors `true` : **Bool** and `not` : **Bool** → **Bool**, where `not` is the usual boolean negation. Whenever F is an endofunctor corresponding to a strictly positive inductive type, the category of F-algebras has an initial object called the initial algebra, (I, ι). (I, ι) is initial if for any other F-algebra (A, α), there is a unique morphism $\mathrm{iter}_\alpha \colon I \to A$ such that the below commutes.

$$\begin{array}{ccc} F\,I & \xrightarrow{\iota} & I \\ {\scriptstyle F\,(\mathrm{iter}_\alpha)}\downarrow & & \downarrow{\scriptstyle \mathrm{iter}_\alpha} \\ F\,A & \xrightarrow[\alpha]{} & A \end{array}$$

By Lambek's lemma, $\iota \colon F\,I \to I$ is not merely a morphism but an isomorphism, making $F\,I \cong I$, and I a fixed point of F. The initial algebra (I, ι) corresponds exactly to the inductive type—in our example, I would be $\mathbb{N}$ and ι would be the pairing of `zero` and `suc`. We write that $\mathbb{N} \cong \mu X.\mathbf{1} + X$, where μ is a partial operator on functors, taking an endofunctor F to the carrier set of its initial algebra. The morphism iter_α is the iterator for I, and combined with the fact that it is unique, this corresponds to the elimination principle (or the dependent eliminator) for I.

3

Literature Review

This chapter further introduces the context of our project, by surveying existing related literature and locating our work within it.

The main goal of our thesis is to contribute towards providing semantics for higher inductive types (HITs) in HoTT. Despite there already being a lot of work in the literature making use of HITs (e.g. Brunerie (2016); Licata and Shulman (2013)), they still lack a general specification and theoretical foundations. Several approaches have been proposed (e.g. Lumsdaine and Shulman (2019)) but a precise definition of what a higher inductive type is in general is still an open problem. As a first step towards closing this gap, we will work on giving semantics for set-truncated HITs with induction-induction, known as quotient inductive-inductive types (QIITs). These types generalise normal inductive types in two ways. First, they allow the use of point constructors as well as path constructors up to first order equalities, meaning that these types ignore any higher equalities and possess the uniqueness of identity proofs property. Secondly, they allow the use of induction-induction, which introduces a higher dependency in a type's definition, and allows constructors of one sort to refer to constructors of another sort mutually.

Simple inductive types and inductive families

Simple inductive types (like the type $\mathbb{N}$ of natural numbers and the type `List A` of lists of type A) and inductive families (like the type `Vec A n` of vectors of type A and length n) already enjoy fully developed semantics. A well known result due to Dybjer (1996) extends Martin-Löf's encoding of the natural and ordinal numbers using W-types to an encoding of any inductive type represented by a strictly positive endofunctor on the category of sets. This establishes W-types as the universal type for simple inductive types.

Abbott et al. (2005) show that (n-ary) containers are a normal form for simple inductive types, and use them to generalise Dybjer's result to nested inductive types and provide a more complete description of the categorical infrastructure involved. Later, Altenkirch and Morris (2009) introduce indexed containers as a normal form for inductive families, and show that any inductive family can be represented as an indexed W-type, thereby identifying a universal type for inductive families. They also show that indexed W-types can be reduced to W-types.

The situation for (Q)IITs

A similar well-established theoretical foundation for inductive-inductive types (IITs) and QIITs has not yet been found. The first obstacle we face in formalising IITs is that due to the dependency we allow between constructors of different sorts, we have no way of expressing IITs by endofunctors, unlike the cases of simple inductive types and inductive families. This means that we cannot express their semantics as an initial algebra over a functor. Altenkirch et al. (2018) remedy this for the even more general case of QIITs. They specify a way to still represent QIITs using an algebra, but an algebra in a more general sense than what we usually mean (an F-algebra for a functor F). The algebra is constructed incrementally by adding one constructor at a time (where each constructor is represented by a functor), until once all the constructors are added, the QIIT can be obtained as the initial object of the resultant algebra. While this work sets out a general method for specifying IITs and QIITs, the specification is still too broad as it allows non-strictly positive types. Therefore, we want to improve on this work via 'containerification', i.e. we restrict the functors representing the constructors being added to container functors (as well as adding some other restrictions), thereby only allowing strictly positive definitions.

Container model of type theory

Since a constructor is an expression in type theory and we want to write this as a container, we need to be able to interpret any expression in type theory as a container. This means that before we can start working on the 'containerification' of the above work, we need to construct a model of type theory using containers. This idea has already been set out in an abstract by Altenkirch and Kaposi (2021), but the details of this have yet to be

worked out. von Glehn (2015) has presented a polynomial functor model of type theory using comprehension categories as notion of model. The same model was also presented by Atkey (2020) and Kovács (2020) using categories with families (CwFs) as notion of model. We plan on constructing a container model of type theory using CwFs, which has the same contexts and substitutions as the latter two but different types and terms.

A new syntax for (Q)IITs

After working out the full details of the container model of type theory and providing a full semantics for strictly positive IITs and QIITs, we want to analyse the syntax given rise to by this semantics. We expect this syntax to be a refinement of the theory of signatures presented by Kaposi et al. (2019). This syntax treats extending the arguments of a constructor the same way it treats extending the constructors of a type: using type dependency. The syntax resulting from our 'containerified' semantics will treat these two extensions differently, thereby giving rise to an alternative syntax for QIITs, which we conjecture will be complete for our semantics, i.e. if a type cannot be expressed within our syntax, then it is not a part of our semantics. If this is the case, then the syntax would constitute a universal QIIT, which would already be a significant achievement towards our long-term goal of providing a general specification for QIITs. Since such a universal QIIT would be a fairly complicated type, the next step would be to try and simplify it as much as possible, and to come up with a so-called QW-type, which would be a succinct type to which all strictly positive QIITs could be reduced, in much the same way that all strictly positive simple types can be reduced to W-types.

4

Topics Studied

In view of the literature surveyed in chapter 3, it became clear that to be able to start working on my research goals, I needed to carry out more in-depth study in two main areas, namely containers and models of type theory. This chapter summarises my studies and expands on chapter 3 by going into further detail on these two topics.

4.1 Containers

Over the years, containers have been used in various contexts to solve different problems. Our current interest is applying them to get a semantics for HITs. In this section we will motivate, define, and analyse the properties of containers.

Most of this section's content is an adaptation of proofs given in Abbott et al. (2005), however we take a much more type theoretic approach, while theirs is more category theoretic. In certain cases we give different proofs to theirs, we provide examples, and supplement our proofs with Agda code of non-trivial proof steps. The discussion and proof on container exponentials was adapted from Altenkirch et al. (2010).

4.1.1 Inductive Types

Informally, a *(simple) inductive type* X is one which can be defined by providing a list of constructors, each of which is a function (possibly having zero arguments) with codomain X, specifying how to form elements of this type. The simplest example is the set of natural numbers $\mathbb{N}$, whose constructors are `zero` and `suc`. This specification of a (simple) inductive type is however pretty general, and allows us to define types that we can not make sense

of predicatively. For example, we are allowed to define an 'inductive type' `Contra` shown below.

```
data Contra : Type where
  c : ((Contra → Bool) → Bool) → Contra
```

First of all, we cannot make sense of such a definition semantically. When talking about inductive types, we think of them as being defined 'in stages', e.g. for the natural numbers $\mathbb{N}$, the only element we can construct at first is `zero`, then in the second step, we can use this element and construct a new element `suc n` for `n` being `zero`, then in the next step we can use this latest constructed element, and so on. In the case of `Contra`, we do not have a starting point of constructing a first element, so we cannot understand it semantically. Moreover, these types of definitions lead to contradictions when making certain assumptions (such as classical logic), and they admit non-terminating functions (in fact, Agda does not allow us to define such a type).

The condition we would like to impose to avoid such definitions is roughly that for an inductive type X, we allow X to occur in the input types of its constructors, but only to the right of arrows ($\rightarrow$) (The Univalent Foundations Program, 2013). For example, we allow constructors like $c\colon (N \rightarrow X) \rightarrow X$ for the type X, but not $d\colon (X \rightarrow N) \rightarrow X$ or $e\colon ((X \rightarrow N) \rightarrow N) \rightarrow X$.

Inductive types that obey the condition explained above are called strictly positive types, and they can be defined inductively as follows.

Definition 4.1. A *strictly positive type* in n variables, having type variables $X_1, \ldots, X_n$, can be built up inductively by the following rules (Abel and Altenkirch, 2000):

- if K is a type with no type variables, i.e. is a constant type, then K is a strictly positive type
- every type variable X_i, for $1 \leq i \leq n$, is a strictly positive type
- if F and G are strictly positive types, then their product $F \times G$ and their coproduct $F + G$ are strictly positive types

- if K is a constant type and F is a strictly positive type, then $K \to F$ is a strictly positive type
- if F is a strictly positive type in $n+1$ variables, then $\mu X.F$ and $\nu X.F$ are strictly positive types in n variables ■

While this provides a syntactic definition of strictly positive inductive types, we are still missing a semantic description. The initial algebra semantics of inductive types discussed in Section 2.4.5 provides us a general way of talking about the initial algebra of an endofunctor corresponding to an inductive type, but it does not tell us which endofunctors actually have initial algebras, i.e. which endofunctors correspond to inductive types that are strictly positive. This is the problem containers aim to address, namely, they are a canonical form for strictly positive inductive types and therefore give us a semantic description of them. Endofunctors arising from containers are precisely those corresponding to strictly positive inductive types.

4.1.2 Defining Containers

Now that we have motivated the need for containers, we define what a container is.

Definition 4.2. A *(unary) container* is given by a pair of types $S : \mathsf{Set}$ and $P : S \to \mathsf{Set}$, which we write as $S \lhd P$. ■

The idea is that we can fully represent a strictly positive inductive type by its 'shapes' and 'positions', where S is the type of shapes and P is the type of positions indexed by S, which to every shape assigns the set of positions at which data can be stored. To make better sense of this definition, we define the extension of a container, also known as a container functor.

Definition 4.3. A *container functor* associated to a container $S \lhd P$ is a functor naturally isomorphic to the functor $[\![S \lhd P]\!] : \mathsf{Set} \to \mathsf{Set}$ with the following actions on objects and morphisms.

- Given an $X : \mathsf{Set}$, $[\![S \lhd P]\!]\, X \coloneqq \sum(s : S)(P s \to X)$.
- Given $X, Y : \mathsf{Set}$ and a morphism $f \colon X \to Y$,

$$[\![S \lhd P]\!]\, f : [\![S \lhd P]\!]\, X \to [\![S \lhd P]\!]\, Y.$$

Given $s : S$ and $g \colon P s \to X$,

$$[\![S \lhd P]\!]\, f\,(s, g) \coloneqq (s, f \circ g).$$

That $[\![S \lhd P]\!]$ preserves identity morphisms follows from $[\![S \lhd P]\!]\, id\,(s, g) = (s, g)$, and that it preserves composition follows from $[\![S \lhd P]\!]\,(f \circ h)\,(s, g) = (s, (f \circ h) \circ g) = (s, f \circ (g \circ h)) = [\![S \lhd P]\!]\, f\,(s, h \circ g) = ([\![S \lhd P]\!]\, f) \circ ([\![S \lhd P]\!]\, h)\,(s, g)$ by associativity of morphisms in **Set**. ■

The container functor associated to $S \lhd P$ maps a type X to a choice of shape $s : S$ and for every position $P\,s$ associated to s, a value of type X to be stored at that position.

Example 4.4. The container representation of the list data type shown below

```
data List (A : Type) : Type where
  [] : List A
  _::_ : A → List A → List A
```

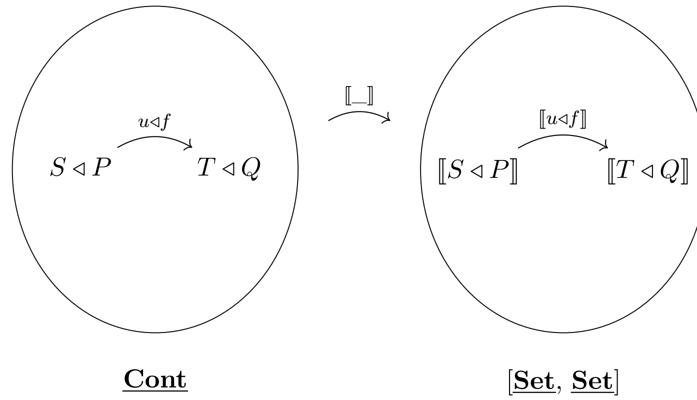
is given by $(n : \mathbb{N}) \lhd (\mathrm{Fin}\, n)$. The shape of a list is a natural number $n : \mathbb{N}$ representing its length, and given a length n, the positions of a list are the elements of a finite set of size n, $\mathrm{Fin}\, n$. The container functor associated to $(n : \mathbb{N}) \lhd (\mathrm{Fin}\, n)$ as defined in definition 4.3 allows us to represent concrete lists. For example, the list of **Char**s [‘h’, ‘e’, ‘l’, ‘l’, ‘o’] is represented as $\sum (5 : \mathbb{N})((0 \to \text{‘}h\text{’}; 1 \to \text{‘}e\text{’}; 2 \to \text{‘}l\text{’}; 3 \to \text{‘}l\text{’}; 4 \to \text{‘}o\text{’}) : \mathrm{Fin}\, 5 \to \mathbf{Char})$.

We will show later (example 4.19) that the container functor associated to `List` is isomorphic to the carrier set of the initial algebra over `List`’s endofunctor, i.e.

$$[\![(n : \mathbb{N}) \lhd (\mathrm{Fin}\, n)]\!]\, A \cong \mu X.1 + A \times X.$$

4.1.3 Categories of Containers

Now that we have defined containers and container functors, we can view them as objects in two different categories, related by the functor $[\![_]\!]$.



Definition 4.5. The *category of (unary) containers*, which we refer to hereafter as **Cont**, is defined as follows:

- Objects are (unary) containers as defined in definition 4.2, i.e. pairs $S : \mathsf{Set}$ and $P : S \to \mathsf{Set}$, written as $S \triangleleft P$.
- Morphisms $(S \triangleleft P) \to (T \triangleleft Q)$ are pairs $u \colon S \to T$ and $f \colon (s : S) \to Q\,(u\,s) \to P\,s$, written as $u \triangleleft f$.

Note how a morphism between containers is a function on shapes, together with a function that assigns to every target position a source position. This definition makes sense as is it always possible to pinpoint a target position's source, but not vice versa.

Example 4.6. The tail function on lists can be represented as a container morphism from the list container $(n : \mathbb{N}) \triangleleft (\mathrm{Fin}\,n)$ to itself, by defining the following.

```
u-lst : ℕ → ℕ
u-lst zero = zero
u-lst (suc n) = n

f-lst : (n : ℕ) → Fin (u-lst n) → Fin n
f-lst zero r = r
f-lst (suc n) r = fsuc r
```

The interesting cases are when the length $n \neq 0$, as the tail of the empty list is the empty list. `u-lst` represents the length of a list decreasing by

1 when taking its tail, while `f-lst` represents the index of a specific entry increasing by one when going from the tail to the original list.

In this example, we would not have been able to assign a target position to every source position. In particular, the head of the list has no target position.

Now that we have defined the category of containers, we are interested in relating a container $S \lhd P$ to its functorial interpretation $[\![S \lhd P]\!]$. We do this by defining the container extension functor $[\![_]\!]$.

Definition 4.7. The *container extension functor* $[\![_]\!] \colon \underline{\mathbf{Cont}} \to [\underline{\mathbf{Set}}, \underline{\mathbf{Set}}]$ is defined as follows:

- Given an object $S \lhd P$ in $\underline{\mathbf{Cont}}$, this is mapped to its container functor $[\![S \lhd P]\!]$ as defined in definition 4.3.
- Given a morphism $(u \lhd f) \colon (S \lhd P) \to (T \lhd Q)$ in $\underline{\mathbf{Cont}}$, this is mapped to the natural transformation $[\![u \lhd f]\!] \colon [\![S \lhd P]\!] \to [\![T \lhd Q]\!]$ with components $[\![u \lhd f]\!]_X \colon [\![S \lhd P]\!]\, X \to [\![T \lhd Q]\!]\, X$ for any $X : \mathsf{Set}$, defined as

$$[\![u \lhd f]\!]_X\, (s, h) \coloneqq (u\, s, \lambda\, q \to h(f\, s\, q)),$$

  for $s : S$ and $h \colon P\, s \to X$.

We say that a functor $F \colon \underline{\mathbf{Set}} \to \underline{\mathbf{Set}}$ is a *container functor* if it is naturally isomorphic to some functor $[\![S \lhd P]\!]$ for a container $S \lhd P$ (refer to definition 4.3). ■

Our motivation for containers was to provide a semantic representation for strictly positive functors. The natural next step is to think about mappings between these functors, which are natural transformations as defined in definition 4.7. The corresponding notion for containers is container morphisms as seen in definition 4.5. Rather surprisingly, there is a bijective correspondence between natural transformations on container functors and their representation as container morphisms. This means that every (polymorphic) function between strictly positive inductive types is uniquely representable as a container morphism. This correspondence can be shown by proving that the container extension functor $[\![_]\!]$ is full and faithful.

Theorem 4.8. The functor $[\![_]\!] : \underline{\mathbf{Cont}} \to [\underline{\mathbf{Set}}, \underline{\mathbf{Set}}]$ is full and faithful.

Proof. We need to show that given containers $S \lhd P$ and $T \lhd Q$, there is a bijection between $\underline{\mathbf{Cont}}(S \lhd P, T \lhd Q)$ and $[\underline{\mathbf{Set}}, \underline{\mathbf{Set}}]([\![S \lhd P]\!], [\![T \lhd Q]\!])$. So assume $X : \mathsf{Set}$.

$\int_{X:\mathrm{Set}} ([\![S \lhd P]\!]\, X \to [\![T \lhd Q]\!]\, X)$	
$= \int_{X:\mathrm{Set}} (\sum(s : S)(P\, s \to X) \to [\![T \lhd Q]\!] X)$	expanding definition of $[\![S \lhd P]\!]\, X$
$\cong \int_{X:\mathrm{Set}} ((s : S) \to (P\, s \to X) \to [\![T \lhd Q]\!] X)$	currying in $\underline{\mathbf{Set}}$: $\Pi\,((\Sigma\, A\, B)\, C) \cong \Pi\,(A\,(\Pi\, B\, C))$
$\cong (s : S) \to \int_{X:\mathrm{Set}} ((P\, s \to X) \to [\![T \lhd Q]\!] X)$	$\int$ and Π commute
$\cong (s : S) \to [\![T \lhd Q]\!]\,(P\, s)$	covariant Yoneda lemma: for $F : \underline{\mathbf{Set}} \to \underline{\mathbf{Set}}, A : \underline{\mathbf{Set}}$, $\int_{X:\mathrm{Set}} (A \to X, F\, X) \cong F\, A$
$= (s : S) \to \sum(t : T)(Q\, t \to P\, s)$	expanding definition of $[\![T \lhd Q]\!]\, X$
$\cong \sum(f : S \to T)((s : S) \to Q(f\, s) \to P\, s)$	type theoretic axiom of choice (see below)
$= (S \lhd P) \to (T \lhd Q)$	definition of container morphism

The type theoretic axiom of choice used in the penultimate step refers to the following.

```
tt-aoc : {A : Type} {B : A → Type} {C : (a : A) → B a → Type} →
         Iso ((a : A) → Σ (B a) (λ b → C a b))
             (Σ ((a : A) → B a) (λ f → (a : A) → C a (f a)))
tt-aoc = iso
           (λ f → (λ a → fst (f a)) , λ a → snd (f a))
           (λ {(f , g) → λ a → f a , g a})
           (λ _ → refl)
           (λ _ → refl)
```

□

The above gives us a bijection between morphisms $(S \triangleleft P) \to (T \triangleleft Q)$ in **Cont** and morphisms $[\![S \triangleleft P]\!] \to [\![T \triangleleft Q]\!]$ in [**Set**, **Set**].

Example 4.9. Consider functors $F\,X := X^m$ and $G\,X := X^n$. Written as containers, F is represented by $(1 \triangleleft \mathrm{Fin}\,m)$ and G by $(1 \triangleleft \mathrm{Fin}\,n)$. Then there is a bijection between natural transformations $X^m \implies X^n$ and functions $(u \triangleleft f)$ with $u\colon 1 \to 1$ and $f\colon \mathrm{Fin}\,n \to \mathrm{Fin}\,m$, which is isomorphic to functions $n \to m$. We know that there are precisely m^n of the latter, so we can conclude that there are also m^n natural transformations $X^m \implies X^n$.

We now turn our attention to the category **Cont** and look at its closure properties, namely, we show that it is closed under products, coproducts, composition, and exponentials.

Products and Coproducts

First of all, since we are considering functors **Set** $\to$ **Set** and since their category has products and coproducts, we can show that (a) $[\![_]\!]$ preserves them, and as a result that (b) **Cont** inherits them from [**Set**, **Set**].

Theorem 4.10. When considering container functors of type **Set** $\to$ **Set**, the following hold.

1. The container extension functor $[\![_]\!]$ preserves products and coproducts.
2. **Cont** inherits finite products and coproducts from [**Set**,**Set**].

Proof. Since $[\![_]\!]$ is a fully faithful functor of type **Cont** $\to$ [**Set**, **Set**], then if we show that products and coproducts of container functors in [**Set**,**Set**] are themselves container functors (i.e. that $[\![_]\!]$ preserves products and coproducts), we can reflect them to products and coproducts in **Cont** along $[\![_]\!]$.

1. We first compute the product of container functors. Assume X : Set.

$$
\begin{aligned}
&[\![S \triangleleft P]\!]\, X \times [\![T \triangleleft Q]\!]\, X \\
&= \sum(s : S)(P\, s \to X) \times \sum(t : T)(Q\, t \to X) \\
&\cong \sum((s,t) : S \times T)(P\, s + Q\, t \to X) \\
&= [\![((s,t) : S \times T) \triangleleft (P\, s + Q\, t)]\!]\, X
\end{aligned}
$$

The crucial step from the second line to the third line is the following isomorphism.

```
Σ-comm : {A C D : Type} {B : A → Type} {E : D → Type} →
         Iso (Σ (Σ A (λ a → B a → C))
                (λ _ → Σ D (λ d → E d → C)))
             (Σ (Σ A (λ _ → D)) (λ ad → B (fst ad) ⊎ E (snd ad) → C))
Σ-comm = iso (λ {((a , f) , (d , g)) → (a , d) , λ {(inl b) → f b ; (inr e) → g e}})
             (λ {((a , d) , f) → (a , λ b → f (inl b)) , (d , λ e → f (inr e))})
             (λ {((a , d) , f) →
                  ΣPathP (refl , funExt λ {(inl b) → refl ; (inr e) → refl})})
             (λ _ → refl)
```

Next, we compute the coproduct of container functors. Again assume X : Set.

$$
\begin{aligned}
&[\![S \triangleleft P]\!]\, X + [\![T \triangleleft Q]\!]\, X \\
&= \sum(s : S)(P\, s \to X) + \sum(t : T)(Q\, t \to X) \\
&\cong \sum(b : \mathrm{Bool})(b = \mathrm{true} \to \sum(s : S)(P\, s \to X); \\
&\quad b = \mathrm{false} \to \sum(t : T)(Q\, t \to X)) \\
&\cong \sum(\sum(b : \mathrm{Bool})(b = \mathrm{true} \to s : S; b = \mathrm{false} \to t : T)) \\
&\quad (\lambda\, (\mathrm{true}, s) \to P\, s \to X; (\mathrm{false}, t) \to Q\, t \to X) \\
&\cong \sum(S + T)(\lambda\, (\mathrm{inl}\, s) \to P\, s \to X; (\mathrm{inr}\, t) \to Q\, t \to X) \\
&= [\![(S + T) \triangleleft (\lambda\, (\mathrm{inl}\, s) \to P\, s; (\mathrm{inr}\, t) \to Q\, t)]\!]\, X
\end{aligned}
$$

We make use of the fact that $A+B \cong \sum(b : \mathrm{Bool})(b = \mathrm{true} \to A; (b = \mathrm{false}) \to B)$ twice throughout the proof. From the third to the fourth line, we make use of the below isomorphism.

```
Σ-assoc : {A : Type} {B : A → Type} {C : Σ A B → Type} →
            Iso (Σ A (λ a → Σ (B a) (λ Ba → C (a , Ba))))
                (Σ (Σ A B) C)
Σ-assoc = iso (λ {(a , (b , c)) → (a , b) , c})
              (λ {((a , b) , c) → a , (b , c)})
              (λ _ → refl)
              (λ _ → refl)
```

We have shown that $[\![_]\!]$ preserves products and coproducts by showing that products and coproducts of container functors are themselves container functors.

2. We now give a more detailed explanation and proof as to why $\underline{\mathbf{Cont}}$ inherits products and coproducts from $[\underline{\mathbf{Set}},\underline{\mathbf{Set}}]$ along $[\![_]\!]$. We illustrate the case for products; the case for coproducts follows similarly.

   For any two containers $C = S \triangleleft P$ and $C' = T \triangleleft Q$, define

$$(S \triangleleft P) \mathbin{\tilde{\times}} (T \triangleleft Q) = ((s,t) : S \times T) \triangleleft (P\,s + Q\,t)$$

   as given by our computation in part 1, as an object in $\underline{\mathbf{Cont}}$. We will show that $\tilde{\times}$ is in fact the product in $\underline{\mathbf{Cont}}$ by showing the universal property of the product, i.e. that for any container D,

$$\underline{\mathbf{Cont}}(D, C \mathbin{\tilde{\times}} C') \cong \underline{\mathbf{Cont}}(D, C) \mathbin{\tilde{\times}} \underline{\mathbf{Cont}}(D, C').$$

$$\begin{aligned}
&\underline{\mathbf{Cont}}(D, C \mathbin{\tilde{\times}} C') && \\
&\cong [\underline{\mathbf{Set}}, \underline{\mathbf{Set}}]([\![D]\!], [\![C \mathbin{\tilde{\times}} C']\!]) && [\![_]\!] \text{ full and faithful} \\
&\cong [\underline{\mathbf{Set}}, \underline{\mathbf{Set}}]([\![D]\!], [\![C]\!] \times [\![C']\!]) && [\![_]\!] \text{ preserves products} \\
&\cong [\underline{\mathbf{Set}}, \underline{\mathbf{Set}}]([\![D]\!], [\![C]\!]) \times [\underline{\mathbf{Set}}, \underline{\mathbf{Set}}]([\![D]\!], [\![C']\!]) && [\underline{\mathbf{Set}}, \underline{\mathbf{Set}}] \text{ has products} \\
&\cong \underline{\mathbf{Cont}}(D, C) \mathbin{\tilde{\times}} \underline{\mathbf{Cont}}(D, C') && [\![_]\!] \text{ full and faithful}
\end{aligned}$$

   This shows that $\underline{\mathbf{Cont}}$ has products as defined by reflecting our compuation in part 1 along $[\![_]\!]$, and similarly it has coproducts:

$$(S \triangleleft P) \times (T \triangleleft Q) = ((s,t) : S \times T) \triangleleft (P\,s + Q\,t),$$

$$(S \triangleleft P) + (T \triangleleft Q) = (S + T) \triangleleft (\lambda\,(\mathrm{inl}\, s) \to P\, s; (\mathrm{inr}\, t) \to Q\, t). \quad \square$$

Composition

We now look at what happens when composing containers. If $[\![S \triangleleft P]\!], [\![T \triangleleft Q]\!] : \underline{\mathbf{Set}} \to \underline{\mathbf{Set}}$ and $X : \mathsf{Set}$, then we can consider $([\![S \triangleleft P]\!] \circ [\![T \triangleleft Q]\!])\, X = [\![S \triangleleft P]\!] \circ ([\![T \triangleleft Q]\!]\, X)$ of type Set. We show that this is itself a container functor and obtain a definition for composition in $\underline{\mathbf{Cont}}$.

Theorem 4.11. When considering container functors of type $\underline{\mathbf{Set}} \to \underline{\mathbf{Set}}$, $[\![_]\!]$ preserves composition.

Proof. Assume $X : \mathsf{Set}$.

$$\begin{aligned}
&([\![S \triangleleft P]\!] \circ [\![T \triangleleft Q]\!])\, X \\
&= [\![S \triangleleft P]\!] \circ ([\![T \triangleleft Q]\!]\, X) \\
&= \sum(s : S)(P\, s \to \sum(t : T)(Q\, t \to X)) && \text{definition of } [\![_]\!] \\
&\cong \sum(s : S)(\sum(f : P\, s \to T)((p : P\, s) \to Q(f\, p) \to X)) && \texttt{tt-aoc} \\
&\cong \sum(\sum(s : S)(f : P\, s \to T))((p : P\, s) \to Q(f\, p) \to X) && \Sigma\texttt{-assoc} \\
&= [\![(\sum(s : S)(f : P\, s \to T)) \triangleleft ((p : P\, s) \to Q(f\, p))]\!]\, X && \text{definition of } [\![_]\!]
\end{aligned}$$

$\square$

We therefore define container composition as

$$(S \triangleleft P) \circ (T \triangleleft Q) = (\sum(s : S)(f : P\, s \to T)) \triangleleft ((p : P\, s) \to Q(f\, p)).$$

Example 4.12. By the definition above, the container representation of `List (List A)` type, where $A : \mathrm{Set}$ is the type of entries of the nested list, is as follows:

$$C = (\sum(n : \mathbb{N})(f : \mathrm{Fin}\, n \to \mathbb{N})) \triangleleft ((p : \mathrm{Fin}\, n) \to \mathrm{Fin}\,(f\, p)).$$

The shape of the above container is precisely $[\![(n : \mathbb{N}) \triangleleft (\mathrm{Fin}\, n)]\!]\, \mathbb{N} \cong \mathrm{List}\, \mathbb{N}$. The length of this list represents the length of the outer list, and its entries

represent the length of each of the inner lists. The positions then assign data to each of the slots dictated by the lengths.

For example, the list $[[\text{`}a\text{'}, \text{`}b\text{'}], [], [\text{`}c\text{'}]]$ of type `List (List Char)` is represented by

$$\begin{aligned}\sum(\sum(3)(\lambda\{0 \to 2;\\ 1 \to 0;\\ 2 \to 1\}))\\ \lambda\{0 \to \lambda\{0 \to \text{`}a\text{'}; 1 \to \text{`}b\text{'}\}\\ 1 \to \lambda()\\ 2 \to \lambda\{0 \to \text{`}c\text{'}\}\}\end{aligned}$$

of type $[\![C]\!]$ `Char`.

Exponentiation

The last closure property of **Cont** we look at is exponentiation. First of all, we note that for two functors F and G in [**Set**,**Set**], their exponential G^F does not necessarily exist. For assume that it does exist. Then we can compute what kind of form it will have. Assume $A : \mathsf{Set}$.

$G^F(A)$	
$\cong \int_{X:\mathrm{Set}} (A \to X) \to G^F(X)$	covariant Yoneda lemma
$\cong [\underline{\mathbf{Set}}, \underline{\mathbf{Set}}]((A \to _), G^F)$	natural transformations are morphisms in $[\underline{\mathbf{Set}},\underline{\mathbf{Set}}]$
$\cong [\underline{\mathbf{Set}}, \underline{\mathbf{Set}}]((A \to _) \times F, G)$	uncurrying: since we assume the exponential G^F exists, we have $[\underline{\mathbf{Set}}, \underline{\mathbf{Set}}](X, Z^Y) \cong [\underline{\mathbf{Set}}, \underline{\mathbf{Set}}](X \times Y, Z)$
$\cong \int_{X:\mathrm{Set}} (A \to X) \times F(X) \to G(X)$	morphisms in $[\underline{\mathbf{Set}},\underline{\mathbf{Set}}]$ are natural transformations
$\cong \int_{X:\mathrm{Set}} (A \to X) \to F(X) \to G(X)$	currying: since $\underline{\mathbf{Set}}$ has exponentials, we have $\underline{\mathbf{Set}}(X \times Y, Z) \cong \underline{\mathbf{Set}}(X, Z^Y) \cong \underline{\mathbf{Set}}(X, Y \to Z)$

Since $G^F(A)$ must be of type Set (size wise, i.e. it must be of type Set_0) and it is isomorphic to $\int_{X:\mathrm{Set}}(A \to X) \to F(X) \to G(X)$, then the latter must also have type Set. However, there are cases where the homset from a functor F to a functor G is larger than a Set. For example, in the category of classical sets, $\int_{X:\mathrm{Set}} \mathcal{P}(X) \to \mathcal{P}(\mathcal{P}(X))$, where $\mathcal{P}$ is the covariant powerset functor, is not a Set.

It was already shown in Abbott et al. (2005) that containers are closed under exponentiation by constant containers. A *constant container* is of the form $(K \triangleleft 0)$ whose extension functor is a constant functor equal to K. Abbott et al. (2005) computed $K \to [\![F]\!]\, X$ in $[\underline{\mathbf{Set}},\underline{\mathbf{Set}}]$ and then reflected it along $[\![_]\!]$ to define

$$F^K = (f : K \to S) \triangleleft (\sum(k : K)(P(f\, k)))$$

for container $F = S \triangleleft P$ and constant container K.

Altenkirch et al. (2010) extended on this result by showing that $\underline{\mathbf{Cont}}$ has

all exponentials, thereby proving that it is Cartesian closed. The proof goes as follows.

We first note that exponentiating with a container having shape **1** (i.e. a representable functor) is straightforward. Consider F in [**Set**,**Set**] and P, X : Set.

$(F^{[\![\mathbf{1} \triangleleft P]\!]})\, X$	
$\cong \int_{Y:\mathrm{Set}} (X \to Y) \to [\![\mathbf{1} \triangleleft P]\!] Y \to F\, Y$	calculation above of exponential
$\cong \int_{Y:\mathrm{Set}} (X \to Y) \to (P \to Y) \to F\, Y$	definition of container functor
$\cong \int_{Y:\mathrm{Set}} (X \to Y) \times (P \to Y) \to F\, Y$	uncurrying in **Set**
$\cong \int_{Y:\mathrm{Set}} (X + P \to Y) \to F\, Y$	$(X \to Y) \times (P \to Y) \cong (X + P \to Y)$
$\cong F(X + P)$	covariant Yoneda lemma

$F(X + P)$ is indeed a Set since F is an endofunctor on **Set**, so this computation is valid.

We also note that

$[\![S \triangleleft P]\!]\, X = \sum(s : S)(P\, s \to X)$	
$\cong \sum(s : S)([\![\mathbf{1} \triangleleft P\, s]\!]\, X)$	
$\cong [\![\sum(s : S)(\mathbf{1} \triangleleft P\, s)]\!]\, X$	coproduct of arbitrarily many container functors is closed under $[\![_]\!]$: $\sum(i : I)[\![(s : S_i) \triangleleft P_i\, s]\!]\, X \cong [\![\sum(i : I)((s : S_i) \triangleleft P_i\, s)]\!]\, X$

and since $[\![_]\!]$ is full and faithful, we can reflect along it to obtain an isomorphism betwen $S \triangleleft P$ and the coproduct of containers S (which can be

written as a container $(S \lhd 0))$ and $\mathbf{1} \lhd P\, s$

$$S \lhd P \cong \sum(s : S)(\mathbf{1} \lhd P\, s).$$

So now if $F \colon \underline{\mathbf{Set}} \to \underline{\mathbf{Set}}$, we can calculate

$F^{[\![S \lhd P]\!]}$	
$\cong F^{[\![\sum(s:S)(\mathbf{1}) \lhd P\, s]\!]}$	isomorphism above
$\cong F^{\sum(s:S)[\![\mathbf{1} \lhd P\, s]\!]}$	closure of coproducts under $[\![_]\!]$
$\cong \Pi(s : S)(F^{[\![\mathbf{1} \lhd P\, s]\!]})$	$C^{\sum(i:I)(A_i)} \cong \Pi(i : I)(C^{A_i})$ (a generalisation of $C^{A+B} \cong C^A \times C^B$ to i many) in [$\underline{\mathbf{Set}}$,$\underline{\mathbf{Set}}$]
$\cong \Pi(s : S)(F(_ + P\, s))$	result above of exponentiating with a container having shape $\mathbf{1}$

where $F(_ + P\, s)$ is defined by $(F(_ + P\, s))\, X = F(X + P\, s)$.

Hence $F^{[\![S \lhd P]\!]} \cong \Pi(s : S)(F(_ + P\, s))$, and when $F \colon \underline{\mathbf{Set}} \to \underline{\mathbf{Set}}$ is a container functor i.e. of the form $[\![T \lhd Q]\!]$, the right hand side is a container functor by the closure properties of container functors seen previously in this section. Hence we have the following result.

Theorem 4.13. $[\![_]\!]$ preserves exponentiation on functors of type $\underline{\mathbf{Set}} \to \underline{\mathbf{Set}}$.

We can reflect exponentiation in $\underline{\mathbf{Set}} \to \underline{\mathbf{Set}}$ to $\underline{\mathbf{Cont}}$ to obtain a definition for exponentiation of containers. When expanding the definition above for $F = T \lhd Q$, we get

$$\begin{aligned} T \lhd Q^{S \lhd P} = (&\textstyle\sum(f \colon S \to T)(g \colon (s : S) \to Q(f\, s) \to \mathbf{1} + P\, s)) \lhd \\ &(\textstyle\sum(s : S)(\sum(q : Q(f\, s))(g\, s\, q = \mathrm{inl}\ \mathrm{tt} \to \bot))) \end{aligned}$$

Given that $\underline{\mathbf{Cont}}$ has a terminal object $\mathbf{1} \lhd \mathbf{0}$, products, and exponentials, this makes $\underline{\mathbf{Cont}}$ a Cartesian closed category. Altenkirch et al. (2010) show that it is not locally Cartesian closed. However, contrary to what is said in the paper, this does not mean that the category of containers cannot be a

model of type theory. Indeed, the category of setoids is not locally Cartesian closed, however it is still a model of type theory. Local Cartesian closure is stronger of a requirement than we need for a category to be a model of type theory.

So far, we have been looking at the simplest version of containers—unary containers, the ones having functors of type $\underline{\mathbf{Set}} \to \underline{\mathbf{Set}}$ representing inductive types with at most 1 type variable (like $\mathbb{N}$ with 0 type variables, `List A` with 1 type variable). In order to move forward and in particular to be able to reason about fixed points of containers, we need to define containers with multiple parameters. Before we do this, we summarise what we have so far.

Going back to our original goal of representing strictly positive inductive types by containers, from the results and closure properties of $\underline{\mathbf{Cont}}$ that we have seen so far, we can give an interpretation of any non-inductive strictly positive type in 1 variable as a container.

Theorem 4.14. Every non-inductive strictly positive type in 1 variable can be interpreted as a container.

Proof. Let K be a constant type (i.e. has no type variables) and let F and G be strictly positive inductive types represented by containers $(S \triangleleft P)$ and $(T \triangleleft Q)$ respectively. Then we have the following container interpretations.

$$\begin{aligned}
K &\mapsto (K \triangleleft 0) \\
F + G &\mapsto ((S + T) \triangleleft (\lambda\,(\mathrm{inl}\, s) \to P\, s; (\mathrm{inr}\, t) \to Q\, t)) \\
F \times G &\mapsto (((s, t) : S \times T) \triangleleft (P\, s + Q\, t)) \\
K \to F &\mapsto ((f : K \to S) \triangleleft (\textstyle\sum (k : K)(P(f\, k))))
\end{aligned}$$

□

The next section details how we can extend theorem 4.14 to the full range of strictly positive types (definition 4.1), i.e. give a container interpretation of types in n variables and types that are inductive.

4.1.4 Initial Algebras and Terminal Coalgebras

An important property of the category of containers is that it is closed under taking the least and greatest fixed points. This allows us to represent inductive and coinductive types using containers. In order to be able to talk about fixed points, we introduce a slightly more general kind of container, one with multiple parameters.

We now redefine the category of containers and the container extension functor for containers in multiple parameters, or *n-ary containers*. These 'new' definitions are very close to the previous ones, only with minor adjustments to account for an indexing set, hence we overload the notation used previously. Similarly, the constructions in section 4.1.3 still hold for these containers, with a few adjustments.

Definition 4.15. Given an index set I, the *category of containers in I parameters*, which we denote $\underline{\mathbf{Cont}}_I$, is defined as follows:

- Objects are pairs $S : \mathsf{Set}$ and $P \colon I \to S \to \mathsf{Set}$, written as $S \lhd P$.
- Morphisms $(S \lhd P) \to (T \lhd Q)$ are pairs $u \colon S \to T$ and $f \colon (s : S) \to (Q\, i\, (u\, s)) \to (P\, i\, s)$. ■

Definition 4.16. The *container extension functor* $[\![_]\!] \colon \underline{\mathbf{Cont}}_I \to [\underline{\mathbf{Set}}^n, \underline{\mathbf{Set}}]$ (where $n = |I|$) is defined as follows:

- Given an object $(S \lhd P)$ in $\underline{\mathbf{Cont}}_I$, this is mapped to the functor $[\![S \lhd P]\!] \colon \underline{\mathbf{Set}}^n \to \underline{\mathbf{Set}}$ defined below.
  - Given an object $\boldsymbol{X} : \mathsf{Set}^n$, $[\![S \lhd P]\!]\, \boldsymbol{X} \coloneqq \sum(s : S)((i : I) \to P\, i\, s \to X_i)$.
  - Given $\boldsymbol{X}, \boldsymbol{Y} : \mathsf{Set}^n$, a morphism $f \colon \boldsymbol{X} \to \boldsymbol{Y}$, $s : S$, and $g \colon (i : I) \to P\, i\, s \to X_i$,

$$[\![S \lhd P]\!]\, f\, (s, g) \coloneqq (s, \lambda\, i \to f_i \circ (g\, i)).$$

- Given a morphism $(u \lhd f) \colon (S \lhd P) \to (T \lhd Q)$ in $\underline{\mathbf{Cont}}_I$, this is mapped to the natural transformation $[\![u \lhd f]\!] \colon [\![S \lhd P]\!] \to [\![T \lhd Q]\!]$ with components

$[\![u \lhd f]\!]_{\boldsymbol{X}} \colon [\![S \lhd P]\!]\, \boldsymbol{X} \to [\![T \lhd Q]\!]\, \boldsymbol{X}$ for any $\boldsymbol{X} : \mathsf{Set}^n$, defined as

$$[\![u \lhd f]\!]_{\boldsymbol{X}}(s, h) \coloneqq (u\, s, \lambda\, i\, q \to h\, i\, (f_i\, s\, q)),$$

for $s : S$ and $h \colon (i : I) \to Q\, i\, (u\, s) \to P\, i\, s$. ■

Here we consider I to be a finite set, but it is possible to generalise these developments to the infinite case.

Having defined the above, we can now get back to thinking about fixed points of containers. We want to show that if $F(\boldsymbol{X}, Y)$ is a container functor $\underline{\mathbf{Set}}^{n+1} \to \underline{\mathbf{Set}}$ for $\boldsymbol{X} : \mathsf{Set}^n$ and $Y : \mathsf{Set}$, then $\mu Y.F(\boldsymbol{X}, Y)$ and $\nu Y.F(\boldsymbol{X}, Y)$ are also container functors $\underline{\mathbf{Set}}^n \to \underline{\mathbf{Set}}$. μ and ν are partial operators taking a functor F to the object part of its initial algebra μF or terminal coalgebra νF, if they exist (this was discussed in more detail in section 2.4.5).

Consider F in $\underline{\mathbf{Cont}}_{I+1}$. Then F can be written as $(S \lhd P, Q)$ for $S : \mathsf{Set}$, $P \colon I \to S \to \mathsf{Set}$, and $Q \colon S \to \mathsf{Set}$, and has extension

$$[\![F]\!](\boldsymbol{X}, Y) = \sum (s : S)((i : I) \to P\, i\, s \to X_i) \times (Q\, s \to Y).$$

To show $\mu Y.F(\boldsymbol{X}, Y)$ and $\nu Y.F(\boldsymbol{X}, Y)$ are container functors with respect to $\boldsymbol{X}$, we need to compute $(A_\mu \lhd B_\mu)$ and $(A_\nu \lhd B_\nu)$ such that $\mu Y.[\![F]\!](\boldsymbol{X}, Y) \cong [\![A_\mu \lhd B_\mu]\!]\, \boldsymbol{X}$ and $\nu Y.[\![F]\!](\boldsymbol{X}, Y) \cong [\![A_\nu \lhd B_\nu]\!]\, \boldsymbol{X}$.

Computing A_μ and A_ν is relatively straightforward. Note that since A_μ is the shape of the container $(A_\mu \lhd B_\mu)$, we can write it as $[\![A_\mu \lhd B_\mu]\!]\, 1 = \sum (a : A_\mu)(B_\mu a \to 1) \cong A_\mu$. So we have

$$\begin{aligned} A_\mu &\cong [\![A_\mu \lhd B_\mu]\!]\, 1 \\ &\cong \mu Y.[\![F]\!](1, Y) \\ &\cong \mu Y. \sum (s : S)(Q\, s \to Y) \\ &= \mu Y.[\![S \lhd Q]\!]\, Y \\ &\cong W\, S\, Q. \end{aligned}$$

Similarly we get

$$A_\nu \cong M\, S\, Q.$$

Now we need to compute $W\, S\, Q \vdash B_\mu$ and $M\, S\, Q \vdash B_\nu$. We show the

process for computing B_μ. In the rest of this construction, for the sake of simplicity we ignore the index set I and consider $P\colon S \to \mathsf{Set}$, when we should strictly speaking consider $P\colon I \to S \to \mathsf{Set}$ (equivalently, we will be assuming $I = 1$). It is straightforward to generalise the construction here to the setting where we have more than 1 parameter, i.e. $I \neq 1$.

Let $G = (A \triangleleft B)$ in $\underline{\mathbf{Cont}}_I$. Then we can compose the container functors of F and G, denoted by $[\![F]\!][[\![G]\!]]$, as follows.

$$\underline{\mathbf{Set}}^I \xrightarrow{(\mathrm{id},[\![G]\!])} \underline{\mathbf{Set}}^I \times \underline{\mathbf{Set}} \cong \underline{\mathbf{Set}}^{I+1} \xrightarrow{[\![F]\!]} \underline{\mathbf{Set}}$$

We can then lift this to a functor on containers $-[-]\colon \underline{\mathbf{Cont}}_{I+1} \times \underline{\mathbf{Cont}}_I \to \underline{\mathbf{Cont}}_I$ defined by:

$$\begin{aligned} F[G] &= (S \triangleleft P, Q)[A \triangleleft B] \\ &\coloneqq (s : S)(f\colon Q\, s \to A) \triangleleft (P\, s + \sum (q : Q\, s)(B(f\, q))). \end{aligned}$$

We need such a G to be the required fixed point for F, i.e. we would like to find a G such that $F[G] \cong G$. Now observe that an isomorphism $\psi\colon [\![S \triangleleft Q]\!]\, A \cong A$ induces an isomorphism $F[G] \cong G$. This is because

$$[\![F[G]]\!]\, 1 \cong (s : S)(f\colon Q\, s \to A) = [\![S \triangleleft Q]\!]\, A$$

and $[\![G]\!]\, 1 \cong A$, hence ψ is an isomorphism of shapes of $F[G]$ and G, i.e. $\psi\colon [\![F[G]]\!]\, 1 \cong [\![G]\!]\, 1$. Then what does an isomorphism of positions of $F[G]$ and G look like? It is a family of isomorphisms

$$s : S, f\colon Q\, s \to A \vdash \phi_{s,f} : (P\, s + \sum (q : Q\, s)(B(f\, q))) \cong B(\psi_{s,f}).$$

So an isomorphism between $F[G]$ and G must be of the form

$$(\psi, \phi^{-1})\colon F[G] \to G$$

(we had to invert ϕ due to the definition of container morphisms, see definition 4.5).

In order to obtain such an isomorphism, we will need a pair (B, ϕ) to be an *initial family* over ψ, i.e. a family $a : A_\mu \vdash B(a)$ equipped with a morphism

$\phi_{s,f} : (P\,s + \sum(q : Q\,s)(B(f\,q))) \cong B(\psi_{s,f})$ as above, which is initial in the category of such families and morphisms. Then by Lambek's lemma, the morphism ϕ we obtain will be an isomorphism.

While we do not go into the proof's details here, Abbott et al. (2005) show that given a container $F = (S \lhd P, Q)$ in $\underline{\mathbf{Cont}}_{I+1}$, $A : \mathsf{Set}$, and a fixed point $\psi\colon [\![S \lhd Q]\!]\, A \cong A$, there exists an initial family $(A \lhd \mathrm{Pos}_\psi)$ over ψ, and $\mathrm{Pos}_\psi\,(\sup s\, f) = P\,s + \sum(q : Q\,s)(\mathrm{Pos}_\psi\,(f\,q))$.

Hence we set $B_\mu\,(\sup s\, f)$ to $\mathrm{Pos}_\psi = P\,s + \sum(q : Q\,s)(B_\mu(f\,q))$, and we get one of the two main results.

Theorem 4.17. Given a container $F = (S \lhd P, Q)$ in $\underline{\mathbf{Cont}}_{I+1}$, we have that

$$[\![W\,S\,Q \lhd Pos_\psi]\!]\,\boldsymbol{X} \cong \mu Y.[\![F]\!](\boldsymbol{X}, Y).$$

Moreover, writing $\mu F \coloneqq W\,S\,Q \lhd Pos_\psi$, we get that $\mu[\![F[-]]\!] \cong [\![\mu F]\!]$, so by reflection along $[\![_]\!]$ we get that $\underline{\mathbf{Cont}}_I$ is closed under least fixed points.

Computing B_ν is slightly more involved but we get a similar result, shown below.

Theorem 4.18. Given a container $F = (S \lhd P, Q)$ in $\underline{\mathbf{Cont}}_{I+1}$, we have that

$$[\![M\,S\,Q \lhd Pos_\psi]\!]\,\boldsymbol{X} \cong \nu Y.[\![F]\!](\boldsymbol{X}, Y).$$

Moreover, writing $\nu F \coloneqq M\,S\,Q \lhd Pos_\psi$, we get that $\nu[\![F[-]]\!] \cong [\![\nu F]\!]$, so by reflection along $[\![_]\!]$ we get that $\underline{\mathbf{Cont}}_I$ is closed under greatest fixed points.

Example 4.19. We will use the constructions discussed in this section to work out the container functor of the `List` data type.

An inductive type is the initial algebra over a functor representing its sort(s) and constructor(s). To this end, the functor representing the sort and constructors of `List` is

$$L\,A\,X = 1 + A \times X,$$

where A is the type of entries of the list. The `List` data type itself is

represented by the fixed point of this functor, namely

$$L_\mu\, A = \mu X.(1 + A \times X).$$

Our aim is to show that L_μ is a container functor with respect to A, by coming up with an A_μ and a B_μ such that

$$L_\mu\, A = \mu X.[\![L]\!](A, X) \cong [\![A_\mu \triangleleft B_\mu]\!]\, A.$$

Following our general construction, we write L as the container $(S \triangleleft P, Q)$ in $\underline{\mathbf{Cont}}_2$, with S, P, and Q being the following.

$$\begin{aligned}
&S : \mathsf{Set}\\
&S = \top + \top\\[1em]
&P\colon S \to \mathsf{Set}\\
&P(\text{inl tt}) = \bot\\
&P(\text{inr tt}) = \top\\[1em]
&Q\colon S \to \mathsf{Set}\\
&Q(\text{inl tt}) = \bot\\
&Q(\text{inr tt}) = \top
\end{aligned}$$

S represents the possible constructors one can choose from, P tells us how many As a constructor has, and Q tells us how many Xs a constructor has.

We set A_μ to be $W\,S\,Q$, which is

$$\begin{aligned}
W\,S\,Q &= W(\top + \top)(\lambda\,\{(\text{inl tt}) \to \bot; (\text{inr tt}) \to \top\})\\
&\cong W(\text{Bool})(\lambda\,\{(\text{false}) \to \bot; (\text{true}) \to \top\})\\
&\cong \mathbb{N}.
\end{aligned}$$

So A_μ is set to $\mathbb{N}$, with constructors `sup (inl tt) (λ ())` (for `zero`) and `sup (inr tt) (λ {tt →` n`})` for n of type $W\,S\,Q$ (for `succ`).

Now we set $B_\mu\,(\texttt{sup s f})$ to $P\,s + \sum(q : Q\,s)(B_\mu(f\,q))$. We compute:

$$\begin{aligned}
B_\mu\,(\texttt{sup (inl tt) } (\lambda\ ())) &= P(\mathrm{inl\ tt}) + \sum(q : Q(\mathrm{inl\ tt}))(B_\mu(\lambda()))\\
&\cong \bot + \bot\\
&\cong \bot\\
B_\mu\,(\texttt{sup (inr tt) } (\lambda\,\texttt{tt} \to n)) &= P(\mathrm{inr\ tt}) + \sum(q : Q(\mathrm{inr\ tt}))(B_\mu((\lambda\,\{\mathrm{tt} \to n\})\,q))\\
&\cong \top + \top \times B_\mu(n)\\
&\cong \top + B_\mu(n)
\end{aligned}$$

So $B_\mu\,(\mathrm{zero}) = \bot$ and $B_\mu\,(\mathrm{succ\ n}) = \top + B_\mu(n)$. Hence $B_\mu \cong \mathrm{Fin}$.

Hence we have that $L_\mu\,A = \mu X.[\![L]\!](A, X) \cong [\![(n : \mathbb{N}) \lhd (\mathrm{Fin}\,n)]\!]A$.

We end this section with our initial motivation—an extended version of theorem 4.14 covering the full range of strictly positive types.

Theorem 4.20. Every strictly positive inductive type in n variables can be interpreted as a container.

Proof. We already saw how to interpret constant types, products, coproducts, and arrow types in theorem 4.14. The remaining cases needed to cover all the cases of definition 4.1 are given below.

Let F be a strictly positive inductive type represented by the container $(S \lhd P, Q)$, and let X_i be a type variable. Then we have the following container interpretations.

$$\begin{aligned}
X_i &\mapsto (1 \lhd (\mathrm{if}\,i = j\,\mathrm{then}\,\top\,\mathrm{else}\,\bot))\\
\mu X.F &\mapsto (W\,S\,Q \lhd Pos_\psi)\\
\nu X.F &\mapsto (M\,S\,Q \lhd Pos_\psi)
\end{aligned}$$

□

4.1.5 Generalisations

In this section, we primarily looked at unary containers, which allow one parameter and represent functors on the category of sets, $\underline{\mathbf{Set}} \to \underline{\mathbf{Set}}$:

$$F_{\mathbb{N}} : \mathsf{Set} \to \mathsf{Set}$$
$$F_{\mathbb{N}}\, X := 1 + X,$$

as well as n-ary containers allowing a finite number n of parameters, representing functors $\underline{\mathbf{Set}}^n \to \underline{\mathbf{Set}}$:

$$F_{\mathrm{List}} : (A : \mathsf{Set}) \to \mathsf{Set} \to \mathsf{Set}$$
$$F_{\mathrm{List}}\, A\, X := 1 + A \times X.$$

Further generalisations of containers exist in the literature. One such generalisation is *indexed containers* (Altenkirch and Morris (2009), Altenkirch et al. (2015)), which represent functors on the category of families indexed by a (possibly infinite) indexing type I (see section 2.4.1), $(\underline{\mathbf{I}} \to \underline{\mathbf{Set}}) \to (\underline{\mathbf{I}} \to \underline{\mathbf{Set}})$, used for inductive families:

$$F_{\mathrm{Vec}} : (A : \mathsf{Set}) \to (\mathbb{N} \to \mathsf{Set}) \to (\mathbb{N} \to \mathsf{Set})$$
$$F_{\mathrm{Vec}}\, A\, X\, n := (n \equiv 0) + (m : \mathbb{N}) \times (n \equiv \mathrm{suc}\, m) \times (A \times X\, m).$$

Indexed containers generalise ordinary and n-ary containers presented in this section, and can be seen as the type theoretic equivalent of dependent polynomial functors (Gambino and Hyland, 2004). They were shown to be a normal form for strictly positive families in much the same way as containers are a normal form for strictly positive types.

Indexed containers are themselves a special case of *generalised containers*, which represent representable functors over an arbitrary category $\underline{\mathbf{C}}$, having the form $\underline{\mathbf{C}} \to \underline{\mathbf{Set}}$ (to get indexed containers, set $\underline{\mathbf{C}} = (\underline{\mathbf{I}} \to \underline{\mathbf{Set}}) \times \underline{\mathbf{I}}$). A generalised container is written as $S \triangleleft P$ with $S : \mathsf{Set}$ and $P : S \to |\underline{\mathbf{C}}|$. Such a container functor would be defined by $[\![S \triangleleft P]\!]\, X := \sum(s : S)(\underline{\mathbf{C}}(P\, s, X))$ for $X : |\underline{\mathbf{C}}|$. Generalised containers are the kind of containers we will need for the 'containerification' of the semantics of (Q)IITs.

In summary, we have the following types of containers, starting from the

least general to the most general.

container type	container functor type	example
ordinary	$\underline{\mathbf{Set}} \to \underline{\mathbf{Set}}$	$\mathbb{N}$: Set
n-ary	$\underline{\mathbf{Set}}^n \to \underline{\mathbf{Set}}$	List A : Set
indexed	$(\underline{\mathbf{I}} \to \underline{\mathbf{Set}}) \to (\underline{\mathbf{I}} \to \underline{\mathbf{Set}})$	Vec A: $\mathbb{N} \to$ Set
generalised	$\underline{\mathbf{C}} \to \underline{\mathbf{Set}}$	representable functors

4.2 Models of type theory

A type theory is a formal system in which we can derive certain kinds of judgments. Some examples of type theories are the simply-typed lambda calculus, the calculus of constructions, and Martin-Löf type theory. We define a type theory by listing its kinds of judgments and their syntax, and listing the derivation rules that can be used in proofs (or derivations) of the judgments.

Example 4.21. We have the following two judgments in Martin-Löf type theory.

$\Gamma \vdash$	Γ is a valid context
$\Gamma \vdash A$	A is a valid type in context Γ

These judgments are used in the derivation rules for forming new contexts, among many other derivation rules.

$$\frac{}{\diamond \vdash}\ \text{emp} \qquad\qquad \frac{\Gamma \vdash \quad \Gamma \vdash A}{\Gamma . A \vdash}\ \text{comp}$$

Developing a notion of semantics for dependent type theories is desirable mainly because it is easier to show that a mathematical structure is a type theory by proving it is an instance of this semantics, than by formulating an interpretation function for it directly. To this end, many different notions of a model of type theory have been proposed over the years, such as Cartmell's contextual categories (Cartmell, 1986), Jacobs's comprehension categories (Jacobs, 1993), and Dybjer's categories with families or CwFs (Dybjer, 2003). A model constitutes a sound semantics for a type theory,

i.e. an interpretation of the type theory such that any judgment that can be derived in the type theory can also be derived in the semantics.

We present the notion of model that we are interested in, namely CwFs. We feel that this notion of model is the most naturally related to the syntax of type theory, and is less categorical than other notions. Our definition and discussion are adapted from Dybjer (2003), Hofmann (1997), and Kaposi (2013).

Definition 4.22. A *category with families* (CwF) consists of:

- A category $\underline{\mathbf{C}}$ whose objects interpret contexts and whose morphisms interpret context morphisms, having a terminal object interpreting the empty context.
- A functor $T\colon \underline{\mathbf{C}}^{\mathrm{op}} \to \underline{\mathbf{Fam}}$ that interprets types and terms. If $\Gamma : |\underline{\mathbf{C}}|$, then we write
$$T(\Gamma) = (Ty(\Gamma), Tm(\Gamma, \cdot)).$$
If $\gamma\colon \Gamma \to \Delta$, then
$$T(\gamma)\colon (Ty(\Delta), Tm(\Delta, \cdot)) \to (Ty(\Gamma), Tm(\Gamma, \cdot))$$
$$T(\gamma)(A, a) = (A[\gamma], a[\gamma])$$
where $A[\gamma]$ interprets type substitution and $a[\gamma]$ interprets term substitution.
- A context comprehension operation which associates to every $\Gamma : |\underline{\mathbf{C}}|$ and $A : Ty(\Gamma)$
  - an object $\Gamma.A : |\underline{\mathbf{C}}|$
  - a morphism $p\colon \Gamma.A \to \Gamma$ in $\underline{\mathbf{C}}$,
  - and a term $q : Tm(\Gamma.A, A[p])$,

  such that given a context $\Delta : |\underline{\mathbf{C}}|$, a context morphism $\gamma\colon \Delta \to \Gamma$, and a term $a : Tm(\Delta, A[\gamma])$, there exists a unique context morphism $\theta\colon \Delta \to \Gamma.A$, denoted by $\theta = \langle \gamma, a \rangle$, such that $p \circ \theta = \gamma$ and $q[\theta] = a$. ■

We note that there are several equivalent ways of defining a CwF. One such alternative is that instead of defining one functor $T\colon \underline{\mathbf{C}}^{\mathrm{op}} \to \underline{\mathbf{Fam}}$

to interpret types and terms, we define two functors $Ty\colon \underline{\mathbf{C}}^{\mathrm{op}} \to \underline{\mathbf{Set}}$ and $Tm\colon (\int Ty)^{\mathrm{op}} \to \underline{\mathbf{Set}}$. This is the approach used in section 5.1.

The definition of CwFs encapsulates all the rules of type theory for context formation and substitution and congruence of definitional equality.

Example 4.23. The set model of type theory is defined as follows.

- The base category $\underline{\mathbf{C}}$ is set to the category of sets and functions $\underline{\mathbf{Set}}$, so that contexts are sets and context morphisms are functions. The terminal object is the set with one element $\{*\}$ or $\mathbf{1}$, and this represents the empty context.
- The functor $Ty\colon \underline{\mathbf{Set}}^{\mathrm{op}} \to \underline{\mathbf{Set}}$ is defined on objects and morphisms as

$$\begin{aligned} Ty(\Gamma)&\colon \mathrm{Set} \\ Ty(\Gamma) &\coloneqq \Gamma \to \mathrm{Set} \end{aligned}$$

$$\begin{aligned} Ty(\Delta \xrightarrow{\sigma} \Gamma)&\colon Ty(\Gamma) \to Ty(\Delta) \\ Ty(\Delta \xrightarrow{\sigma} \Gamma)\, A &\coloneqq A \circ \sigma, \end{aligned}$$

  with $A \circ \sigma\colon Ty(\Delta) = \Delta \to \mathrm{Set}$.
- The functor $Tm\colon (\int Ty)^{\mathrm{op}} \to \underline{\mathbf{Set}}$ is defined on objects as

$$\begin{aligned} Tm(\Gamma, A)&\colon \mathrm{Set} \\ Tm(\Gamma, A) &\coloneqq (\gamma : \Gamma) \to A\,\gamma. \end{aligned}$$

  Before we define Tm on morphisms, we look at the objects and morphisms of the category $(\int Ty)^{\mathrm{op}}$.

  - Objects are pairs $(\Gamma : \mathrm{Set}, Ty(\Gamma))$.
  - A morphism $(\Gamma, A) \to (\Delta, B)$ is a morphism $f\colon \Delta \to \Gamma$ and a proof $p : (Ty\, f)\, A = B$, and by the definition of Ty on morphisms, this reduces to $p : A \circ f = B$.

  So for $f\colon \Delta \to \Gamma$ and $p : A \circ f = B$, Tm is defined on morphisms as

$$\begin{aligned} Tm((\Gamma, A) \xrightarrow{(f,p)} (\Delta, B))&\colon Tm(\Gamma, A) \to Tm(\Delta, B) \\ Tm((\Gamma, A) \xrightarrow{(f,p)} (\Delta, B))\, a &\coloneqq \lambda\delta \to a(f\,\delta), \end{aligned}$$

where $a \colon (\gamma : \Gamma) \to A\,\gamma$, and $\delta : \Delta$. $a(f\,\delta)$ is of type $A\,(f\,\delta) = (A \circ f)\,\delta = B\,\delta$ by p, as required.

Note: Alternatively, we could view morphisms in $(\int Ty)^{\mathrm{op}}$ as $f \colon \Delta \to \Gamma$ of type $(\Gamma, A) \to (\Delta, A \circ f)$. Then

$$Tm((\Gamma, A) \xrightarrow{f} (\Delta, A \circ f)) \colon Tm(\Gamma, A) \to Tm(\Delta, A \circ f)$$
$$Tm((\Gamma, A) \xrightarrow{f} (\Delta, A \circ f))\, a \coloneqq \lambda\delta \to a(f\,\delta),$$

with $a(f\,\delta) : (A \circ f)\,\delta$ as required.

- The context comprehension operation associates to a Γ : Set and an $A : Ty(\Gamma)$ the set $\Gamma.A$ of pairs $(\rho : \Gamma, u : A\rho)$. Then $p \colon \Gamma.A \to \Gamma$ is defined by the first projection out of $\Gamma.A$, and $q : Tm(\Gamma.A, Ap)$ by the second projection.

5

Future Work Plan

In chapter 3, I surveyed the most relevant literature to my thesis and identified three main goals for the course of my PhD. I will restate these goals here, and provide more details about each one.

My primary aim is to contribute towards the long-term goal of providing semantics for inductive types in HoTT, called HITs. To this end, a realistic starting point is considering a subset of HITs, namely those types that are defined with their equalities, but whose higher equalities are quotiented away, known as quotient inductive types (QITs). In this sense these types are not very 'high'. Many meaningful examples of such set-truncated types also allow the use of induction-induction (e.g. Cauchy reals, surreal numbers, syntax of type theory in type theory). Therefore we are more generally interested in investigating QITs with the added power of induction-induction, called quotient inductive-inductive types (QIITs).

In order to provide a general semantics and theoretical foundation for QITs and QIITs, I plan on working on the following subprojects.

1. Construct a container model of type theory
2. Refine an existing initial algebra semantics for QIITs by 'containerification', so that it only permits strictly positive QIITs
3. Specify a syntax for QIITs arising from the above semantics

5.1 Container Model of Type Theory

For (1), I plan on formalising a container model of type theory using categories with families (CwFs). Some details and an incomplete Agda formalistion have already been set out in an abstract by Altenkirch and Kaposi

(2021), however there are still a number of issues that need to be resolved. One issue is that their construction does not yet take into account h-levels, and does not view contexts and types as h-sets, which would be required for this model. Doing this requires a generalisation of CwFs which they call coherent CwFs, where types are allowed to be groupoids (as opposed to just sets), but this notion has also not been fully formalised. Our first objective is therefore to work out the full details required for a complete formalisation of the container model. As a starting point towards our CwF, we give some of the most important definitions involved based on the above mentioned abstract.

- We set the base category $\underline{\textbf{Con}}$ with objects being contexts and morphisms being substitutions to the category of containers $\underline{\textbf{Cont}}$. So contexts are interpreted as containers $S \triangleleft P$ for $S : \mathsf{Set}$ and $P \colon S \to \mathsf{Set}$, and substitutions are interpreted as container morphisms $u \triangleleft f$ as presented in section 4.1.

- To interpret types in a given context, we provide the functor

  $$\begin{gathered} \mathrm{Ty} \colon \underline{\textbf{Con}} \to \underline{\textbf{Set}}_1 \\ \mathrm{Ty}\,\Gamma := \textstyle\int [\![\Gamma]\!] \to \mathrm{Set} \end{gathered}$$

  of dependent containers. Given a context Γ, a type A is interpreted as a generalised container whose container functor is of type

  $$\textstyle\int [\![\Gamma.S \triangleleft \Gamma.P]\!] \to \underline{\textbf{Set}}.$$

  If we recall the type of objects of the category of elements (see section 2.4.2), we know that

  $$|\textstyle\int [\![\Gamma.S \triangleleft \Gamma.P]\!]| = (X : \mathsf{Set}) \times ((s : \Gamma.S) \times (\Gamma.P\, s \to X)).$$

  So a type A in context Γ interpreted as the container $\Gamma.S \triangleleft \Gamma.P$, will consist of components $A.S : \mathsf{Set}$ and $A.P \colon A.S \to |\textstyle\int [\![\Gamma.S \triangleleft \Gamma.P]\!]|$, which is in turn made up of three components, each one a mapping from $A.S$ to a corresponding entry in the product type. (Note that a type therefore depends on both the shapes ($\Gamma.S$) and the positions ($\Gamma.P$) of its context Γ, which is a different approach to existing container

models of type theory in the literature.)

- To interpret terms of a given type in a given context, we provide the functor

$$
\begin{aligned}
&\mathrm{Tm} \colon \textstyle\int \mathrm{Ty} \to \underline{\mathbf{Set}} \\
&\mathrm{Tm}\,(\Gamma : \mathrm{Con})(A : \mathrm{Ty}\,\Gamma) \coloneqq (X : \mathrm{Set})(x : [\![\Gamma]\!]\, X) \to A(X, x) \\
&\qquad = \int_{X:\mathsf{Set}} (x : [\![\Gamma]\!] X) \to A(X, x)
\end{aligned}
$$

of dependent container morphisms i.e. dependent natural transformations.

More definitions are required to specify a CwF, but we will stop here for the scope of this report.

5.2 Containerification of Semantics for (Q)IITs

For (2), I will improve on existing work by Altenkirch et al. (2018) to provide an initial algebra semantics for QIITs. Initial algebra semantics for inductive types and inductive families is already well established. This is partially due to the fact that it is relatively straightforward to specify these types via endofunctors (see $F_{\mathbb{N}}, F_{\mathrm{List}}$, and F_{Vec} in section 4.1.5). The same cannot be said for IITs. Consider the IIT of contexts and types within that context, as defined in section 2.3. The type `Con` and the type family `Ty` are defined simultaneously and depend on each other mutually. Due to this high degree of dependency in IITs, we cannot specify such a type via an endofunctor and obtain initial algebra semantics that way. However, we still know what an algebra of this type would look like, namely a record type with entries for the sorts and constructors.

```
record ConTyAlg : Type₁ where
  field
    -- sorts
    C : Type                    -- Con
    T : C → Type                -- Ty
    -- constructors
    emp-con : C                 -- ◇
```

```
ext-con : (Γ : C) → T Γ → C -- _,_
emp-ty : (Γ : C) → T Γ        -- ι
sig-ty : (Γ : C) (A : T Γ) (B : T (ext-con Γ A)) → T Γ -- σ
```

So one possibility of still retrieving the IIT as an initial algebra of this form is to devise a general scheme to construct suitable categories of such algebras, by starting out with the sorts and incrementally adding the constructors, until all the constructors have been added, and the IIT can be retrieved as the carrier set of the initial object of the final category of algebras.

This is precisely the process formalised in Altenkirch et al. (2018), only they consider the more general case of allowing set-truncated equalities, i.e. they consider QIITs. This means that their work also describes cases where we add the path constructor `sig-eq : (Γ : Con) → (A : Ty Γ) → (B : Ty (Γ , A)) → ((Γ , A) , B) ≡ (Γ , σ Γ A B)` to `Con`.

Roughly speaking, the authors first desribe how to specify the sorts of a QIIT via a category, then they describe how to specify each constructor via pairs of presheaves L and R, where L specifies the arguments of a constructor and R specifies the target, and then they show that incrementally adding each constructor to the initial category of sorts gives rise to a category of algebras, whose initial algebra corresponds to the QIIT. The limitation of this work is that there is no guarantee that this initial algebra does in fact exist. In other words, this work guarantees that *if* the initial algebra exists, then it corresponds to the QIIT, but the construction described might also allow the specification of categories of algebras which do not have an initial algebra, i.e. it allows the specification of non-strictly positive inductive types. To solve this problem, we will introduce several restrictions to the presheaves L and R to make sure we do not allow such types to be defined, via a process of 'containerification' of the presheaves. This will ensure that only strictly positive types can be defined in our scheme.

Let us consider a subset of the IIT shown previously and give a brief idea of how we would go about constructing the categories of algebras for this IIT.

```
data Con' : Type
data Ty' : Con' → Type

data Con' where
```

```
_,_ : (Γ : Con') → Ty' Γ → Con'

data Ty' where
  σ : (Γ : Con') (A : Ty' Γ) (B : Ty' (Γ , A)) → Ty' Γ
```

Our first category of algebras $\mathcal{A}_0$ will represent the sorts `Con'` and `Ty'`. We can easily model these sorts using the category of families on sets $\underline{\mathbf{Fam}}$ (see section 2.4.1), whose objects are pairs (C, T) where $C : \mathsf{Set}$ and $T\colon S \to \mathsf{Set}$, and whose morphisms $(C, T) \to (C', T')$ are pairs of functions (f, g) with $f\colon C \to C'$ and $g\colon (c : C) \to T\,c \to T'\,(f\,c)$.

The next category of algebras $\mathcal{A}_1$ will represent the sorts with the constructor `_,_`. We specify this constructor using two presheaves L_{ext} and R_{ext}, with the first specifying the left hand side $(\Gamma :$ `Con'`$) \to$ `Ty'` Γ (or the list of arguments), and the second specifying the right hand side `Con'` (or the target).

$$
\begin{aligned}
L_{\text{ext}}\colon \mathcal{A}_0 &\to \underline{\mathbf{Set}} \\
R_{\text{ext}}\colon \textstyle\int L_{\text{ext}} &\to \underline{\mathbf{Set}} \\
&= (a_0 : \mathcal{A}_0) \times (L_{\text{ext}}\,a_0) \to \underline{\mathbf{Set}}
\end{aligned}
$$

$$
\begin{aligned}
L_{\text{ext}}(C, T) &\coloneqq \sum(\Gamma : C)(T\,\Gamma) \\
R_{\text{ext}}((C, T), l : L_{\text{ext}}(C, T)) &\coloneqq C
\end{aligned}
$$

Now given $\mathcal{A}_n, \mathcal{A}_{n+1}$ can roughly be constructed as

$$
\mathcal{A}_{n+1} \coloneqq \sum(a_n : \mathcal{A}_n)((l : L_{n+1}\,a_n) \to R_{n+1}\,a_n\,l),
$$

so $\mathcal{A}_1$ has objects of type $\sum(a_0 : \mathcal{A}_0)(e : (l : L_{\text{ext}}\,a_0) \to R_{\text{ext}}\,a_0\,l))$.

Similarly, we construct the next category of algebras $\mathcal{A}_2$ by first specifying

the next constructor σ via presheaves:

$$\begin{aligned} L_\sigma &: \mathcal{A}_1 \to \underline{\mathbf{Set}} \\ R_\sigma &: \textstyle\int L_\sigma \to \underline{\mathbf{Set}} \\ &\quad = (a_1 : \mathcal{A}_1)(L_\sigma\, a_1) \to \underline{\mathbf{Set}} \end{aligned}$$

$$\begin{aligned} L_\sigma(C, T, e) &:= \sum(\Gamma : C) \sum(A : T\,\Gamma)(T\,(e\,\Gamma\,A)) \\ R_\sigma((C, T, e)\,((\Gamma, A, t_{\Gamma,A}) : L_\sigma\,(C, T, e))) &:= T\,\Gamma \end{aligned}$$

and then constructing $\mathcal{A}_2$ that has objects $\sum(a_1 : \mathcal{A}_1)(e\colon (l : L_\sigma\, a_1) \to R_\sigma\, a_1\, l)$. Since we have added all the constructors, $\mathcal{A}_2$ is the final category of algebras, whose initial algebra would correpond to the `Con' Ty'` type.

The two important points to note about the representable functors L and R are:

- We will want to restrict L and R to be container functors as this will only allow the definition of strictly positive inductive types. Specifically, since the domains of these functors are categories of algebras, we will need to use generalised containers.
- L and R interpret expressions in type theory (left and right hand sides of a constructor) as containers, hence in general we want to be able to interpret any type theoretic expression as a container. This is one motivation for our container model of type theory, detailed in section 5.1.

5.3 Syntax for (Q)IITs

For (3), I will specify a general syntax for IITs and QIITs given rise to by the semantics discussed in section 5.2. Such a syntax is desirable as it encapsulates precisely what it means to be an IIT or a QIIT, and allows us to then prove theorems about these classes of types.

One such theorem is that IITs can be reduced to inductive families, i.e. any IIT can be rewritten as an inductive family, making IITs and inductive families equally as expressive. This theorem is part of the so called ‘folklore’ of type theory, as it is believed to be true but there is no proof of it in full

generality. von Raumer (2020) provides a starting point for this reduction, but does not succeed in completing it. The syntax for IITs used in this reduction attempt is the one presented by Kaposi et al. (2019) for QIITs, excluding the parts about quotienting. We conjecture that the reason the reduction from IITs to inductive families could not be completed is due to this syntax, and that if instead we use the syntax resulting from our 'containerified' semantics, we will get unstuck. While Kaposi et al. (2019) treat adding arguments to a constructor and adding a constructor to an algebra in the same way (using context extension), our syntax will treat them differently. We hope that this will allow us to complete the reduction from IITs to inductive families.

APPENDIX

Cubical Agda Code

```
-- Definition of category with homsets which are h-sets
record Category {m n : Level} : Type (ℓ-suc (ℓ-max m n)) where
  field
    -- Structure
    Obj : Type m
    Hom : Obj → Obj → Type n
    _∘_ : {A B C : Obj} → Hom B C → Hom A B → Hom A C
    id : {A : Obj} → Hom A A
    -- Properties
    assoc : {A B C D : Obj} (h : Hom C D) (g : Hom B C) (f : Hom A B) →
            (h ∘ g) ∘ f ≡ h ∘ (g ∘ f)
    id-rneutr : {A B : Obj} (f : Hom A B) → f ∘ id ≡ f
    id-lneutr : {A B : Obj} (f : Hom A B) → id ∘ f ≡ f
    homs-are-sets : (A B : Obj) → isSet (Hom A B)

open Category

-----
-- CONTAINERS IN 1 PARAMETER
-----

record Container : Type₁ where
  constructor _◁_&_&_
  field
    S : Type
    P : S → Type
    isSetS : isSet S
    isSetP : ∀ {s : S} → isSet (P s)
```

```
open Container

-- Category Cont with objects Container

record _⇒_ (C₁ C₂ : Container) : Type where
  constructor _◁_
  field
    u : S C₁ → S C₂
    f : (s : S C₁) → P C₂ (u s) → P C₁ s

open _⇒_

_∘c_ : {C₁ C₂ C₃ : Container} → C₂ ⇒ C₃ → C₁ ⇒ C₂ → C₁ ⇒ C₃
_∘c_ (v ◁ g) (u ◁ f) = (λ a → v (u a)) ◁ λ a ga → f a (g (u a) ga)

id-∘c : {C : Container} → C ⇒ C
id-∘c = (λ s → s) ◁ λ _ p → p

assoc-c : {C₁ C₂ C₃ C₄ : Container} (h : C₃ ⇒ C₄) (g : C₂ ⇒ C₃) (f : C₁ ⇒ C₂) →
          h ∘c (g ∘c f) ≡ (h ∘c g) ∘c f
assoc-c (w ◁ h) (v ◁ g) (u ◁ f) = refl

isSet⇒ : (C₁ C₂ : Container) → isSet (C₁ ⇒ C₂)
u (isSet⇒ (A ◁ B & set-A & set-B) (C ◁ D & set-C & set-D) m n p q i j) a =
  set-C (u m a) (u n a) (λ k → u (p k) a) (λ k → u (q k) a) i j
f (isSet⇒ (A ◁ B & set-A & set-B) (C ◁ D & set-C & set-D) m n p q i j) a =
  isSet→SquareP
    {A = λ i j → D (set-C (u m a) (u n a) (λ k → u (p k) a) (λ k → u (q k) a) i j) → B a}
    (λ i j → isSetΠ λ _ → set-B)
    (λ k → f (p k) a)
    (λ k → f (q k) a)
    (λ _ → f m a)
    (λ _ → f n a)
    i j

Cont : Category {ℓ-suc ℓ-zero} {ℓ-zero}
Cont = record
          { Obj = Container
          ; Hom = _⇒_
```

```
    ; _∘_ = _∘c_
    ; id = id-∘c
    ; assoc = assoc-c
    ; id-rneutr = λ m → refl
    ; id-lneutr = λ m → refl
    ; homs-are-sets = isSet⇒
    }

-- Category of endofunctors on Set

record Functor {m m' n n'} (C : Category {m} {m'}) (D : Category {n} {n'}) : Type
      (ℓ-max m (ℓ-max m' (ℓ-max n n'))) where
  field
    -- Structure
    func-obj : Obj C → Obj D
    func-mor : {A B : Obj C} → (Hom C) A B → (Hom D) (func-obj A) (func-obj B)
    -- Properties
    func-id : {A : Obj C} → func-mor (id C {A}) ≡ (id D {func-obj A})
    func-comp : {U V W : Obj C} (g : (Hom C) V W) (f : (Hom C) U V) →
                func-mor ((_∘_) C g f) ≡ (_∘_) D (func-mor g) (func-mor f)

open Functor

record NaturalTransformation {n n'} {C D : Category {n} {n'}} (F G : Functor C D) : Type
      (ℓ-max n n') where
  constructor _,nat:_
  field
    mors : (A : Obj C) → (Hom D) (func-obj F A) (func-obj G A)
    nat : (X Y : Obj C) (f : (Hom C) X Y) →
          (_∘_ D) (func-mor G f) (mors X) ≡ (_∘_ D) (mors Y) (func-mor F f)

open NaturalTransformation

record h-set : Type₁ where
  field
    set : Type
    is-set : isSet set

open h-set
```

```
isSet→ : (X Y : h-set) → isSet (set X → set Y)
isSet→ X Y = isSetΠ λ x → is-set Y

_•_ : ∀ {a b c} {A : Type a} {B : Type b} {C : Type c} → (B → C) → (A → B) → (A → C)
f • g = λ x → f (g x)

SetC : Category {ℓ-suc ℓ-zero} {ℓ-zero}
SetC = record
        { Obj = h-set
        ; Hom = λ A B → set A → set B
        ; _∘_ = _•_ -- λ f g a → f (g a)
        ; id = λ a → a
        ; assoc = λ _ _ _ → refl
        ; id-rneutr = λ _ → refl
        ; id-lneutr = λ _ → refl
        ; homs-are-sets = isSet→
        }

∘-f : {F G H : Functor SetC SetC} → NaturalTransformation G H → NaturalTransformation F G →
      NaturalTransformation F H
mors (∘-f {F} {G} {H} (β-m ,nat: β-n) (α-m ,nat: α-n)) = (λ A fa → β-m A (α-m A fa))
nat (∘-f {F} {G} {H} (β-m ,nat: β-n) (α-m ,nat: α-n)) = λ X Y f →
  (λ i fx → β-n X Y f i (α-m X fx)) • (λ i fx → β-m Y (α-n X Y f i fx))

id-mor-f : {F : Functor SetC SetC} → NaturalTransformation F F
id-mor-f = (λ A FA → FA) ,nat: λ _ _ f → refl

assoc-f : {F G H I : Functor SetC SetC} (γ : NaturalTransformation H I)
          (β : NaturalTransformation G H) (α : NaturalTransformation F G) →
          (∘-f {F} {G} {I} (∘-f {G} {H} {I} γ β) α) ≡ (∘-f {F} {H} {I} γ (∘-f {F} {G} {H} β α))
assoc-f {F} {G} {H} {I} (γ-m ,nat: γ-n) (β-m ,nat: β-n) (α-m ,nat: α-n) =
  cong₂ _,nat:_ refl (isProp→PathP (λ i → isPropΠ3
  λ X Y f → isSet→ (func-obj F X) (func-obj I Y) _ _) _ _)

id-rneutr-f : {F G : Functor SetC SetC} (α : NaturalTransformation F G) →
              ∘-f {F} {F} {G} α (id-mor-f {F}) ≡ α
id-rneutr-f {F} {G} α =
  cong₂ _,nat:_ refl (isProp→PathP (λ i → isPropΠ3
```

```
  λ X Y f → isSet→ (func-obj F X) (func-obj G Y) _ _) _ _)

id-lneutr-f : {F G : Functor SetC SetC} (f : NaturalTransformation F G) →
                ∘-f {F} {G} {G} (id-mor-f {G}) f ≡ f
id-lneutr-f {F} {G} α =
  cong₂ _,nat:_ refl (isProp→PathP (λ i → isPropΠ3
  λ X Y f → isSet→ (func-obj F X) (func-obj G Y) _ _) _ _)

isSetNatTrans : (F G : Functor SetC SetC) → isSet (NaturalTransformation F G)
mors (isSetNatTrans F G α β p q i j) X s =
  is-set (func-obj G X) (mors α X s) (mors β X s) (λ k → mors (p k) X s) (λ k → mors (q k) X s) i j
nat (isSetNatTrans F G α β p q i j) X Y xy k fx = cube i j k
  where
    cube : Cube (λ j k → nat (p j) X Y xy k fx) (λ j k → nat (q j) X Y xy k fx)
                (λ i k → nat α X Y xy k fx) (λ i k → nat β X Y xy k fx)
                (λ i j → func-mor G xy (mors (isSetNatTrans F G α β p q i j) X fx))
                (λ i j → is-set (func-obj G Y) (mors α Y (func-mor F xy fx))
                         (mors β Y (func-mor F xy fx))
                         (λ k → mors (p k) Y (func-mor F xy fx))
                         (λ k → mors (q k) Y (func-mor F xy fx)) i j)
    cube = isSet→SquareP (λ i j → isOfHLevelPath 2 (is-set (func-obj G Y)) _ _) _ _ _ _

Func : Category {ℓ-suc ℓ-zero} {ℓ-suc ℓ-zero}
Func = record
          { Obj = Functor SetC SetC
          ; Hom = λ F G → NaturalTransformation F G
          ; _∘_ = λ {F} {G} {H} gh fg → ∘-f {F} {G} {H} gh fg
          ; id = λ {F} → id-mor-f {F}
          ; assoc = λ {F} {G} {H} {I} → assoc-f {F} {G} {H} {I}
          ; id-rneutr = λ {F} {G} → id-rneutr-f {F} {G}
          ; id-lneutr = λ {F} {G} → id-lneutr-f {F} {G}
          ; homs-are-sets = isSetNatTrans
          }

-- Functor ⟦_⟧ : Cont → Func

record cont-func (A : Type) (B : A → Type) (X : h-set) : Type where
  constructor _<_
```

```
  field
    shape : A
    pos : B shape → set X

open cont-func

isSetContFunc : (A : Type) (B : A → Type) (X : h-set) (isSetA : isSet A)
                (isSetB : ∀ {a : A} → isSet (B a)) → isSet (cont-func A B X)
shape (isSetContFunc A B X setA setB s₁ s₂ p q i j) =
  setA (shape s₁) (shape s₂) (λ k → shape (p k)) (λ k → shape (q k)) i j
pos (isSetContFunc A B X setA setB s₁ s₂ p q i j) =
  isSet→SquareP
    {A = λ i j → B (setA (shape s₁) (shape s₂) (λ k → shape (p k)) (λ k → shape (q k)) i j) → set X}
    (λ _ _ → isSetΠ (λ _ → is-set X))
    (λ k → pos (p k))
    (λ k → pos (q k))
    (λ _ → pos s₁)
    (λ _ → pos s₂)
    i j

cont-mor : {A : Type} {B : A → Type} {X Y : h-set} (f : set X → set Y) →
           cont-func A B X → cont-func A B Y
cont-mor f (s < g) = s < λ b → f (g b)

⟦_⟧-obj : Container → Functor SetC SetC
⟦ (A ◁ B & set-A & set-B) ⟧-obj = record
                                   { func-obj = λ X →
                                           record { set = cont-func A B X ;
                                                    is-set = isSetContFunc A B X set-A set-B
                                                    } ;
                                     func-mor = λ {X} {Y} f → cont-mor {A} {B} {X} {Y} f;
                                     func-id = refl ;
                                     func-comp = λ _ _ → refl
                                   }
⟦_⟧-mor : {C D : Container} → C ⇒ D → NaturalTransformation ⟦ C ⟧-obj ⟦ D ⟧-obj
mors (⟦_⟧-mor (u ◁ f)) X (s < p) = u s < λ q → p (f s q)
nat (⟦_⟧-mor (u ◁ f)) X Y xy i (s < p) = u s < λ q → xy (p (f s q))
```

```
⟦_⟧-comp : {U V W : Container} (g : V ⇒ W) (f : U ⇒ V) → ⟦ g ∘c f ⟧-mor ≡ ∘-f ⟦ g ⟧-mor ⟦ f ⟧-mor
⟦_⟧-comp {S ◁ P & set-S & set-P} {T ◁ Q & set-T & set-Q} {U ◁ R & set-U & set-R} g f =
                                        cong₂ _,nat:_ refl
                                          (funExt λ X → funExt (λ Y → funExt (λ xy →
                                          isProp→PathP
                                              (λ _ → λ p q i j sp →
                                              isSetContFunc U R Y set-U set-R
                                                  ((cont-mor xy • mors ⟦ g ∘c f ⟧-mor X) sp)
                                                  ((mors ⟦ g ∘c f ⟧-mor Y • cont-mor xy) sp)
                                                  (funExt⁻ p sp)
                                                  (funExt⁻ q sp)
                                                  i j)
                                          _ _)))

⟦_⟧ : Functor Cont Func
⟦_⟧ = record
        { func-obj = ⟦_⟧-obj ;
          func-mor = ⟦_⟧-mor ;
          func-id = refl ;
          func-comp = ⟦_⟧-comp
        }

------

-- Example

ListC : Container
ListC = ℕ ◁ Fin & isSetℕ & isSetFin

ListF : Functor SetC SetC
ListF = ⟦ ListC ⟧-obj


-- Proof 1 that the functor ⟦_⟧ is full and faithful
-- Adapted from 'Containers: Constructing strictly positive types'

_-fully-faithful : {m m' n n' : Level} {C : Category {m} {n}} {D : Category {m'} {n'}} →
                        Functor C D → Type (ℓ-max (ℓ-max m n) n')
_-fully-faithful {C = C} {D = D} F = (X Y : Obj C) → Iso ((Hom C) X Y) ((Hom D)
```

```
                                        ((func-obj F) X) ((func-obj F) Y))

fun : (X Y : Container) → X ⇒ Y → NaturalTransformation ⟦ X ⟧-obj ⟦ Y ⟧-obj
fun X Y = ⟦_⟧-mor {X} {Y}

inv : (X Y : Container) → NaturalTransformation ⟦ X ⟧-obj ⟦ Y ⟧-obj → X ⇒ Y
inv (A ◁ B & _ & set-B) (C ◁ D & _ & _) (mors ,nat: nat) =
  (λ a → shape (cd a)) ◁ (λ a d → pos (cd a) d)
    where
      Ba : A → h-set
      Ba a = record { set = B a ; is-set = set-B }
      cd : (a : A) → cont-func C D (Ba a)
      cd a = mors (Ba a) (a < λ ba → ba)

sec : (X Y : Container) → ∀ nat-trans → (fun X Y) ((inv X Y) nat-trans) ≡ nat-trans
sec (A ◁ B & set-A & set-B) (C ◁ D & set-C & set-D) (mors ,nat: nat) =
  cong₂
    _,nat:_
    (funExt
      {f = NaturalTransformation.mors
            (fun (A ◁ B & set-A & set-B) (C ◁ D & set-C & set-D)
            (inv (A ◁ B & set-A & set-B) (C ◁ D & set-C & set-D) (mors ,nat: nat)))}
      {g = mors}
      λ X → funExt
                {f = NaturalTransformation.mors
                      (fun (A ◁ B & set-A & set-B) (C ◁ D & set-C & set-D)
                      (inv (A ◁ B & set-A & set-B) (C ◁ D & set-C & set-D) (mors ,nat: nat))) X}
                {g = mors X}
                λ {(s' < h') → funExt⁻ (nat (record { set = B s' ; is-set = set-B }) X h') (s' < λ x → x)}})
    (isProp→PathP (λ i → isPropΠ3
    (λ X Y f → isSetΠ (λ _ → isSetContFunc C D Y set-C set-D) _ _)) _ _)

ret : (X Y : Container) → ∀ mor → (inv X Y) ((fun X Y) mor) ≡ mor
ret C₁ C₂ (u ◁ f) = refl

⟦_⟧-fully-faithful : ⟦_⟧ -fully-faithful
⟦_⟧-fully-faithful X Y = iso (fun X Y) (inv X Y) (sec X Y) (ret X Y)
```

```
-- Proof 2 that the functor ⟦_⟧ is full and faithful

record ProFunctor {m m'} (C : Category {m} {m'}) : Type (ℓ-suc (ℓ-max m m')) where
  field
    -- Structure
    profunc-obj : Obj C → Obj C → Obj SetC
    profunc-mor : {P P' Q Q' : Obj C} → (Hom C) P P' → (Hom C) Q Q' →
                  (Hom SetC) (profunc-obj P' Q) (profunc-obj P Q')
    -- Properties
    profunc-id : {P Q : Obj C} → profunc-mor (id C {P}) (id C {Q}) ≡ (id SetC {profunc-obj P Q})
    profunc-comp : {P Q R S T U : Obj C} (g : (Hom C) Q R) (g' : (Hom C) P Q)
                   (f : (Hom C) T U) (f' : (Hom C) S T) →
                   profunc-mor ((_∘_ C) g g') ((_∘_ C) f f') ≡
                   (_∘_ SetC) {profunc-obj R S} {profunc-obj Q T} {profunc-obj P U}
                   (profunc-mor g' f) (profunc-mor g f')

open ProFunctor

-- Hom-functor in general

hom-functor : ∀ {m} → (C : Category {m} {ℓ-zero}) → ProFunctor {m} {ℓ-zero} C
hom-functor C = record
                  { profunc-obj = λ X Y →
                      record { set = (Hom C) X Y ; is-set = (homs-are-sets C) X Y }
                  ; profunc-mor = λ aa' bb' a'b → (_∘_ C) ((_∘_ C) bb' a'b) aa'
                  ; profunc-id = funExt λ f → (id-rneutr C ((C ∘ id C) f)) · id-lneutr C f
                  ; profunc-comp = λ {P} {Q} {R} {S} {T} {U} qr pq tu st →
                    funExt λ rs →
                      cong (λ X → (C ∘ X) ((C ∘ qr) pq)) (assoc C tu st rs) ·
                      assoc C tu ((C ∘ st) rs) ((C ∘ qr) pq) ·
                      cong (λ X → (C ∘ tu) X) (sym (assoc C ((C ∘ st) rs) qr pq)) ·
                      sym (assoc C tu ((C ∘ ((C ∘ st) rs)) qr) pq)
                  }

nats : (F G : Functor SetC SetC) → ProFunctor SetC
nats F G = record
             { profunc-obj = λ c c' →
                                 record { set = (Hom SetC) (func-obj F c) (func-obj G c') ;
```

```
                                    is-set = isSetΠ (λ _ → is-set (func-obj G c’)) }
      ; profunc-mor = λ f g h x → func-mor G g (h (func-mor F f x))
      ; profunc-id =
                  funExt λ f → funExt λ fx →
                    cong (λ X → func-mor G (λ a → a) (f X)) (funExt⁻ (func-id F) fx) ∙
                    funExt⁻ (func-id G) (f fx)
      ; profunc-comp = λ {P} {Q} {R} {S} {T} {U} qr pq tu st →
                  funExt λ rs → funExt λ Fp →
                  cong (λ X → func-mor G (tu ∙ st) (rs X)) (funExt⁻ (func-comp F qr pq) Fp) ∙
                  funExt⁻ (func-comp G tu st) (rs (func-mor F qr (func-mor F pq Fp)))
      }

record ∫ {m m’} {C : Category {m} {m’}} (F : ProFunctor C) : Type (ℓ-suc (ℓ-max m m’)) where
  field
    funcs : (c : Obj C) → set (profunc-obj F c c)
    nat : (c d : Obj C) (f : (Hom C) c d) →
          profunc-mor F ((id C) {c}) f (funcs c) ≡ profunc-mor F f ((id C) {d}) (funcs d)

open ∫


H^ : (A : Obj SetC) → Functor SetC SetC
H^ A = record
          { func-obj = λ B → record { set = (Hom SetC) A B ; is-set = isSetΠ (λ _ → is-set B) } ;
            func-mor = λ g p a → g (p a);
            func-id = refl ;
            func-comp = λ _ _ → refl }

yoneda-lemma : (F : Functor SetC SetC) (A : Obj SetC) → Iso (∫ (nats (H^ A) F)) (set (func-obj F A))
yoneda-lemma = {!!}

⟦_⟧-fully-faithful’ : (C₁ C₂ : Container) → Iso (∫ (nats ⟦ C₁ ⟧-obj ⟦ C₂ ⟧-obj)) (C₁ ⇒ C₂)
⟦_⟧-fully-faithful’ = {!!}
```

Bibliography

Abbott, M., Altenkirch, T., and Ghani, N. (2005). Containers: Constructing strictly positive types. *Theoretical Computer Science*, 342(1):3–27. Applied Semantics: Selected Topics.

Abel, A. and Altenkirch, T. (2000). A predicative strong normalisation proof for a λ-calculus with interleaving inductive types. volume 1956, pages 227–243.

Altenkirch, T. (2019). *Categories for the lazy functional programmer*. http://www.cs.nott.ac.uk/~psztxa/mgs.2019/ [Accessed: July 2022].

Altenkirch, T., Capriotti, P., Dijkstra, G., Kraus, N., and Nordvall Forsberg, F. (2018). Quotient inductive-inductive types. In Baier, C. and Dal Lago, U., editors, *Foundations of Software Science and Computation Structures*, pages 293–310, Cham. Springer International Publishing.

Altenkirch, T., Ghani, N., Hancock, P., Mcbride, C., and Morris, P. (2015). Indexed containers. *Journal of Functional Programming*, 25.

Altenkirch, T. and Kaposi, A. (2021). A container model of type theory. In *TYPES 2021*.

Altenkirch, T., Levy, P., and Staton, S. (2010). Higher-order containers. In Ferreira, F., Löwe, B., Mayordomo, E., and Mendes Gomes, L., editors, *Programs, Proofs, Processes*, pages 11–20, Berlin, Heidelberg. Springer Berlin Heidelberg.

Altenkirch, T. and Morris, P. (2009). Indexed containers. In *2009 24th Annual IEEE Symposium on Logic In Computer Science*, pages 277–285.

Atkey, R. (2020). Interpreting dependent types with containers. Talk at the MSP101 seminar, University of Strathclyde.

Brunerie, G. (2016). *On the homotopy groups of spheres in homotopy type theory*. PhD thesis.

Cartmell, J. (1986). Generalised algebraic theories and contextual categories. *Annals of Pure and Applied Logic*, 32:209–243.

Dybjer, P. (1996). Representing inductively defined sets by wellorderings in Martin-Löf's type theory.

Dybjer, P. (2003). Internal type theory.

Gambino, N. and Hyland, M. (2004). Wellfounded trees and dependent polynomial functors. In Berardi, S., Coppo, M., and Damiani, F., editors, *Types for Proofs and Programs*, pages 210–225, Berlin, Heidelberg. Springer Berlin Heidelberg.

Hofmann, M. (1997). *Syntax and semantics of dependent types*, pages 13–54. Springer London, London.

Jacobs, B. (1993). Comprehension categories and the semantics of type dependency. *Theoretical Computer Science*, 107(2):169–207.

Kaposi, A. (2013). First year report. An introduction to type theory with a notation using De Bruijn indices and explicit substitutions.

Kaposi, A., Kovács, A., and Altenkirch, T. (2019). Constructing quotient inductive-inductive types. *Proc. ACM Program. Lang.*, 3(POPL).

Kovács, A. (2020). The container model of type theory. Talk at the type theory seminar, Eötvös Loránd University.

Leinster, T. (2016). Basic category theory.

Licata, D. R. and Shulman, M. (2013). Calculating the fundamental group of the circle in homotopy type theory.

Lumsdaine, P. L. and Shulman, M. (2019). Semantics of higher inductive types. *Mathematical Proceedings of the Cambridge Philosophical Society*, 169(1):159–208.

Martin-Löf, P. (1971). Hauptsatz for the intuitionistic theory of iterated inductive definitions. In *Studies in Logic and the Foundations of Mathematics*, volume 63, pages 179–216. Elsevier.

Martin-Löf, P. (1972). An intuitionistic theory of types. *Twenty-Five Years of Constructive Type Theory*.

Martin-Löf, P. (1982). Constructive mathematics and computer programming. In *Studies in Logic and the Foundations of Mathematics*, volume 104, pages 153–175. Elsevier.

Martin-Löf, P. and Sambin, G. (1984). *Intuitionistic type theory*, volume 9. Bibliopolis Naples.

Milewski, B. (2018). *Category Theory for Programmers*. Blurb, Incorporated.

The Univalent Foundations Program (2013). *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study.

von Glehn, T. (2015). *Polynomials and models of type theory*. PhD thesis, University of Cambridge.

von Raumer, J. (2020). *Higher inductive types, inductive families, and inductive-inductive types*. PhD thesis, University of Nottingham, UK.

Index

category of elements, 10
category of families, 10
category with families, 44
container
- n-ary container, 36
- n-ary container extension functor, 36
- indexed container, 42
- category of n-ary containers, 36
- category of (unary) containers, 24
- constant container, 32
- container extension functor, 25
- container functor, 22
- generalised container, 42
- unary container, 22

definitional equality, 5

end, 11

hom functor, 13

inductive type, 5

profunctor, 10
propositional equality, 5

representable functor, 14

type
- inductive type, 20
- strictly positive type, 21

wedge, 11