Containers, Cubes, and Coherences:
Containers in Homotopy Type Theory

by

Stefania Damato

Functional Programming Lab
School of Computer Science
University of Nottingham

2026

A thesis submitted to the University of Nottingham for the degree of
Doctor of Philosophy

Abstract

Martin-Löf's dependent type theory is a formal language for programming and constructive mathematics. It acts as the basis for modern proof assistants like Agda, which are tools for doing computer-assisted mathematics. Central to Martin-Löf's type theory is the notion of an inductive type, which allows us to define a new type by providing a list of constructors specifying how to form elements of the type. Homotopy type theory is a more recent development which interprets type theory in a homotopy theoretic way, whereby types are seen as spaces, elements as points in a space, and the identity type 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). Inductive types of this kind are referred to as higher inductive types. Ordinary inductive types and inductive families in Martin-Löf's type theory are well understood and already enjoy fully developed semantics thanks to containers, which are semantic gadgets used to carve out a class of particularly well-behaved inductive types, called strictly positive types, in terms of 'shapes' and 'positions'. A similar well-established theoretical foundation for higher inductive types has not yet been found.

The aim of this thesis is to explore the theoretical foundations and applications of container theory in the context of homotopy type theory, to contribute towards the long-term goal of providing semantics for higher inductive types via generalisations of established container theory. The thesis is the result of several projects carried out on this theme.

Chapter 1 presents an overview of the different kinds of containers that have been developed in the literature, while giving explicit proofs of some of their closure properties that are missing in the literature, and giving an account of generalised containers.

Chapter 2 showcases a formalisation in `Cubical Agda` of the result that 'containers are closed under least and greatest fixed points', generalising the original result by Abbott, Altenkirch, and Ghani into a setting where uniqueness of identity proofs is not assumed.

Chapter 3 presents a construction of a groupoid category with families of containers, which is fully formalised in `Cubical Agda`. Constructing a model of type theory using containers was motivated in the first place by our exploration into using containers to provide semantics for quotient inductive-inductive types, which we also discuss

towards the end of the chapter.

Chapter 4 investigates applying the theory of containers to the study of monad distributive laws. We characterise monadic, monadic-directed, and directed-monadic container distributive laws, that have been formalised in `Cubical Agda`.

Acknowledgements

I would like to express my sincerest gratitude to all those who supported me throughout the course of my PhD. I start by thanking my supervisor, Thorsten Altenkirch, for the countless discussions, whiteboard explanations, guidance, and for his passion for all things type theory. I also thank my second supervisor, Nicolai Kraus, for consistently checking in on my progress and providing advice and encouragement.

I am very lucky to have been part of the Functional Programming Lab (FP lab) in Nottingham. I will dearly miss the weekly lab meetings, trips to the pub, insightful debates, and the superb atmosphere in the whole group. As a bonus, having so many knowledgeable people who are always happy to answer questions right down the corridor from me was an invaluable resource. Many thanks go to Tom de Jong for being the most helpful postdoc a PhD student can hope for. Thank you for going above and beyond your job description, proofreading my work, offering career advice, supporting me and other PhD students in the FP lab, and creating this thesis template! Thanks to Graham Hutton for examining my annual reviews and always giving me very practical feedback, and thanks to the remaining faculty members, Ulrik Buchholtz and Dan Marsden, for all of your influence and insights. And of course, thank you to all the current and former PhD students of the FP lab, who made me look forward to go into the office everyday, and who made my time there so enjoyable: Josh Chen, Zac Garby, Brandon Hewer, Aref Mohammadzadeh, Jacob Neumann, Stiéphen Pradal, Johannes Schipp von Branitz, Filippo Sestini, Zhili Tian, Mark Williams, and Sky Wilshaw.

Outside the FP lab, I want to thank my collaborators: Axel Ljungström, for always enthusiastically sharing your `Cubical Agda` expertise with me whenever I have questions, and Chris Purdy, for having so many creative research ideas. Working with you both has been great fun, and I hope we have the opportunity to do it again in the future! I received funding on a few occasions from the COST Action EuroProofNet (CA20111), for which I am very thankful. I also want to thank the members of the Rambling and Hiking Club and the Mature Students' Network, for making Nottingham feel like home the last few years.

I relocated to Budapest during the last weeks of writing this thesis. I want to thank David and Eleanor Berry for their hospitality, as well as the ELTE type theory group for their warm welcome.

I also want to acknowledge my examiners, Fredrik Nordvall Forsberg and Dan Marsden, for their careful reading of my thesis and helpful suggestions.

I cannot forget the support of those who helped me get a PhD position in the first place: Duncan Paul Attard, Jean Paul Ebejer, Adrian Francalanza, and Dario Landa-Silva. Thank you for believing in me from the very beginning. Lastly, I'm very grateful for the support of my family and friends back in Malta and abroad who have been with me on this journey. I could not have done this without you!

You must always have faith in people. And most importantly, you must always have faith in yourself. — Elle Woods

Contents

Chapter 0

Introduction

We give a brief overview of the thesis as a guide to the reader, and list the main contributions of each chapter. We also make note of which parts of the thesis are based on our publications (or unpublished work), and give credit where it is due to our collaborators on these projects.

The reader is assumed to be familiar with basic category theory notions such as functors, natural transformations, limits and colimits etc. Having some experience with (vanilla) `Agda` is helpful, since some of our definitions are given as `Agda` code, while knowledge of `Cubical Agda` is not assumed, as we give a brief introduction before we use it.

Our work takes place in different flavours of dependent Martin-Löf type theory. In order to be compatible with homotopy type theory, we do not assume uniqueness of identity proofs, and when we require a type to be an h-set we state this explicitly. Most of the work presented has an accompanying formalisation in `Cubical Agda`, which implements a version of cubical type theory, although when possible we try to remain independent of any one model of homotopy type theory.

0.1 Outline & Contributions

Chapter 0: Introduction

Section 0.2 gives a brief introduction to (homotopy) type theory, which can be skipped by the reader already familiar with this field. Section 0.3 introduces and gives examples of different classes of inductive types. Section 0.4 defines some category theory constructs that appear later, and Section 0.5 explicates notations that will be used thereafter.

Declaration of Authorship Sections 0.2 and 0.3 and parts of Section 0.4 are adapted from my master's thesis [Dam20] and my first year review report [Dam22].

Chapter 1: A Survey of Containers

This chapter provides an overview of the different kinds of containers in the literature, namely unary, n-ary, and indexed containers, and introduces generalised containers. We look at categories of containers and relate them to categories of functors, and look at closure properties of containers.

Contributions

- Definitions and proofs of containers stated in a type theoretic setting
- Theorem 1.3.21: Proof that n-ary containers are closed under exponentiation
- Theorem 1.4.8: Proof that indexed containers are closed under exponentiation
- Section 1.5: Formal definitions, closure properties, and formalisation of generalised containers

Declaration of Authorship The majority of this chapter is a collection of various existing results on containers. Most of these results are presented in category theory, but we work in and therefore present them in type theory. Many of the results on unary containers were first shown by Abbott, Altenkirch, and Ghani [AAG03; AAG04; AAG05; Abb03], while those on indexed containers were shown by Altenkirch, Ghani, Hancock, McBride, and Morris [AM09; AGH+15]. To our knowledge, Theorems 1.3.21 and 1.4.8 as stated above are new, and due to Altenkirch and myself. Generalised containers were mentioned in an abstract by Altenkirch and Kaposi [AK21], but the results in Section 1.5 are new and due to Altenkirch and myself, while the formalisation was done by myself.

Chapter 2: Inductive & Coinductive Containers

We present a formalisation in `Cubical Agda` of the results stating that container functors are closed under initial algebras (least fixed points) and terminal coalgebras (greatest fixed points), first proved by Abbott et al. We do so while making no h-set assumptions, thereby generalising the original results from talking about h-sets to types of any h-level. Most of the chapter details our experience using coinductive types in `Cubical Agda`, the challenges that were involved, and our workarounds.

Contributions

- Section 2.1.3: Stating container definitions in terms of the wild category of types `Type` instead of the category of sets **Set**
- A formalisation of Theorem 2.3.1: container functors are closed under least fixed points, with no h-set assumptions
- A formalisation of Theorem 2.3.4: container functors are closed under greatest fixed points, with no h-set assumptions

Declaration of Authorship This chapter is a result of joint work with Thorsten Altenkirch and Axel Ljungström. A version of this chapter has previously been published as a conference paper at ITP 2025 [DAL25b]. The aforementioned closure results on containers that we formalised were originally proved by Abbott et al. [AAG05]. Our proof of Theorem 2.3.1 follows their proof very closely. Our proof of Theorem 2.3.4 also

roughly follows theirs, but considerably more care was required to convince `Agda` of termination and to avoid relying on h-set assumptions — two issues that do not show up in the original proof.

Chapter 3: A Container Model of Type Theory

We present a detailed account of the container model of type theory first suggested by Altenkirch and Kaposi. We show that their container model is a groupoid CwF, and formalise this construction fully in `Cubical Agda`. We explain how this project is a prerequisite for our approach of providing categorical semantics for quotient inductive-inductive types (QIITs), and present some conjectures on this matter.

Contributions

- Definition 3.3.3: A precise account of the structure required for a model of type theory to allow for groupoids of types
- Section 3.4: A detailed presentation of the container model of Altenkirch and Kaposi as a GCwF, fully formalised in `Cubical Agda`. We improve and extend their existing incomplete formalisation by
  - making explicit when types are h-sets (whereas before this was implicit),
  - switching out certain definitions (e.g. the pushout), defining a univalence principle for generalised containers (GenCons) and proving several associated lemmas, and proving functoriality of Ty in a specific way all in a bid to set things up nicely for our coherence proofs, and
  - completing all the necessary constructions and proofs for the GCwF construction, in particular, stating and proving the coherence laws related to Ty, and giving proofs relating to terms and context comprehension that were incomplete or omitted in their formalisation.

Declaration of Authorship Our container groupoid CwF builds on the definition and formalisation presented in Altenkirch and Kaposi's abstract [AK21]. Our GCwF definition (Definition 3.3.3) uses some ideas and terminology from [Che25] and should be equivalent to that in [AKX26]. The idea of obtaining semantics for QIITs via containers was suggested to me by Altenkirch, and the discussion in this chapter is based on our joint investigation on the topic. The formalisation in `Cubical Agda` was done by myself, though I benefitted greatly from Axel Ljungström's advice regarding how to set things up to make later proofs easier, and also Altenkirch's feedback throughout.

Chapter 4: Distributive Laws of Monadic Containers

We give characterisations of monadic, monadic-directed, and directed-monadic container distributive laws that have been fully formalised in `Cubical Agda`. We provide numerous examples, including ones that illustrate how these characterisations can be used to show the existence and uniqueness of distributive laws between monadic containers.

Contributions

- Definition 4.3.2: Definition of a distributive law between monadic containers
- Definition 4.3.8: Definition of a monadic container that is the composite of two others
- Definitions 4.3.13 and 4.3.14: Definitions of monadic-directed and directed-monadic container distributive laws
- Various existence and uniqueness of distributive laws proofs, such as Lemmas 4.3.4, 4.3.6 and 4.3.16.
- An accompanying formalisation in `Cubical Agda` of many of our definitions and results

Declaration of Authorship This chapter is a result of joint work with Chris Purdy. We benefitted from discussions with Thorsten Altenkirch on his previous work (with Gun Pinyo) on monadic containers [AP17]. The chapter is based on a conference paper that was published at CALCO 2025 [PD25b]. Our characterisation of monadic container distributive laws and compatible composites follow those of Beck [Bec69]. Our mixed distributive law definitions in Section 4.3.2 combine our characterisation with those of Ahman and Uustalu's [AU13]. Section 4.4 replicates Zwart and Marsden's approach to see if we can carry their results over to our monadic container setting.

0.2 An Introduction to 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** (MLTT) . 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 of the judgments.

In this dissertation, we will always be working in some flavour of MLTT, which Martin-Löf developed, among other things, as a foundation for intuitionistic mathematics. Intuitionism, introduced by Brouwer, is a philosophy of mathematics and a variety of constructivism, which states that a mathematical object exists only if it can be constructed. In particular, in contrast to the widely accepted **Zermelo–Fraenkel set theory with choice** (ZFC), it does not assume the law of the excluded middle, which states that any proposition is either true or false. It is therefore instructive to briefly explain the basic concepts of MLTT in relation to how they are treated in set theory.

0.2.1 Type theory vs. set theory

Most mathematicians today accept and use set theory as the foundation of mathematics. Set theory organises mathematical objects into collections called sets, such as the set of natural numbers $\mathbb{N} = \{0, 1, 2, 3, \dots\}$. All mathematical objects can be represented as sets. For instance, the number 0 is simply a shorthand representation for the empty set $\{\}$, 1 is the set containing 0 $\{\{\}\}$, 2 is the set containing 0 and 1 $\{\{\}, \{\{\}\}\}$, and so on. All other mathematical objects such as relations, functions, lines, curves, and algebraic structures, can be defined in terms of sets.

MLTT organises mathematical objects into types instead of sets, such as the type $\mathbb{N}$ of the natural numbers. Types collect objects of the same nature or structure together, and there are specific ways of forming new types from existing ones. There are four basic judgements in MLTT, the first two of which are 'A is a type' and 'A and B are equal types'.

The third basic judgement in MLTT expresses that 'a mathematical object (or term) a is of type A', and is written as $a : A$. For instance, to express that 3 is of type $\mathbb{N}$, we write $3 : \mathbb{N}$. We note that $a : A$ differs from $a \in A$ in that the former is a judgment whereas the latter is a proposition which can be true or false. We can think of $a \in A$ as a relation about two pre-existing objects a and A which may or may not hold, and $a : A$ as an atomic statement such that we cannot talk about a term a without specifying its type. In MLTT, we can only construct terms of a certain pre-existing type, so that the type comes first and its terms come later.

The fourth basic judgement in MLTT is the judgement $a =_A b$ for the terms $a, b : A$, expressing that 'a and b are **definitionally equal**' (the subscript $_A$ is sometimes elided). MLTT also admits a different kind of equality called **propositional equality**, which is a particular type constructor that relates two terms of the same type. Definitional equality is the stronger kind of equality as it is a judgment (i.e. a statement in the metatheory of the language), while propositional equality is treated like any other type in type theory. To clarify this distinction, we provide an example. When defining a function $f\colon \mathbb{N} \to \mathbb{N}$ by $f(x) \coloneqq x + 5$, that $f(2)$ and 7 are equal is definitional—it is simply a matter of expanding out a definition. On the other hand, that $x + 5$ is equal to $5 + x$ does not follow directly from any definitions but is a proposition that can be proved, and is hence a propositional equality.

The last difference between the theories we will mention here is that contrary to set theory, MLTT is its own deductive system. Set theoretic foundations consist of two levels: the deductive system of first-order logic, together with the axioms of a particular theory, such as ZFC. Therefore, the proofs exist within first-order logic, which is a separate universe to that of the mathematical objects they talk about. However, MLTT encodes both proofs and mathematical objects in a single language. Notions like negation, conjunction, and disjunction can be encoded as types themselves within MLTT.

0.2.2 Propositions as types

How do we encode concepts from predicate logic, such as negation and conjunction, within MLTT? Due to the inuitionistic nature of MLTT, in MLTT we do not think of truth, but rather of evidence. Instead of saying that a proposition P holds or is true, we say that we have evidence for P. We associate P to a type and then construct an element of this type as a witness. Hence, the typing judgment $a : A$ can be read both as 'a is an element (or term) of type A', and as 'a is a witness (or proof) of the proposition A'.

This notion is known as the *Curry–Howard correspondence* or *propositions–as–types*. The basic idea is that derivations in natural deduction, or proofs, and terms in lambda

calculus, or computations, are essentially equivalent. Proofs correspond to programs, and the statement a proof is proving is the type of the program. Hence types play the role of propositions, and terms of a type A are proofs of the proposition A. We combine this with the Brouwer-Heyting-Kolmogorov (BHK) interpretation of logical operators in intuitionistic logic. This gives us that the meaning of a proposition A is a proof, or evidence, for A, and this proof can be expressed in terms of proofs of the sub-parts of A, if there are any [Tro91]. The way to provide evidence for a given proposition is then shown below for types A and B and property P.

Logic	**Type Theory**
$A \implies B$	$A \to B$
$A \wedge B$	$A \times B$
$A \vee B$	$A + B$
$A \iff B$	$(A \to B) \times (B \to A)$
True	$\top$
False	$\bot$
$\neg A$	$A \to \bot$
$\forall x : A.P(x)$	$\Pi_{x:A} P(x)$
$\exists x : A.P(x)$	$\Sigma_{x:A} P(x)$

Table 0.2.1: Logic connectives and their type theoretic equivalents.

Evidence for $A \Rightarrow B$ is a function which transforms evidence for A to evidence for B. To provide evidence for $A \wedge B$ we provide a pair, with the first element being evidence for A and the second element being evidence for B. Evidence for $A \vee B$ consists of either evidence for A or evidence for B, i.e. the sum, or disjoint union, of A and B. $A \Leftrightarrow B$ follows similarly to $A \Rightarrow B$ but in both directions. True is represented by the type with one element $\top$, and false is represented by the empty type $\bot$. Evidence for $\neg A$ is a function which takes evidence for A and inhabits the empty type, i.e. leads to a contradiction. For the translation of the quantifiers $\forall$ (for all) and $\exists$ (exists), we require dependent types. MLTT differs from other type theories which also follow the Curry-Howard correspondence, by extending this correspondence to predicate logic using dependent types. To provide evidence for $\forall x : A.P(x)$, we use the dependent function type which assigns to any element x of A evidence for $P(x)$. To provide evidence for $\exists x : A.P(x)$, we use the dependent pair type which specifies a particular element x of A and provides evidence for $P(x)$.

0.2.3 Equality in type theory

We already mentioned that MLTT admits both definitional and propositional equalities. Different treatments of these equalities give rise to different type theories, and therefore different variants of MLTT.

In an **intensional type theory**, propositional equality, denoted interchangeably by Id

and ≡, is simply an inductive type family:

$$\begin{aligned}&\mathsf{data}\ \mathsf{Id}\ \{A : \mathsf{Type}\} : A \to A \to \mathsf{Type}\ \mathsf{where}\\ &\quad \mathsf{refl} : (a : A) \to \mathsf{Id}\ a\ a\end{aligned}$$

Its induction principle roughly states that if we can show that a property $P : \mathsf{Id}\ x\ y \to \mathsf{Type}$ holds for the reflexivity case, then we can prove that $P\ p$ holds for any equality $p : \mathsf{Id}\ x\ y$ — this is known as the J-rule, or path induction.[1]

$$\begin{aligned}&\mathsf{J} : \{A : \mathsf{Type}\}\ (P : (x\ y : A) \to \mathsf{Id}\ x\ y \to \mathsf{Type}) \to ((x : A) \to P\ x\ x\ (\mathsf{refl}\ x)) \to\\ &\quad (x\ y : A)\ (p : \mathsf{Id}\ x\ y) \to P\ x\ y\ p\end{aligned}$$

Having an intensional equality in practice is quite restrictive, since it becomes unclear how to talk about observational equality. For example, function extensionality becomes unprovable, so that even if we know that two functions behave in exactly the same way and are indistinguishable, we still cannot prove them to be equal because they might not have been constructed in the same way. One can always introduce function extensionality as an axiom, but this then introduces other 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:

$$\frac{\Gamma \vdash p : \mathsf{Id}\ x\ y}{\Gamma \vdash x = y}$$

While it is convenient to work with, extensional type theory does not have decidable type-checking [Hof95], and it implies that all types are h-sets. In extensional type theories, **uniqueness of identity proofs** (UIP) holds. UIP for a type A states that for any two elements a, a' of A, any two equalities of a and a' are equal:

$$\mathrm{UIP}_A : \prod_{a,a':A} \prod_{p,q:\mathsf{Id}\ a\ a'} \mathsf{Id}\ p\ q.$$

This disallows us from talking about types with equalities higher than h-props.

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. HoTT postulates Voevodsky's univalence axiom, which roughly states that isomorphic structures are (propositionally) equal. 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 type, i.e. equalities.

[1] There is an equivalent formulation of path induction which is sometimes more convenient to use, known as *based* path induction [Pau93], which starts by fixing an $a : A$, as follows.

$$\begin{aligned}&\mathsf{J'} : \{A : \mathsf{Type}\}\ (a : A)\ (P : (y : A) \to \mathsf{Id}\ a\ y \to \mathsf{Type}) \to (P\ a\ (\mathsf{refl}\ a)) \to\\ &\quad (y : A)\ (p : \mathsf{Id}\ a\ y) \to P\ y\ p\end{aligned}$$

While HoTT extends 'vanilla' MLTT with the univalence axiom and HITs, **cubical type theory** gives HoTT computational meaning; in particular, it makes the univalence axiom computable. It does so by replacing Martin-Löf's identity type with maps out of the interval (pre-)type.

0.2.4 The proof assistant `Agda`

A lot of the content in this thesis is supported by formalisations carried out in `Agda` or `Cubical Agda`, and throughout the thesis we will use codewords, examples, and code snippets written in (`Cubical`) `Agda`'s syntax. `Agda` [Agd25c] is a dependently-typed programming language and proof assistant implementing a version of intensional Martin-Löf type theory. Being based on the propositions as types paradigm, if we represent a proposition as a type, and an element (or term) of this type as a program, then if this program type checks, it constitutes a proof in intensional type theory that the proposition holds. `Cubical Agda` extends `Agda` with primitives from cubical type theory—for background on this flavour of `Agda` see Section 2.1.1.

`Agda` supports inductive data types, such as the unit type, the empty type, and the natural numbers.

```
data ⊤ : Type where
  tt : ⊤

data ⊥ : Type where

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

It also supports record types, which allow us to store named fields in the type, such as the Σ-type and the (coinductive) type of streams.

```
record Σ (A : Type) (B : A → Type) : Type where
  constructor _,_
  field
    fst : A
    snd : B fst


record Stream (A : Type) : Type where
  coinductive
  field
    hd : A
    tl : Stream A
```

`Agda` supports pattern matching on inductive types, and dually supports copattern matching on record and coinductive types. When defining a function whose domain is an inductive type, `Agda` allows us to pattern match on arguments of that type, i.e. generate separate clauses for each possible way that argument can be constructed. Dually, when defining a function whose codomain is a record or coinductive type, `Agda` allows us to copattern match on the function's result, i.e. separate the different projections of the result into clauses. Below, isEven uses pattern matching, while from uses copattern matching.

```
isEven : ℕ → Type
isEven zero = ⊤
isEven (suc zero) = ⊥
isEven (suc (suc n)) = isEven n
```

```
from : ℕ → Stream ℕ
hd (from n) = n
tl (from n) = from (suc n)
```

`Agda`'s built-in propositional equality type is (a variant of) the inductive type family Id shown in Section 0.2.3 [Agd25b]. `Agda` uses the syntax $(a : A) \to B\ a$ rather than $\Pi_{a:A} B\ a$; we will use both notations interchangeably throughout the thesis, but will prefer the latter when not writing `Agda` code. The syntax $\{a : A\} \to B\ a$ is also available in `Agda`, and denotes the same construction but with a being an implicit argument.

0.3 Overview of Inductive Types

An inductive type is a type given by a list of constructors, each specifying a way to form an element of the type. Defining types inductively is a central idea in Martin-Löf type theory (MLTT).

Simple types (a.k.a. Ordinary types) [Mar72]

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

- The unit type having one (non-recursive) constructor tt.

```
data ⊤ : Type where
  tt : ⊤
```

- 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
```

- The type of binary trees storing data in the leaves. Note that this type has a parameter A (placed before the colon) 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
```

Dependent types (a.k.a. Inductive families a.k.a. Indexed inductive types) [AM09; AGH+15]

Types that depend on a value from an input. Some examples are:

- The type of vectors of length $n : \mathbb{N}$ having elements of type A. Here, A is a parameter whereas $\mathbb{N}$ is an index (placed after the colon). 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 : A) (xs : Vec A n) → Vec A (suc n)
```

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

```
data Fin : ℕ → Type where
  fz : {n : ℕ} → Fin (suc n)
  fs : {n : ℕ} → Fin n → Fin (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. An example is the mutual definition of even and odd numbers. 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
  o-suc : even → odd
```

Inductive-inductive types (IITs) [Nor13]

Types with more than one sort, of type A : Type, B: $A \to$ 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 [Xie19].) An example is 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) → Ty (Γ , A) → Ty Γ
```

Quotient Inductive types (QITs) [AK16]

Types having not only point constructors as we have seen so far, but also path constructors of equalities on the type. These types are also explicitly set-truncated (hence the word *quotient*), so that their path constructors are only first-order equalities, and any higher order equalities are trivial. An example is 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) [ACD+18]

Types that combine the last two classes together, i.e. set-truncated types that allow induction-induction as well as path constructors. An example is the same Con Ty example given for IITs, except that Con and Ty Γ are now set-truncated, and 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) [Uni13, Chapter 6]

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. An example is 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, such as loop, loop ∘ loop, loop^{-1} ∘ loop ∘ loop, etc.

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

0.4 Category Theory Preliminaries

Categories

What we call a category is called a precategory in the HoTT book [Uni13].

Definition 0.4.1. A *category* $\underline{\mathbf{C}}$ consists of the following:

- A type $|\underline{\mathbf{C}}|$ of objects.
- For any objects $x, y : |\underline{\mathbf{C}}|$, an h-set of morphisms $\underline{\mathbf{C}}(x, y)$. An element f of this h-set will sometimes be denoted by $f : x \to y$ or $x \xrightarrow{f} y$.
- For any object $x : |\underline{\mathbf{C}}|$, an identity morphism $\mathrm{id}_x : \underline{\mathbf{C}}(x, x)$.
- For any objects $x, y, z : |\underline{\mathbf{C}}|$, a composition operator

$$\_ \circ \_ : \underline{\mathbf{C}}(y, z) \to \underline{\mathbf{C}}(x, y) \to \underline{\mathbf{C}}(x, z).$$

- For any objects $x, y : |\underline{\mathbf{C}}|$ and morphism $f : \underline{\mathbf{C}}(x, y)$, the equalities

$$\begin{aligned} \mathrm{idl} &: \mathrm{id}_y \circ f \equiv f \\ \mathrm{idr} &: f \circ \mathrm{id}_x \equiv f. \end{aligned}$$

- For any objects $w, x, y, z : |\underline{\mathbf{C}}|$ and morphisms $w \xrightarrow{f} x \xrightarrow{g} y \xrightarrow{h} z$, the equality

$$\mathrm{assoc} : h \circ (g \circ f) \equiv (h \circ g) \circ f. \qquad \Diamond$$

Powers and copowers

Every category $\underline{\mathbf{C}}$ having all products is powered over $\underline{\mathbf{Set}}$. This means that we have an operator $\prod^{\underline{\mathbf{C}}}$ which, for every $I : \mathrm{Set}$ and $X : \left|\underline{\mathbf{C}}\right|$, gives an object $\prod^{\underline{\mathbf{C}}}_{i:I} X : \left|\underline{\mathbf{C}}\right|$, called the *power* of X by I, corresponding to the $|I|$-fold product of X with itself, where $|I|$ is the cardinality of I.

Dually, every category having all coproducts is copowered over $\underline{\mathbf{Set}}$. We have an operator $\sum^{\underline{\mathbf{C}}}$ which, for every $I : \mathrm{Set}$ and $X : \left|\underline{\mathbf{C}}\right|$, gives an object $\sum^{\underline{\mathbf{C}}}_{i:I} X : \left|\underline{\mathbf{C}}\right|$, called the *copower* of X by I, corresponding to the $|I|$-fold coproduct of X with itself, where $|I|$ is the cardinality of I.

We will make use of dependent versions of the power and copower operators, i.e. for $I : \mathrm{Set}$ and $X : I \to \left|\underline{\mathbf{C}}\right|$ we have

$$\begin{aligned} &\prod_{i:I}^{\underline{\mathbf{C}}} (X\, i) : \left|\underline{\mathbf{C}}\right| \\ &\sum_{i:I}^{\underline{\mathbf{C}}} (X\, i) : \left|\underline{\mathbf{C}}\right|, \end{aligned}$$

which correspond to the $|I|$-fold product and coproduct of X respectively, but now X depends on $i : I$.

Category of elements

Given a functor $F : \underline{\mathbf{C}} \to \underline{\mathbf{Set}}$, 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 $u : c \to d$ and $f : (F\,u)\,F_c \equiv F_d$.

Ends

In order to make precise some notation used in the thesis, 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.

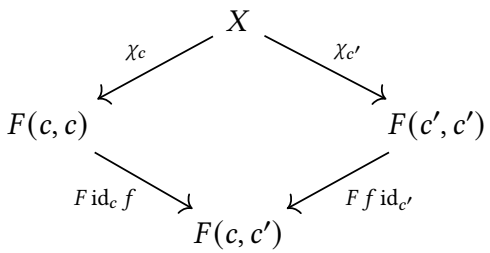
Definition 0.4.2. Given a category $\underline{\mathbf{C}}$, a *profunctor*[2] $F : \underline{\mathbf{C}}^{\mathrm{op}} \times \underline{\mathbf{C}} \to \underline{\mathbf{Set}}$ is defined as follows:

- 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 functor laws for identity and composition are stated as $F(\mathrm{id}_c, \mathrm{id}_d) \equiv \mathrm{id}_{F(c,d)}$, and for $c_1 \xrightarrow{g'} c_2 \xrightarrow{g} c_3$ and $d_1 \xrightarrow{f'} d_2 \xrightarrow{f} d_3$, $F((g \circ g'), (f \circ f')) \equiv F(g, f') \circ F(g', f)$ 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 0.4.3. Given a profunctor $F : \underline{\mathbf{C}}^{\mathrm{op}} \times \underline{\mathbf{C}} \to \underline{\mathbf{Set}}$, a *wedge* on F is an object $X : \mathrm{Set}$ 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.

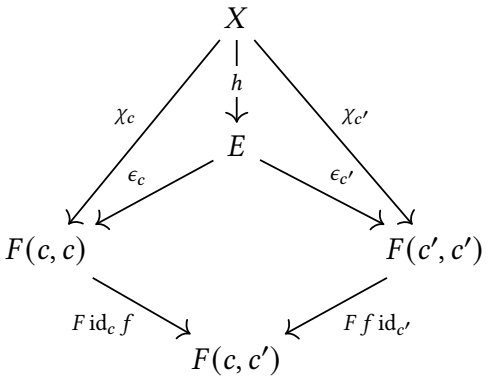


◊

Definition 0.4.4. 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

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

below.



◊

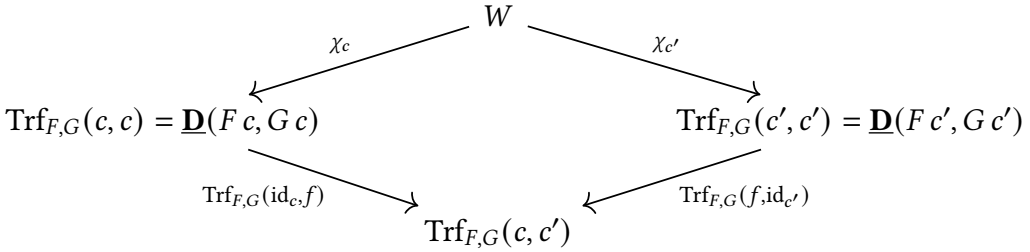
Example 0.4.5. If $\underline{\mathbf{C}}$ is a locally small category (i.e. all of its homsets are small sets), then we can define the profunctor

$$\begin{aligned}
\mathrm{Hom}_{\underline{\mathbf{C}}}(-,-) &: \underline{\mathbf{C}}^{\mathrm{op}} \times \underline{\mathbf{C}} \to \underline{\mathbf{Set}} \\
\mathrm{Hom}_{\underline{\mathbf{C}}}(c,c') &: \mathrm{Set} \\
\mathrm{Hom}_{\underline{\mathbf{C}}}(c,c') &\coloneqq \underline{\mathbf{C}}(c,c') \\
\mathrm{Hom}_{\underline{\mathbf{C}}}(c \xrightarrow{f} c', d \xrightarrow{g} d') &: \mathrm{Hom}_{\underline{\mathbf{C}}}(c',d) \to \mathrm{Hom}_{\underline{\mathbf{C}}}(c,d') \\
\mathrm{Hom}_{\underline{\mathbf{C}}}(c \xrightarrow{f} c', d \xrightarrow{g} d')\, h &\coloneqq g \circ h \circ f
\end{aligned}$$

Now consider functors $F, G : \underline{\mathbf{C}} \to \underline{\mathbf{D}}$. We can define the functor $\mathrm{Trf}_{F,G}$ of transformations $F \to G$ as below.

$$\begin{aligned}
\mathrm{Trf}_{F,G} &: \underline{\mathbf{C}}^{\mathrm{op}} \times \underline{\mathbf{C}} \to \underline{\mathbf{Set}} \\
\mathrm{Trf}_{F,G}(c,c') &: \mathrm{Set} \\
\mathrm{Trf}_{F,G}(c,c') &\coloneqq \underline{\mathbf{D}}(F\,c, G\,c') \\
\mathrm{Trf}_{F,G}(c \xrightarrow{f} c', d \xrightarrow{g} d') &: \underline{\mathbf{D}}(F\,c', G\,d) \to \underline{\mathbf{D}}(F\,c, G\,d') \\
\mathrm{Trf}_{F,G}(c \xrightarrow{f} c', d \xrightarrow{g} d')\, h &\coloneqq G\,g \circ h \circ F\,f
\end{aligned}$$

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



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

$$\mathrm{Trf}_{F,G}(\mathrm{id}_c, f)\, h_1 = G\,f \circ h_1 \circ F\,\mathrm{id}_c$$

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

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

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

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 $\mathrm{Trf}_{F,G}$ represents a natural transformation from F to G, an end of $\mathrm{Trf}_{F,G}$ 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

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

This integral notation will be used throughout the thesis. ◊

0.5 Notation

$\bot$	The empty type.
$\top$	The unit type with the singular element tt.
Fin n	The type of finite sets of size n, having elements fz, fs fz, ...
$:=$	Definitional equality, specifically used when first defining something.
$=$	Definitional equality.
$\equiv$ or Id	Propositional equality.
$\cong$	Equivalence of types. In HoTT, there are several different notions of type equivalence, which however are all equivalent and can be used interchangeably. When we are working with h-sets, the notion of equivalence boils down to isomorphism.
J	The elimination principle for Martin-Löf's identity type.
Set	The universe of homotopy-sets (h-sets), or 0-types, i.e. the universe of types X such that for any two elements $x, y : X$, if $p, q : x \equiv y$ then $p \equiv q$.

$\underline{\mathbf{Set}}$	The category whose objects are (small) h-sets and whose morphisms are functions.
$\underline{\mathbf{Set}}^n$	The n-ary product category of $\underline{\mathbf{Set}}$.
$[\underline{\mathbf{C}}, \underline{\mathbf{C}}]$	The category of endofunctors on $\underline{\mathbf{C}}$ whose morphisms are natural transformations.
$\lvert\underline{\mathbf{C}}\rvert$	The type of objects of the category $\underline{\mathbf{C}}$.
$A \times B$	The type of products of types $A, B : \mathsf{Type}$, with elements (a, b) where $a : A$ and $b : B$. The first and second projections out of this type are $\pi_1 : A \times B \to A$ and $\pi_2 : A \times B \to B$.
$A \to B$	The type of functions from $A : \mathsf{Type}$ to $B : \mathsf{Type}$, with elements $\lambda x \to f\ x$ for $f : A \to B$.
$A + B$	The sum type of $A, B : \mathsf{Type}$, with elements $\mathsf{inl}\ a$ for $a : A$ and $\mathsf{inr}\ b$ for $b : B$.
$\sum_{a:A} B\ a$; $\sum(a : A)(B\ a)$	The dependent pair type where $A : \mathsf{Type}$ and $B : A \to \mathsf{Type}$, with elements (a, b) where $a : A$ and $b : B\ a$.
$\prod_{a:A} B\ a$; $\prod(a : A)(B\ a)$	The dependent function type where $A : \mathsf{Type}$ and $B : A \to \mathsf{Type}$, with elements $\lambda x \to f\ x$ for $f : A \to B$.
$F\ (X \xrightarrow{f} Y)$	The morphism map of the functor F acting on morphism $f : X \to Y$, used to clearly show the domain and codomain of f.
$\mathrm{ap}_f\ p$	For $A, B : \mathsf{Type}$, $f : A \to B$, elements $a, a' : A$, and a proof $p : a \equiv a'$, then $\mathrm{ap}_f\ p$ has type $f\ a \equiv f\ a'$.
$a \approx b$	A dependent path from a to b over some path p. This is usually followed by a yellow box specifying p. In the HoTT book [Uni13], this is denoted by transport $(p, a) = b$.

Chapter 1

A Survey of Containers

The theory of containers [AAG03; AAG04; AAG05; Abb03; AM09; AGH+15] was developed to capture the concept of strictly positive data types in programming, and to support total generic programming with data types. It has proved very useful in providing semantics for inductive types and inductive families. Much of the original development of containers ([AAG03; AAG04; AAG05; Abb03]) uses a heavily categorical language, where containers are presented as constructions in the internal language of locally cartesian closed categories (LCCCs) with disjoint coproducts and W-types, referred to as Martin-Löf categories, and a standard set-theoretic metatheory is used.

Throughout this thesis, we depart from the use of LCCCs and instead present containers and their established results in type theory, in a similar style to that used in [AM09; AGH+15]. Since containers will be used extensively in the coming chapters, this chapter motivates their use, defines the different kinds of containers, and includes proofs of their closure properties, doing so in the language of type theory.

Containers are a special case of the polynomial functors studied by Gambino and Hyland [GH04]. Specifically, polynomial functors on the LCCC of sets are equivalent to indexed container functors, and in a special case we get (ordinary) container functors (when the object I in [GH04, Section 5] is the terminal object).

1.1 Overview of Containers

We start by giving an informal overview of the different kinds of containers we will consider.

A container is a specific representation of an inductive or coinductive type, which also has an associated *signature functor* representation. Semantically, an inductive type corresponds to the initial algebra of its signature functor (and a coinductive type is the

terminal coalgebra of the same functor), see [Uni13, Section 5.4] and [Hag87, Section 1.3]. We will also refer to initial algebras as *least fixed points* and terminal coalgebras as *greatest fixed points* of a signature functor.

We define operators μ and ν to be least fixed point and greatest fixed point operators respectively. They have different types for different kinds of containers. For example, we can take a partial fixed point of an n-ary signature functor $F : \underline{\mathbf{Set}}^{n+1} \to \underline{\mathbf{Set}}$ to obtain $\mu F, \nu F : \underline{\mathbf{Set}}^{n} \to \underline{\mathbf{Set}}$, where μF refers to the carrier of F's initial algebra and νF refers to the carrier of F's terminal coalgebra. Similarly, we can take a partial fixed point of an indexed signature functor $F : \underline{\mathbf{Set}}^{I+J} \to \underline{\mathbf{Set}}^{J}$ to obtain $\mu F, \nu F : \underline{\mathbf{Set}}^{I} \to \underline{\mathbf{Set}}^{J}$. The examples given here are signature functors whose least fixed points are the inductive types indicated, all of which can be expressed as different kinds of containers.

The simplest kind of containers is *unary containers*, which represent endofunctors on $\underline{\mathbf{Set}}$. One such example is the endofunctor for natural numbers $\mathbb{N}$.

$$
\begin{aligned}
&F_{\mathbb{N}} : \mathrm{Set} \to \mathrm{Set} \\
&F_{\mathbb{N}}\, X \coloneqq \top + X
\end{aligned}
$$

n-ary containers are containers having a finite number n of parameters, which represent functors of type $\underline{\mathbf{Set}}^{n} \to \underline{\mathbf{Set}}$, where $\underline{\mathbf{Set}}^{n}$ is the n-ary product category of $\underline{\mathbf{Set}}$. Unary containers are trivially n-ary containers for $n = 1$. An example is the signature functor for the List data type, parameterised by A which is the type of entries of the list.

$$
\begin{aligned}
&F_{\mathsf{List}} : \mathrm{Set}^2 \to \mathrm{Set} \\
&F_{\mathsf{List}}\, (A, X) \coloneqq \top + A \times X
\end{aligned}
$$

Indexed containers represent functors of type $\underline{\mathbf{Fam}}_I \to \underline{\mathbf{Fam}}_J$, where $\underline{\mathbf{Fam}}_I$ refers to the category of I-indexed families of sets. These extend the previous types of containers in order to be able to represent inductive families. One example is the signature functor for the family Vec of vectors over some type A, where $I = \top + \mathbb{N}$ and $J = \mathbb{N}$.

$$
\begin{aligned}
&F_{\mathsf{Vec}} : \mathrm{Set} \to (\mathbb{N} \to \mathrm{Set}) \to (\mathbb{N} \to \mathrm{Set}) \\
&F_{\mathsf{Vec}}\, A\, X\, n \coloneqq (n \equiv \mathsf{zero}) + \sum_{m:\mathbb{N}} (n \equiv \mathsf{suc}\, m) \times A \times X\, m
\end{aligned}
$$

Generalised containers represent functors of type $\underline{\mathbf{C}} \to \underline{\mathbf{Set}}$ for an arbitrary category $\underline{\mathbf{C}}$. Examples include functor representations of data type constructors. For instance, consider the successor constructor for Fin.

$$\mathsf{fs} : \{n : \mathbb{N}\} \to \mathsf{Fin}\ n \to \mathsf{Fin}\ (\mathsf{suc}\ n)$$

Following [ACD+18], we can represent this constructor via two functors, one encoding the constructor's arguments (which we can call F_L), and another encoding the constructor's target type. The former can be represented as below.

$$F_L : (\mathbb{N} \to \mathrm{Set}) \to \mathrm{Set}$$

$$F_L\ X \coloneqq \sum_{n:\mathbb{N}} (X\ n)$$

Below is a table of which closure properties hold for the different kinds of containers introduced above. The symbol ✓ indicates that the property holds, ❋ indicates that the property holds under extra conditions, and **N/A** indicates that in their current formulation, this property does not even make sense to state. A theorem or equation number indicates that a proof or statement can be found in this thesis, while a citation directs the reader to a proof elsewhere, and having neither means the result follows almost identically to a simpler counterpart.

	unary	n-ary	indexed	generalised
products	✓ Thm. 1.3.13	✓	✓ Eqn. 1.4.6	❋ Thm. 1.5.5
coproducts	✓ Thm. 1.3.13	✓	✓ Eqn. 1.4.7	✓ Thm. 1.5.4
composition	✓ Eqn. 1.3.16	✓ Thm. 1.3.15	✓ [AGH+15]	**N/A**
exponentiation	✓ Thm. 1.3.19	✓ Eqn. 1.3.23	✓ Thm. 1.4.8	**N/A**
fixed points	**N/A**[1]	✓ Thms. 2.3.1 and 2.3.4	✓ [AM09; AGH+15]	**N/A**[1]

[1] In the unary and generalised cases, we cannot iterate the process of taking fixed points.

1.2 Strictly Positive Types

Containers are used to express strict positivity semantically. Strict positivity is a condition we impose on inductive and coinductive definitions, in order to be able to properly express their induction principle without leading to inconsistencies in our theory (see [Uni13, Section 5.6]). It is roughly the restriction whereby the constructors of X are only allowed to have X appear in input types that are arrows, if it appears on their right. This disallows us from defining types like Contra below, where Contra appears on the left of an arrow.

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

On the other hand, we are allowed to define types like InfTree, the type of trees that are infinitely wide (note that InfTree appears on the right of the input arrow (ℕ → InfTree) of node).

```
data InfTree : Set where
  leaf : InfTree
  node : (ℕ → InfTree) → InfTree
```

One practical reason why types that are not strictly positive are problematic is shown in [Uni13, Section 5.6]. If we accept Contra as a valid inductive type, then assuming classical logic leads to a Cantorian-style paradox.

A syntactic approach to strict positivity was taken by [AA00], where a set of rules detailing when a type is strictly positive is presented. In contrast, containers provide a categorical semantics for strict positivity.

The idea behind the basic definition of a container arises from the W-type, first presented by Martin-Löf [Mar82; Mar84]. W is the type of well-founded, labelled trees. A tree of type W can be infinitely branching, but every path in the tree is finite. W takes two parameters $S : \mathsf{Set}$ and $P : S \to \mathsf{Set}$. We think of S as the type of shapes of the tree, and for a given shape $s : S$, the tree has $(P\ s)$-many positions.

```
data W (S : Set) (P : S → Set) : Set where
  sup-W : (s : S) → (P s → W S P) → W S P
```

The key property of W is that it is the universal type for strictly positive inductive types, i.e. any strictly positive type can be expressed using W.

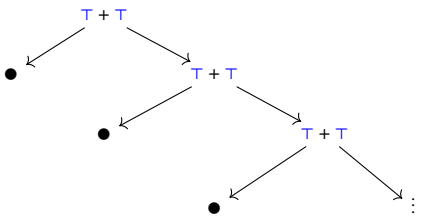
Example 1.2.1. We encode the type of natural numbers $\mathbb{N}$ by defining S and P as below.

$$\begin{aligned} \mathsf{S} &= \top + \top \\ \mathsf{P}\ (\mathsf{inl}\ \_) &= \bot \\ \mathsf{P}\ (\mathsf{inr}\ \_) &= \top \end{aligned}$$

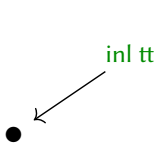
S encodes the possible constructors we can choose (inl is for zero, inr is for succ), and P encodes the number of subtrees (or recursive arguments) each choice of S has. Thus W S P encodes $\mathbb{N}$. The constructors zero and succ are encoded as zz and ss respectively.

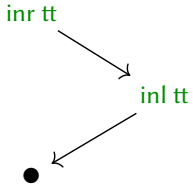
```
zz : W S P
zz = sup-W (inl tt) (λ ())

ss : W S P → W S P
ss n = sup-W (inr tt) (λ _ → n)
```

The type $\mathbb{N}$ can be viewed as the type of well-founded trees whose nodes are labelled by an $s : \mathsf{S}$ and which have $\mathsf{P}\ s$-many subtrees.



For example, the natural numbers zero and succ zero are represented by the trees below respectively.





◊

The takeaway from the above example is that the parameters S and P specify the type $\mathbb{N}$ entirely. Any strictly positive data type can be fully represented by a set of 'shapes' S and a family of 'positions' P over those shapes, at which data can be stored. This is precisely what inspires the definition of a container.

1.3 Unary and n-ary Containers

Definition 1.3.1. A *(unary) container* is given by a pair of types $S : \mathrm{Set}$ and $P : S \to \mathrm{Set}$, which we write as $S \triangleleft P$. ◊

In practice, many data types are parameterised by one or more types. For example, $\mathsf{List}\ (A : \mathrm{Set}) : \mathrm{Set}$ and $\mathsf{Vec}\ (A : \mathrm{Set}) : \mathbb{N} \to \mathrm{Set}$ are both parameterised by the type A of data to be stored in them. In order to be able to reason about such parameterised data types, as well as to construct fixed points of containers, we will need containers having n parameters, or n-ary containers.

Definition 1.3.2. An *n-ary container* is given by a pair $S : \mathrm{Set}$ and $\mathbf{P} : \mathsf{Fin}\ n \to S \to \mathrm{Set}$, which we write as $S \triangleleft \mathbf{P}$. ◊

Example 1.3.3. The signature functor for $\mathbb{N}$ from Section 1.1, $F_{\mathbb{N}}$, has a container representation as follows. Precisely, we mean that $F_{\mathbb{N}}$ is equivalent to the functor interpretation of the below container.

$$\top + \top \triangleleft \lambda \begin{cases} \mathsf{inl}\ \mathsf{tt} \to \bot \\ \mathsf{inr}\ \mathsf{tt} \to \top \end{cases}$$

The shape $\top + \top$ represents a choice between having a zero or suc. If the shape is inl tt, this represents having zero, so the type of positions is $\bot$ as there is no more data to supply. If the shape is inr tt this represents having a successor, so the type of positions is $\top$ since there is one position for a recursive argument: the predecessor. ◊

Example 1.3.4. Given an h-set A, the (unary) container representation of $\mathsf{List}\ A : \mathrm{Set}$ is given by $\mathbb{N} \triangleleft \mathsf{Fin}$. (The precise meaning of this is that $[\![\mathbb{N} \triangleleft \mathsf{Fin}]\!]\, A \cong \mu X.F_{\mathsf{List}}(A, X)$, see Example 1.3.8.) The shape of a list is a natural number n representing its length, and there are n-many positions for data to be stored in a list, represented by $\mathsf{Fin}\ n$, the type of finite sets of size n. ◊

To every (unary) container $S \triangleleft P$ we associate a functor which 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.

Definition 1.3.5. The *container functor* associated to a container $S \triangleleft P$ is the functor $[\![S \triangleleft P]\!] : \underline{\mathbf{Set}} \to \underline{\mathbf{Set}}$ with the following actions on objects and morphisms.

- Given an $X : \mathrm{Set}$, $[\![S \triangleleft P]\!]\, X \coloneqq \sum_{s:S} (P\,s \to X)$.
- Given $X, Y : \mathrm{Set}$ and a morphism $f : X \to Y$,

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

  is defined for $s : S$ and $g : P\,s \to X$ as

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

◊

Container functors are also defined for n-ary containers.

Definition 1.3.6. The *container functor* associated to an n-ary container $S \triangleleft \mathbf{P}$ is the functor $[\![S \triangleleft \mathbf{P}]\!] : \underline{\mathbf{Set}}^n \to \underline{\mathbf{Set}}$ with the following actions on objects and morphisms.

- Given an $\mathbf{X} : \mathrm{Set}^n$, $[\![S \triangleleft \mathbf{P}]\!]\,\mathbf{X} \coloneqq \sum_{s:S}\Big(\prod_{\{i:\mathsf{Fin}\,n\}} \mathbf{P}\,i\,s \to \mathbf{X}\,i\Big)$.
- Given $\mathbf{X}, \mathbf{Y} : \mathrm{Set}^n$ and a morphism $f : \prod_{\{i:\mathsf{Fin}\,n\}} \mathbf{X}\,i \to \mathbf{Y}\,i$,

$$[\![S \triangleleft \mathbf{P}]\!]\,f\,(s, g) \coloneqq (s, f \circ g)$$

  for $s : S$ and $g : \prod_{\{i:\mathsf{Fin}\,n\}} \mathbf{P}\,i\,s \to \mathbf{X}\,i$. ◊

Above, $\mathbf{P}$ can be thought of as n families of families over S. We will sometimes write $P_0, P_1, \ldots$ instead of $\mathbf{P}\ i_0, \mathbf{P}\ i_1, \ldots$ to enumerate the families over S.

Example 1.3.7. Container functors allow us to view strictly positive types simply as memory locations in which data can be stored. The container functor associated to $\mathbb{N} \triangleleft \mathsf{Fin}$ allows us to represent concrete lists. The list of Chars [‘r’, ‘e’, ‘d’] is represented as $(3, (0 \mapsto \text{‘}r\text{’}; 1 \mapsto \text{‘}e\text{’}; 2 \mapsto \text{‘}d\text{’})) : \sum_{n:\mathbb{N}} (\mathsf{Fin}\,\mathrm{n} \to \mathsf{Char})$. ◊

Example 1.3.8. The signature functor of List from Section 1.1, F_{List}, defined as

$$F_{\mathsf{List}}\,(A, X) = \top + A \times X$$

has a *binary* container representation. This means we can define S, P_0, and P_1 as shown below, and get that

$$[\![\mathsf{S} \triangleleft (\mathsf{P}_0, \mathsf{P}_1)]\!]\,(A, X) \cong \top + A \times X.$$

$$\begin{array}{lll} \mathsf{S} : \mathsf{Set} & \mathsf{P}_0 : \mathsf{S} \to \mathsf{Set} & \mathsf{P}_1 : \mathsf{S} \to \mathsf{Set} \\ \mathsf{S} \coloneqq \top + \top & \mathsf{P}_0\ (\mathsf{inl}\ \mathsf{tt}) \coloneqq \bot & \mathsf{P}_1\ (\mathsf{inl}\ \mathsf{tt}) \coloneqq \bot \\ & \mathsf{P}_0\ (\mathsf{inr}\ \mathsf{tt}) \coloneqq \top & \mathsf{P}_1\ (\mathsf{inr}\ \mathsf{tt}) \coloneqq \top \end{array}$$

The type of shapes S reflects the fact that there are two ways to construct a list, i.e. as either nil or cons. P_0 defines positions for the parameter A and P_1 defines positions for the recursive argument X. Hence the container representation of F_{List} is

$$[\![(\top + \top) \triangleleft (\mathsf{P}_0, \mathsf{P}_1)]\!].$$

Example 1.3.4 corresponds to taking the least fixed point of this functor with respect to X. ◊

1.3.1 The category of containers

Now that we have defined containers and container functors, we can view them as objects in two different categories, namely the category **Cont** of containers that we define below, and the functor category [**Set**, **Set**]. We also show that the operation $[\![\_]\!]$ extends to a functor **Cont** $\to$ [**Set**, **Set**] relating the two categories.

Definition 1.3.9. The *category of containers*, which we refer to hereafter as **Cont**, is defined as follows:

- Objects are containers as defined in Definition 1.3.1, i.e. pairs $S : \mathrm{Set}$ and $P : S \to \mathrm{Set}$, written as $S \triangleleft P$.
- Morphisms $(S \triangleleft P) \to (T \triangleleft Q)$ are pairs of shape and position morphisms:

$$\begin{aligned} u &: S \to T \\ f &: \prod_{s:S} Q\,(u\,s) \to P\,s \end{aligned}$$

  written as $u \triangleleft f$. For a container morphism α, we write α_s for the shape component and α_p for the position component.
- Composition of morphisms $S \triangleleft P \xrightarrow{f \triangleleft g} T \triangleleft Q \xrightarrow{h \triangleleft i} U \triangleleft V$ is defined as follows:

$$\begin{aligned} (h \triangleleft i) \circ (f \triangleleft g) &: S \triangleleft P \to U \triangleleft V \\ ((h \triangleleft i) \circ (f \triangleleft g))_s &: S \to U \\ ((h \triangleleft i) \circ (f \triangleleft g))_s &\coloneqq h \circ f \\ ((h \triangleleft i) \circ (f \triangleleft g))_p &: \prod_{s:S} V\,(h \circ f\,(s)) \to P\,s \\ ((h \triangleleft i) \circ (f \triangleleft g))_p\,s\,r &\coloneqq g\,s\,(i\,(f\,s)\,r). \end{aligned}$$

- The identity morphism $\mathrm{id}_{S \triangleleft P} : S \triangleleft P \to S \triangleleft P$ is defined as follows:

$$\begin{aligned} \mathrm{id}_s &: S \to S \\ \mathrm{id}_s\ s &\coloneqq s \\ \mathrm{id}_p &: \prod_{s:S} P\,(\mathrm{id}_s\ s) \to P\,s \\ \mathrm{id}_p\ s\,p &\coloneqq p. \end{aligned}$$

Associativity of composition and the left and right identity laws hold by definition. ◊

More generally, we can define the category of n-ary containers $\mathbf{Cont}_n$, the only difference being that now P and Q above have an extra parameter $i : \mathrm{Fin}\,n$.

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 is motivated by the fact that we can always pinpoint the source of a given target position, but not necessarily the reverse.

Example 1.3.10. The tail function on lists, defined as follows

```
tail : List A → List A
tail [] = []
tail (x :: xs) = xs
```

can be represented as a container morphism from the list container $\mathbb{N} \triangleleft \mathsf{Fin}$ 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 = fs r
```

The interesting case is 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 — the head of the list has no target position. ◊

We are now interested in relating a container $S \triangleleft P$ in **Cont** to its functorial interpretation $[\![S \triangleleft P]\!]$. We do this by defining the container extension functor $[\![\_]\!]$.

Definition 1.3.11. The *container extension functor* $[\![\_]\!] : \mathbf{Cont} \to [\mathbf{Set}, \mathbf{Set}]$ is defined as follows:

- Given an object $S \triangleleft P$ in **Cont**, this is mapped to its container functor $[\![S \triangleleft P]\!]$ as defined in Definition 1.3.5.
- Given a morphism $(u \triangleleft f) : (S \triangleleft P) \to (T \triangleleft Q)$ in **Cont**, this is mapped to the natural transformation $[\![u \triangleleft f]\!] : [\![S \triangleleft P]\!] \to [\![T \triangleleft Q]\!]$ with components

$$[\![u \triangleleft f]\!]_X : [\![S \triangleleft P]\!]\, X \to [\![T \triangleleft Q]\!]\, X$$
$$[\![u \triangleleft f]\!]_X\, (s, h) \coloneqq (u\, s, h \circ (f\, s))$$

  for any $X : \mathrm{Set}$, $s : S$, and $h : P\, s \to X$. ◊

More generally, we can also define the functor $[\![\_]\!] : \mathbf{Cont}_n \to [\mathbf{Set}^n, \mathbf{Set}]$, which sends n-ary containers to their corresponding n-ary container functors.

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 1.3.11. The corresponding notion for containers is container morphisms as seen in Definition 1.3.9. It turns out that there is a bijective correspondence between natural transformations on container functors and container morphisms. Since natural transformations on container functors are simply polymorphic functions between strictly positive types, this gives us that every such function is uniquely represented as a container morphism. This correspondence is expressed by the fact that the container extension functor $[\![\_]\!]$ is full and faithful.

Theorem 1.3.12. *The functor* $[\![\_]\!] : \mathbf{Cont} \to [\mathbf{Set}, \mathbf{Set}]$ *is full and faithful.*

Proof. We need to show that the morphism part of $[\![\_]\!]$ is a bijection, i.e. that for any $S \triangleleft P$ and $T \triangleleft Q : |\underline{\mathbf{Cont}}|$,

$$[\![\_]\!] : \underline{\mathbf{Cont}}(S \triangleleft P, T \triangleleft Q) \to [\underline{\mathbf{Set}}, \underline{\mathbf{Set}}]([\![S \triangleleft P]\!], [\![T \triangleleft Q]\!])$$

between homsets is a bijection. We construct an isomorphism between these homsets as follows.

$$\begin{aligned}
&[\underline{\mathbf{Set}}, \underline{\mathbf{Set}}]([\![S \triangleleft P]\!], [\![T \triangleleft Q]\!]) \\
&= \int_{X:\mathrm{Set}} ([\![S \triangleleft P]\!]\, X \to [\![T \triangleleft Q]\!]\, X) \\
&= \int_{X:\mathrm{Set}} \sum_{s:S} (P\, s \to X) \to [\![T \triangleleft Q]\!] X && \text{expanding definition of } [\![S \triangleleft P]\!]\, X \\
&\cong \int_{X:\mathrm{Set}} \prod_{s:S} (P\, s \to X) \to [\![T \triangleleft Q]\!] X && \text{currying in } \underline{\mathbf{Set}}: \\
&&& \Pi\,(\Sigma\, A\, B)\, C \cong \Pi\, A\, (\Pi\, B\, C) \\
&\cong \prod_{s:S} \int_{X:\mathrm{Set}} (P\, s \to X) \to [\![T \triangleleft Q]\!] X && \textstyle\int \text{ and } \Pi \text{ commute} \\
&\cong \prod_{s:S} [\![T \triangleleft Q]\!]\, (P\, s) && \text{covariant Yoneda lemma:} \\
&&& \text{for } F : \underline{\mathbf{C}} \to \underline{\mathbf{Set}}, A : |\underline{\mathbf{C}}|, \\
&&& \textstyle\int_{X:|\underline{\mathbf{C}}|} (\underline{\mathbf{C}}(A, X), F\, X) \cong F\, A \\
&= \prod_{s:S} \sum_{t:T} (Q\, t \to P\, s) && \text{expanding definition of } [\![T \triangleleft Q]\!]\, X \\
&\cong \sum_{u\,:\,S \to T} \left(\prod_{s:S} Q(u\, s) \to P\, s \right) && \text{distributivity of } \Pi \text{ over } \Sigma: \\
&&& \textstyle\prod_{a:A} \sum_{b:B\,a} C\, a\, b \;\cong \\
&&& \textstyle\sum (f :\; \prod_{a:A} B\, a)(\prod_{a:A} C\, a\, (f\, a)) \\
&= \underline{\mathbf{Cont}}(S \triangleleft P, T \triangleleft Q) && \text{definition of container mor-} \\
&&& \text{phism}
\end{aligned}$$

□

As seen in Example 1.3.10, a function on lists (that is polymorphic on the type of data A to be stored in the list) is a pair of functions $u : \mathbb{N} \to \mathbb{N}$ and $f : \prod_{n:\mathbb{N}} \mathsf{Fin}\,(u\, n) \to \mathsf{Fin}\, n$. The function u takes the old list length to the new length, and f assigns to every entry in the new list its position in the old list. Theorem 1.3.12 shows that *any* polymorphic function between any two strictly positive inductive types can be expressed in this form, i.e. as a map between shapes and positions of the respective inductive types. This makes container morphisms a canonical way of representing polymorphic functions.

Another point to note is that the above isomorphism goes from something which lives in $[\underline{\mathbf{Set}}, \underline{\mathbf{Set}}]$ that is large, to something that lives in $\underline{\mathbf{Cont}}$ which is small. This makes container morphisms a useful representation of polymorphic functions on strictly positive types when we want to avoid size issues.

We now turn our attention to the categories $\underline{\mathbf{Cont}}$ and $\underline{\mathbf{Cont}}_n$ and look at their closure properties. Namely, we show that $\underline{\mathbf{Cont}}$ is closed under products, coproducts, and exponentiation. Extending the first two results to $\underline{\mathbf{Cont}}_n$ is an easy exercise, but we show explicitly that $\underline{\mathbf{Cont}}_n$ is also closed under exponentiation. We show that we can

partially apply n-ary containers, which we also refer to as composing n-ary containers. n-ary containers are also closed under taking fixed points (i.e. initial algebras and terminal coalgebras) — this is covered in detail in Chapter 2.

1.3.2 Products and coproducts

Any full and faithful functor reflects limits and colimits in its codomain [Rie16, Lemma 3.3.5]. We show that products and coproducts of objects in the image of $[\![\_]\!]$ in $[\underline{\mathbf{Set}}, \underline{\mathbf{Set}}]$ are themselves in the image of $[\![\_]\!]$. Since $[\![\_]\!]$ is full and faithful, we can reflect them along $[\![\_]\!]$ to obtain products and coproducts in $\underline{\mathbf{Cont}}$.

Theorem 1.3.13. *$\underline{\mathbf{Cont}}$ inherits infinite products and coproducts from $[\underline{\mathbf{Set}}, \underline{\mathbf{Set}}]$.*

Proof. We take I to be a possibly infinite indexing set, and consider the I-ary product and coproduct of a family of unary containers. We assume $X : \mathrm{Set}$.

We start by computing the product of container functors.

$$
\begin{aligned}
&\prod_{i:I} \left([\![S\, i \triangleleft P\, i]\!]\, X \right) \\
&= \prod_{i:I} \left(\sum_{s : S\, i} (P\, i\, s \to X) \right) && \text{definition of } [\![\_]\!] \\
&\cong \sum \left(s : \prod_{i:I} S\, i \right) \left(\prod_{i:I} (P\, i\, (s\, i) \to X) \right) && \text{distributivity of } \Pi \text{ over } \Sigma \\
&\cong \sum \left(s : \prod_{i:I} S\, i \right) \left(\left(\sum_{i:I} P\, i\, (s\, i) \right) \to X \right) && \text{uncurrying in } \underline{\mathbf{Set}} \\
&= \left[\!\!\left[\left(s : \prod_{i:I} S\, i \right) \triangleleft \sum_{i:I} P\, i\, (s\, i) \right]\!\!\right] X && \text{definition of } [\![\_]\!]
\end{aligned}
$$

Next, we compute the coproduct of container functors.

$$
\begin{aligned}
&\sum_{i:I} \left([\![S\, i \triangleleft P\, i]\!]\, X \right) \\
&= \sum_{i:I} \left(\sum_{s : S\, i} (P\, i\, s \to X) \right) && \text{definition of } [\![\_]\!] \\
&\cong \sum \left((i, s) : \sum_{i:I} S\, i \right) (P\, i\, s \to X) && \text{associativity of } \Sigma \\
&= \left[\!\!\left[\left((i, s) : \sum_{i:I} S\, i \right) \triangleleft P\, i\, s \right]\!\!\right] X && \text{definition of } [\![\_]\!]
\end{aligned}
$$

Reflecting along $[\![\_]\!]$, we get that

$$
\prod_{i:I}^{\underline{\mathbf{Cont}}} (S\, i \triangleleft P\, i) = \left(s : \prod_{i:I} S\, i \right) \triangleleft \sum_{i:I} P\, i\, (s\, i)
$$

$$\sum_{i:I}^{\underline{\mathbf{Cont}}}(S\,i \triangleleft P\,i) = \left((i,s) : \sum_{i:I} S\,i\right) \triangleleft P\,i\,s$$

in $\underline{\mathbf{Cont}}$. (Note that $\prod^{\underline{\mathbf{Cont}}}$ and $\sum^{\underline{\mathbf{Cont}}}$ refer to the dependent power and copower respectively — see Section 0.4.) □

The special case where we take the product and coproduct of two containers (i.e. when I is the 2-element set) is as follows.

$$(S \triangleleft P) \times (T \triangleleft Q) = ((s,t) : S \times T) \triangleleft (P\,s + Q\,t)$$
$$(S \triangleleft P) + (T \triangleleft Q) = (x : S + T) \triangleleft \left(\lambda\, x \to \begin{cases} P\,s & \text{if } x = \mathsf{inl}\ s \\ Q\,t & \text{if } x = \mathsf{inr}\ t \end{cases}\right)$$

Similarly, n-ary containers are also closed under products and coproducts.

1.3.3 Composition

Given unary containers $S \triangleleft P$ and $T \triangleleft Q$, we can compose their extension functors to get another endofunctor on $\underline{\mathbf{Set}}$. This turns out to also be a container functor. More generally, given containers $F : |\underline{\mathbf{Cont}}_{n+1}|$ and $G : |\underline{\mathbf{Cont}}_n|$, we can compose their extension functors to get a functor $[\![F]\!]\,[\![G]\!] : \underline{\mathbf{Set}}^n \to \underline{\mathbf{Set}}$ as follows:

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

which we denote by $[\![F]\!]\,[\![G]\!]\,\mathbf{X} = [\![F]\!](\mathbf{X}, [\![G]\!]\mathbf{X})$.

Following this intuition, we compute the composition of container functors and get that this is itself a container functor, thereby obtaining a definition for composition of containers. Below is the general derivation for composing an $n+1$-ary container with an n-ary container.

We write $F = (S \triangleleft \mathbf{P}, Q)$ where $S : \mathrm{Set}$, $\mathbf{P} : \mathsf{Fin}\,n \to S \to \mathrm{Set}$, $Q : S \to \mathrm{Set}$, and $G = (A \triangleleft \mathbf{B})$ where $A : \mathrm{Set}$ and $\mathbf{B} : \mathsf{Fin}\,n \to A \to \mathrm{Set}$. In general, we have the below.

$$[\![F]\!](\mathbf{X}, Y) = \sum_{s:S}\left(\prod_{\{i:\mathsf{Fin}\,n\}} (\mathbf{P}\,i\,s \to \mathbf{X}\,i)\right) \times (Q\,s \to Y) \tag{1.3.14}$$

Theorem 1.3.15. *The composition of $n+1$-ary and n-ary container functors gives a container functor.*

Proof.

$$\begin{aligned} &[\![S \triangleleft \mathbf{P}, Q]\!]\,[\![A \triangleleft \mathbf{B}]\!]\,\mathbf{X} \\ &= \sum_{s:S}\left(\left(\prod_{\{i\}} (\mathbf{P}\,i\,s \to \mathbf{X}\,i)\right) \times (Q\,s \to [\![A \triangleleft \mathbf{B}]\!]\,\mathbf{X})\right) && [\![F]\!]\,[\![G]\!]\,\mathbf{X} = [\![F]\!](\mathbf{X}, [\![G]\!]\mathbf{X}), \text{ and Equation 1.3.14} \end{aligned}$$

$$
\begin{aligned}
&= \sum_{s:S}\Big(\Big(\prod_{\{i\}}(\mathbf{P}\,i\,s \to \mathbf{X}\,i)\Big) \times \Big(Q\,s \to \sum_{a:A}\Big(\prod_{\{i\}}(\mathbf{B}\,i\,a \to \mathbf{X}\,i)\Big)\Big)\Big) && \text{definition of } [\![\_]\!] \\
&\cong \sum_{s:S}\Big(\Big(\prod_{\{i\}}(\mathbf{P}\,i\,s \to \mathbf{X}\,i)\Big) \times \\
&\qquad \Big(\sum_{f\,:\,Q\,s\to A}\Big(\prod_{q:Q\,s}\prod_{\{i\}}(\mathbf{B}\,i\,(f\,q) \to \mathbf{X}\,i)\Big)\Big)\Big) && \text{distributivity of } \Pi \text{ over } \Sigma \\
&\cong \sum\Big(\sum_{s:S}(f : Q\,s \to A)\Big)\Big(\prod_{\{i\}}\Big(\mathbf{P}\,i\,s + \sum_{q:Q\,s}(\mathbf{B}\,i\,(f\,q))\Big) \to \mathbf{X}\,i\Big) && \text{commutativity of } \times \text{ and } (A \to C) \times (B \to C) \cong (A + B) \to C \\
&= \Big[\!\Big[\sum_{s:S}(f : Q\,s \to A) \triangleleft \Big(\lambda\{i\} \to \mathbf{P}\,i\,s + \sum_{q:Q\,s}(\mathbf{B}\,i\,(f\,q))\Big)\Big]\!\Big]\,\mathbf{X} && \text{definition of } [\![\_]\!] \\
&= [\![(S \triangleleft \mathbf{P}, Q)\;[(A \triangleleft \mathbf{B})]]\!]\mathbf{X} && \square
\end{aligned}
$$

This computation gives us a definition of composition of containers $- [-] : \underline{\mathbf{Cont}}_{n+1} \times \underline{\mathbf{Cont}}_n \to \underline{\mathbf{Cont}}_n$ as

$$(S \triangleleft \mathbf{P}, Q)\;[A \triangleleft \mathbf{B}] \coloneqq \Big(\sum_{s:S}(f : Q\,s \to A) \triangleleft \Big(\lambda\{i\} \to \mathbf{P}\,i\,s + \sum_{q:Q\,s}(\mathbf{B}\,i\,(f\,q))\Big)\Big),$$

and hence the above proves that $[\![F]\!]\,\big[[\![G]\!]\big] \cong [\![F\,[G]]\!]$.

If we consider the special case where $n = 1$ and $\mathbf{P}\,\_\,s = \top$, it follows that the composition of two unary containers $S \triangleleft Q$ and $A \triangleleft B$ is

$$(S \triangleleft Q) \circ (A \triangleleft B) = \sum_{s:S}(f : Q\,s \to A) \triangleleft \sum_{q:Q\,s} B\,(f\,q). \tag{1.3.16}$$

Example 1.3.17. We already mentioned that the container representation of the $\mathsf{List}\,T$ type is $\mathbb{N} \triangleleft \mathsf{Fin}$. We can now use the above to calculate the container representation of the $\mathsf{List}\,(\mathsf{List}\,T)$ type. Using Equation 1.3.16 with $S, A = \mathbb{N}$, and $Q, B = \mathsf{Fin}$, we get that the container is

$$\sum_{n:\mathbb{N}}(f : \mathsf{Fin}\,n \to \mathbb{N}) \triangleleft \sum_{q:\mathsf{Fin}\,n}(\mathsf{Fin}\,(f\,q)).$$

The shape of a $\mathsf{List}\,(\mathsf{List}\,T)$ is the length of the outer list n and the lengths of the inner lists given by f. For each inner list q, the positions assign as many elements for that list as dictated by f. $\Diamond$

1.3.4 Exponentiation

The category $\underline{\mathbf{Cont}}$ is closed under exponentiation. This, along with closure under products and having the terminal object ($\top \triangleleft \bot$), makes $\underline{\mathbf{Cont}}$ Cartesian closed.

Firstly, for endofunctors $F, G \colon \underline{\mathbf{Set}} \to \underline{\mathbf{Set}}$, if their exponential G^F exists then it must be of the following form. Assume $A : \mathrm{Set}$.

$$
\begin{aligned}
& G^F(A) \\
\cong & \int_{X:\mathrm{Set}} (A \to X) \to G^F(X) && \text{covariant Yoneda lemma} \\
\cong & \int_{X:\mathrm{Set}} (A \to X) \times F(X) \to G(X) && \text{exponential's universal property: } H \to G^F \cong H \times F \to G \\
\cong & \int_{X:\mathrm{Set}} (A \to X) \to F(X) \to G(X) && \text{currying in } \underline{\mathbf{Set}}
\end{aligned} \tag{1.3.18}
$$

In the case when F and G are container functors, we get that their exponential is also a container functor.

Theorem 1.3.19. *The exponentiation of unary container functors is itself a unary container functor.*

Proof. Consider the case when F above is a (unary) container functor $[\![S \triangleleft P]\!]$.

$$
\begin{aligned}
& G^{[\![S \triangleleft P]\!]}(A) \\
\cong & \int_{X:\mathrm{Set}} (A \to X) \to [\![S \triangleleft P]\!]X \to G(X) && \text{calculation above of exponential} \\
\cong & \int_X \Big(\sum_{s:S} (P\,s \to X) \Big) \to (A \to X) \to G(X) && \text{definition of } [\![\_]\!] \text{ \& commutativity of inputs} \\
\cong & \int_X \Big(\prod_{s:S} (P\,s \to X) \to (A \to X) \to G(X) \Big) && \text{currying in } \underline{\mathbf{Set}} \\
\cong & \prod_{s:S} \int_X (P\,s \to X) \times (A \to X) \to G(X) && \textstyle\prod \text{ and } \int \text{ commute} \\
\cong & \prod_{s:S} \int_X (A + P\,s \to X) \to G(X) && (X \to Z) \times (Y \to Z) \cong (X + Y \to Z) \\
\cong & \prod_{s:S} G(A + P\,s) && \text{covariant Yoneda lemma}
\end{aligned} \tag{1.3.20}
$$

If G is a container functor, then since container functors are closed under products, coproducts, and composition, $\prod_{s:S}(G(- + P\,s))$ is also a container functor. □

Spelling out the construction when $G = [\![T \triangleleft Q]\!]$, we get the below. Assume $A : \mathrm{Set}$.

$$
\begin{aligned}
& [\![T \triangleleft Q]\!]^{[\![S \triangleleft P]\!]}(A) \\
\cong & \prod_{s:S} [\![T \triangleleft Q]\!](A + P\,s) && \text{Equation 1.3.20} \\
\cong & \prod_{s:S} [\![T \triangleleft Q]\!]([\![(l : \top + P\,s) \triangleleft (l \equiv \mathsf{inl}\ \mathsf{tt})]\!]\,A) && X + P\,s \cong [\![(l : \top + P\,s) \triangleleft (l \equiv \mathsf{inl}\ \mathsf{tt})]\!]\,X
\end{aligned}
$$

$$
\begin{aligned}
&\cong \prod_{s:S} \left[\!\!\left[\left(\sum_{t:T} (f : Q\,t \to \top + P\,s)\right) \triangleleft \left(\sum_{q:Q\,t} (f\,q \equiv \mathsf{inl}\ \mathsf{tt})\right) \right]\!\!\right] A && \text{composition of container functors} \\
&\cong \left[\!\!\left[\left(f : \prod_{s:S} \sum_{t:T} (Q\,t \to \top + P\,s)\right) \right.\right. && \\
&\left.\left. \triangleleft \left(\sum_{s:S} \sum_{q:Q\,(\pi_1(f\,s))} (\pi_2(f\,s))\,q \equiv \mathsf{inl}\ \mathsf{tt}\right) \right]\!\!\right] A && \text{product of container functors}
\end{aligned}
$$

By reflection along $[\![\_]\!]$, we get that **Cont** is closed under exponentials:

$$
(T \triangleleft Q)^{(S \triangleleft P)} \cong \left(f : \prod_{s:S} \sum_{t:T} (Q\,t \to \top + P\,s)\right) \triangleleft \left(\sum_{s:S} \sum_{q:Q\,(\pi_1(f\,s))} (\pi_2(f\,s))\,q \equiv \mathsf{inl}\ \mathsf{tt}\right),
$$

thereby making **Cont** Cartesian closed.

The category $\mathbf{Cont}_n$ is also closed under exponentials.

Theorem 1.3.21. *The exponentiation of n-ary container functors is itself an n-ary container functor.*

Proof. It is easy to check that the equivalent of Equation 1.3.20 for n-ary containers, where $G : \mathbf{Set}^n \to \mathbf{Set}$ and $\mathbf{A} : \mathbf{Set}^n$, is

$$
G^{[\![S \triangleleft \mathbf{P}]\!]}(\mathbf{A}) \cong \prod_{s:S} G(\mathbf{A} +_n \mathbf{P}\,s),
$$

where $+_n$ is the coproduct in $\mathbf{Set}^n$, defined pointwise.[1]

The calculation of $[\![T \triangleleft Q]\!]^{[\![S \triangleleft \mathbf{P}]\!]}$ follows similarly as the unary calculation, except that we will need the below to rewrite a function into a coproduct.

$$
\begin{aligned}
&C \to A + B \\
\cong &C \to \sum_{x:\top+B} ((x \equiv \mathsf{inl}\ \mathsf{tt}) \to A) \\
\cong &\sum_{f\,:\,C \to \top+B} \left(\prod_{c:C} ((f\,(c) \equiv \mathsf{inl}\ \mathsf{tt}) \to A)\right) && \text{distributivity of } \Pi \text{ over } \Sigma
\end{aligned}
\tag{1.3.22}
$$

Then the exponential of n-ary container functors is calculated as follows.

$$
\begin{aligned}
&[\![T \triangleleft \mathbf{Q}]\!]^{[\![S \triangleleft \mathbf{P}]\!]}(\mathbf{A}) \\
\cong &\prod_{s:S} [\![T \triangleleft \mathbf{Q}]\!]\,(\mathbf{A} +_n \mathbf{P}\,s) \\
= &\prod_{s:S} \left(\sum_{t:T} \left(\prod_{\{i:\mathsf{Fin}\,n\}} \mathbf{Q}\,i\,t \to \mathbf{A}\,i + \mathbf{P}\,i\,s \right)\right) \\
\cong &\prod_{s:S} \left(\sum_{t:T} \left(\prod_{\{i\}} \sum_{f\,:\,\mathbf{Q}\,i\,t \to \top + \mathbf{P}\,i\,s} \left(\prod_{q:\mathbf{Q}\,i\,t} (f\,q \equiv \mathsf{inl}\ \mathsf{tt}) \to \mathbf{A}\,i \right)\right)\right) && \text{Equation 1.3.22}
\end{aligned}
$$

[1] While $\mathbf{P} : \mathsf{Fin}\,\mathrm{n} \to S \to \mathrm{Set}$, we will abuse notation and write $\mathbf{P}\,s$ instead of $\lambda i \to \mathbf{P}\,i\,s$.

$$\cong \prod_{s:S}\Big(\sum_{t:T}\Big(\sum\Big(f\colon \prod_{\{i\}} \mathbf{Q}\,i\,t \to \top + \mathbf{P}\,i\,s\Big) \Big(\prod_{\{i\}}\prod_{q:\mathbf{Q}\,i\,t}(f\,\{i\}\,q \equiv \mathsf{inl}\,\mathsf{tt}) \to \mathbf{A}\,i\Big)\Big)\Big) \qquad \text{distributivity of } \prod \text{ over } \sum$$

$$\cong \prod_{s:S}\Big[\!\Big[\sum_{t:T}\Big(f\colon \prod_{\{i\}} \mathbf{Q}\,i\,t \to \top + \mathbf{P}\,i\,s\Big) \triangleleft \lambda\{i\} \to \prod_{q:\mathbf{Q}\,i\,t} f\,\{i\}\,q \equiv \mathsf{inl}\,\mathsf{tt}\Big]\!\Big]\,\mathbf{A}$$

$$\cong \Big[\!\Big[f\colon \prod_{s:S}\sum_{t:T}\Big(\prod_{\{i\}} \mathbf{Q}\,i\,t \to \top + \mathbf{P}\,i\,s\Big) \triangleleft \lambda\{i\} \to \sum_{s:S}\sum_{q:\mathbf{Q}\,i\,(\pi_1(f\,s))} (\pi_2(f\,s))\,\{i\}\,q \equiv \mathsf{inl}\,\mathsf{tt}\Big]\!\Big]\,\mathbf{A}$$

□

By reflecting along $[\![\_]\!]$ we get the exponential for n-ary containers.

$$(T \triangleleft \mathbf{Q})^{(S \triangleleft \mathbf{P})} \cong \Big(f\colon \prod_{s:S}\sum_{t:T}\Big(\prod_{\{i:\mathsf{Fin}\,n\}} \mathbf{Q}\,i\,t \to \top + \mathbf{P}\,i\,s\Big)\Big) \triangleleft \Big(\lambda\{i\} \to \sum_{s:S}\sum_{q:\mathbf{Q}\,i\,(\pi_1(f\,s))} (\pi_2(f\,s))\,\{i\}\,q \equiv \mathsf{inl}\,\mathsf{tt}\Big) \tag{1.3.23}$$

1.4 Indexed Containers

While unary containers represent strictly positive endofunctors $\underline{\mathbf{Set}} \to \underline{\mathbf{Set}}$ and n-ary containers represent strictly positive functors $\underline{\mathbf{Set}}^n \to \underline{\mathbf{Set}}$, indexed containers represent strictly positive functors over indexed sets, $\underline{\mathbf{Fam}}_I \to \underline{\mathbf{Fam}}_J$. Indexed containers provide categorical semantics for indexed types, a.k.a. inductive families.

Definition 1.4.1. An *indexed container* with index sets I, J is given by a pair $S\colon J \to \mathrm{Set}$ and $P\colon \prod_{j:J} S\,j \to I \to \mathrm{Set}$, which we write as $S \triangleleft^\star P$. ◊

Definition 1.4.2. Given $I : \mathrm{Set}$, the category $\underline{\mathbf{Fam}}_I$ is defined as follows:[2]

- Objects are I-indexed families of sets, i.e. they have type $I \to \mathrm{Set}$.
- Morphisms $X \to^\star Y$ are I-indexed families of functions, i.e. they have type $\prod_{i:I} X\,i \to Y\,i$.
- The identity morphism $\mathrm{id}^\star\colon X \to^\star X$ is defined as $\mathrm{id}^\star\;i\,x \coloneqq x$.
- Composition of morphisms is defined for $f\colon X \to^\star Y$ and $g\colon Y \to^\star Z$ as

$$g \circ^\star f \coloneqq \lambda\,i \to (g\,i) \circ (f\,i).$$ ◊

$\underline{\mathbf{Fam}}_I$ is closed under products and coproducts, which we denote by $\times^\star$ and $+^\star$ respectively.

[2] A generalisation of this is the construction of a category $\underline{\mathbf{Fam}}_{\underline{\mathbf{C}}}$ over some category $\underline{\mathbf{C}}$, this is sometimes known as the free coproduct completion of $\underline{\mathbf{C}}$.

Definition 1.4.3. The *container functor* associated to an indexed container $S \triangleleft^\star P$, where $S : J \to \mathrm{Set}$ and $P : \prod_{j:J} S\, j \to I \to \mathrm{Set}$, is the functor $[\![S \triangleleft^\star P]\!]^\star : \underline{\mathbf{Fam}}_I \to \underline{\mathbf{Fam}}_J$, defined as follows:

- Given an $X : I \to \mathrm{Set}$ and $j : J$, $[\![S \triangleleft^\star P]\!]^\star\, X\, j \coloneqq \sum_{s:S\,j} (P\, j\, s \to^\star X)$.
- Given $X, Y : I \to \mathrm{Set}$ and a morphism $m : X \to^\star Y$,
$$[\![S \triangleleft^\star P]\!]^\star\, m : [\![S \triangleleft^\star P]\!]^\star\, X \to [\![S \triangleleft^\star P]\!]^\star\, Y$$
is defined for $j : J$, $s : S\, j$, and $f : P\, j\, s \to^\star X$ as
$$[\![S \triangleleft^\star P]\!]^\star\, m\, j\, (s, f) \coloneqq (s, m \circ^\star f). \qquad \Diamond$$

Similarly to unary containers, indexed containers indexed by sets I, J form a category $\underline{\mathbf{ICont}}_{I,J}$ and have a fully faithful functor
$$[\![\_]\!]^\star : \underline{\mathbf{ICont}}_{I,J} \to [\underline{\mathbf{Fam}}_I, \underline{\mathbf{Fam}}_J].$$

Example 1.4.4. The signature functor of the Fin type family, F_{Fin}:
$$F_{\mathsf{Fin}} : (\mathbb{N} \to \mathrm{Set}) \to \mathbb{N} \to \mathrm{Set}$$
$$F_{\mathsf{Fin}}\, X\, n \coloneqq \sum_{m:\mathbb{N}} (n \equiv \mathsf{suc}\, m) \times (\top + X\, m)$$
is expressed as an indexed container by defining

$$S : \mathbb{N} \to \mathrm{Set}$$
$$S\, n \coloneqq \sum_{m:\mathbb{N}} (n \equiv \mathsf{suc}\, m) + \sum_{m:\mathbb{N}} (n \equiv \mathsf{suc}\, m)$$

$$P : \prod_{n:\mathbb{N}} S\, n \to \mathbb{N} \to \mathrm{Set}$$
$$P\, n\, (\mathsf{inl}\, (m, \_))\, k \coloneqq \bot$$
$$P\, n\, (\mathsf{inr}\, (m, \_))\, k \coloneqq k \equiv m.$$

Then $F_{\mathsf{Fin}}\, X\, n \cong [\![S \triangleleft^\star P]\!]^\star\, X\, n$. The data type $\mathsf{Fin} : \mathbb{N} \to \mathrm{Set}$ then corresponds to the least fixed point of this functor: $\mathsf{Fin}\, \mathrm{n} \cong \mu X.[\![S \triangleleft^\star P]\!]^\star\, X\, n$. $\Diamond$

Example 1.4.5. Consider the Vec type family defined as below.

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

Its signature functor F_{Vec}, which we have seen already in Section 1.1, is expressed as an indexed container with corresponding functor $((\mathbb{N} + \top) \to \underline{\mathbf{Set}}) \to (\mathbb{N} \to \underline{\mathbf{Set}})$ as

$$S : \mathbb{N} \to \mathrm{Set}$$
$$S\, n \coloneqq (n \equiv \mathsf{zero}) + \sum_{m:\mathbb{N}} (n \equiv \mathsf{suc}\, m)$$

$$P : \prod_{n:\mathbb{N}} S\, n \to (\mathbb{N}+\top) \to \mathrm{Set}$$
$$P\, n\, (\mathsf{inl}\, \_)\, (\mathsf{inl}\, k) \coloneqq \bot$$
$$P\, n\, (\mathsf{inl}\, \_)\, (\mathsf{inr}\, \mathsf{tt}) \coloneqq \bot$$
$$P\, n\, (\mathsf{inr}\, (m, \_))\, (\mathsf{inl}\, k) \coloneqq k \equiv m$$
$$P\, n\, (\mathsf{inr}\, (m, \_))\, (\mathsf{inr}\, \mathsf{tt}) \coloneqq \top.$$

The $+\top$ in the parameter I was added to account for the extra parameter $A : \mathsf{Set}$. For the definition of P, note that given an $n : \mathbb{N}$, when we have $\mathsf{inl}\ \_ : S\ n$ then it is the case that $n \equiv \mathsf{zero}$, while when we have $\mathsf{inr}\ (m, \_) : S\ n$ it is the case that $n \equiv \mathsf{suc}\ m$. Also, when we have $\mathsf{inl}\ k : \mathbb{N} + \top$, this line is related to the recursive argument X, while when we have $\mathsf{inr}\ \mathsf{tt}$, this line is related to the type A.

So $F_{\mathsf{Vec}}\ A\ X\ n \cong [\![S \triangleleft^\star P]\!]^\star\ (X, A)\ n$, and the data type $\mathsf{Vec} : (A : \mathsf{Set}) \to \mathbb{N} \to \mathsf{Set}$ corresponds to the least fixed point of this functor: $\mathsf{Vec}\ A\ n \cong \mu X.[\![S \triangleleft^\star P]\!]^\star\ (X, A)\ n$. $\Diamond$

Indexed containers are closed under products and coproducts, which are as follows in the binary case.

$$(S \triangleleft^\star P) \times (T \triangleleft^\star Q) = ((s, t) : S \times^\star T) \triangleleft^\star (\lambda j \to P\ j\ s +^\star Q\ j\ t) \tag{1.4.6}$$

$$(S \triangleleft^\star P) + (T \triangleleft^\star Q) = (x : S +^\star T) \triangleleft^\star \left(\lambda x \to \begin{cases} x \equiv \mathsf{inl}\ s \to P\ j\ s \\ x \equiv \mathsf{inr}\ t \to Q\ j\ t \end{cases}\right) \tag{1.4.7}$$

Additionally, they are closed under composition and fixed points (initial algebras and terminal coalgebras) [AM09; AGH+15], as well as exponentiation, as shown below.

Theorem 1.4.8. *The category $\underline{\mathbf{ICont}}_{I,J}$ of I, J-indexed containers is Cartesian closed.*

To show the above theorem, we use a slight generalisation of n-ary containers, ones where $S : \mathsf{Set}$ like before, but $P : I \to S \to \mathsf{Set}$ is indexed by some type I instead of $\mathsf{Fin}\ n$ for some n. We call these I-ary containers. Their corresponding functors are of type $\underline{\mathbf{Fam}}_I \to \underline{\mathbf{Set}}$.

Proof of Theorem 1.4.8. $\underline{\mathbf{ICont}}_{I,J}$ has terminal object $(\lambda\_ \to \top) \triangleleft^\star (\lambda\_\ \_\ \_ \to \bot)$, and we know by [AM09; AGH+15] that $\underline{\mathbf{ICont}}_{I,J}$ has products. What is left to show is that it has exponentials.

The category $\underline{\mathbf{ICont}}_{I,J}$ is equivalent to the category $J \to \underline{\mathbf{Cont}}_I$ of functors from $J : \mathsf{Set}$ viewed as a discrete category to $\underline{\mathbf{Cont}}_I$, where $\underline{\mathbf{Cont}}_I$ is the category of I-ary containers. Hence we can write an indexed container $C : \left|\underline{\mathbf{ICont}}_{I,J}\right|$ as $C : \left|J \to \underline{\mathbf{Cont}}_I\right|$.

We know that $\underline{\mathbf{Cont}}_I$ has exponentials by Equation 1.3.23. If X, Y are objects of type $\left|J \to \underline{\mathbf{Cont}}_I\right|$, then we can define the exponential X^Y pointwise.

$$X^Y : J \to \underline{\mathbf{Cont}}_I$$
$$X^Y\ j \coloneqq X(j)^{Y(j)}$$

$X(j)^{Y(j)}$ is an object in $\underline{\mathbf{Cont}}_I$ and

$$\begin{aligned}(J \to \underline{\mathbf{Cont}}_I)(Z, X^Y) &\cong \prod_{j:J} \underline{\mathbf{Cont}}_I(Z(j), X(j)^{Y(j)}) \\ &\cong \prod_{j:J} \underline{\mathbf{Cont}}_I(Z(j) \times Y(j), X(j)) \\ &\cong (J \to \underline{\mathbf{Cont}}_I)(Z \times Y, X).\end{aligned}$$

□

1.5 Generalised Containers

We introduce a new kind of container that we call generalised containers, that represent strictly positive functors $\underline{\mathbf{C}} \to \underline{\mathbf{Set}}$ for any category $\underline{\mathbf{C}}$. They have also been referred to in the literature as familially representable functors [CJ95].

I have formalised some of the statements in this section in `Cubical Agda`. This code was merged into the `Cubical Agda` library, `agda/cubical`. Formalisations of the statements in this section annotated with a ✿ can be found in the official `agda/cubical` documentation (pull request link: `github.com/agda/cubical/pull/1129`).

Definition 1.5.1 (✿). A *generalised container* over some category $\underline{\mathbf{C}}$ is given by a pair $S : \mathrm{Set}$ and $P \colon S \to |\underline{\mathbf{C}}|$, which we write as $S \mathrel{\underline{\triangleleft}} P$. ◊

Definition 1.5.2 (✿). The *container functor* associated to a generalised container $S \mathrel{\underline{\triangleleft}} P$ is the functor $[\![S \mathrel{\underline{\triangleleft}} P]\!] \colon \underline{\mathbf{C}} \to \underline{\mathbf{Set}}$ with the following actions on objects and morphisms.

- Given an $X : |\underline{\mathbf{C}}|$, $[\![S \mathrel{\underline{\triangleleft}} P]\!] X \coloneqq \sum_{s:S} \underline{\mathbf{C}}(P\,s, X)$.
- Given $X, Y : |\underline{\mathbf{C}}|$ and $f \colon \underline{\mathbf{C}}(X, Y)$,

$$[\![S \mathrel{\underline{\triangleleft}} P]\!] f \colon [\![S \mathrel{\underline{\triangleleft}} P]\!] X \to [\![S \mathrel{\underline{\triangleleft}} P]\!] Y$$
$$[\![S \mathrel{\underline{\triangleleft}} P]\!] f\,(s, g) \coloneqq (s, f \mathbin{\underline{\circ}} g)$$

  where $s : S$, $g : \underline{\mathbf{C}}(P\,s, X)$, and $\underline{\circ}$ denotes composition in $\underline{\mathbf{C}}$. ◊

Similarly to the case of unary containers, we can define the category of generalised containers $\underline{\mathbf{GCont}}$ (✿), as well as the generalised container extension functor (✿)

$$[\![-]\!] \colon \underline{\mathbf{GCont}} \to [\underline{\mathbf{C}}, \underline{\mathbf{Set}}].$$

Theorem 1.5.3 (✿). *The functor $[\![-]\!] \colon \underline{\boldsymbol{GCont}} \to [\underline{\boldsymbol{C}}, \underline{\boldsymbol{Set}}]$ is full and faithful.*

Proof. The proof follows almost identically to the corresponding proof on unary containers (Theorem 1.3.12). □

Theorem 1.5.4. *Generalised containers are closed under coproducts.*

Proof. Fix a category $\underline{\mathbf{C}}$. We show that coproducts of objects in the image of $[\![-]\!]$ are themselves in the image of $[\![-]\!]$. We can then reflect them along the fully-faithful functor $[\![-]\!]$ to get coproducts in $\underline{\mathbf{GCont}}$.

We calculate the I-ary coproduct of generalised container functors for some $I : \mathrm{Set}$.

$$
\begin{aligned}
&\sum_{i:I} \left([\![S\, i \mathrel{\underline{\triangleleft}}_{\underline{\mathbf{C}}} P\, i]\!]_{\underline{\mathbf{C}}}\, X \right) \\
&= \sum_{i:I} \left(\sum_{s:S\, i} \underline{\mathbf{C}}(P\, i\, s, X) \right) && \text{definition of } [\![\_]\!]_{\underline{\mathbf{C}}} \\
&\cong \sum \left((i, s) : \sum_{i:I} S\, i \right) (\underline{\mathbf{C}}(P\, i\, s, X)) && \textstyle\sum (a : A)(\sum (b : B\, a)(C\, a\, b)) \cong \sum (\sum A\, B)\, C \\
&= \left[\!\!\left[\left((i, s) : \sum_{i:I} S\, i \right) \mathrel{\underline{\triangleleft}}_{\underline{\mathbf{C}}} P\, i\, s \right]\!\!\right]_{\underline{\mathbf{C}}} X && \text{definition of } [\![\_]\!]_{\underline{\mathbf{C}}}
\end{aligned}
$$

□

Theorem 1.5.5. *If the category $\underline{\mathbf{C}}$ has (infinite) coproducts, then generalised containers over $\underline{\mathbf{C}}$ are closed under (infinite) products.*

Proof. Assuming $\underline{\mathbf{C}}$ has coproducts, we show that products of objects in the image of $[\![\_]\!]_{\underline{\mathbf{C}}}$ are themselves in the image of $[\![\_]\!]_{\underline{\mathbf{C}}}$. We can then reflect them along the fully-faithful functor $[\![\_]\!]_{\underline{\mathbf{C}}}$ to get products in $\underline{\mathbf{GCont}}$.

We calculate the I-ary product of generalised container functors for some $I : \mathrm{Set}$.

$$
\begin{aligned}
&\prod_{i:I} \left([\![S\, i \mathrel{\underline{\triangleleft}}_{\underline{\mathbf{C}}} P\, i]\!]_{\underline{\mathbf{C}}}\, X \right) \\
&= \prod_{i:I} \left(\sum_{s:S\, i} \underline{\mathbf{C}}(P\, i\, s, X) \right) && \text{definition of } [\![\_]\!]_{\underline{\mathbf{C}}} \\
&\cong \sum \left(f : \prod_{i:I} S\, i \right) \left(\prod_{i:I} \underline{\mathbf{C}}(P\, i\, (f\, i), X) \right) && \text{distributivity of } \Pi \text{ over } \Sigma \\
&\cong \sum \left(f : \prod_{i:I} S\, i \right) \left(\underline{\mathbf{C}} \left(\sum_{i:I} P\, i\, (f\, i), X \right) \right) && \text{universal property of coproducts in } \underline{\mathbf{C}} \\
&= \left[\!\!\left[\left(f : \prod_{i:I} S\, i \right) \mathrel{\underline{\triangleleft}}_{\underline{\mathbf{C}}} \sum_{i:I} (P\, i\, (f\, i)) \right]\!\!\right]_{\underline{\mathbf{C}}} X && \text{definition of } [\![\_]\!]_{\underline{\mathbf{C}}}
\end{aligned}
$$

□

Theorem 1.5.6. *Indexed containers are a special case of generalised containers.*

Proof. We start by noting that $(I \to \mathrm{Set}) \to (J \to \mathrm{Set}) \cong (J \times (I \to \mathrm{Set})) \to \mathrm{Set}$, where $I \to \mathrm{Set}$ is the type of objects of $\underline{\mathbf{Fam}}_I$. Then we define the category $\underline{\mathbf{D}}$, where

- objects are pairs $(j, A) : J \times (I \to \mathrm{Set})$
- morphisms exist only between pairs whose first elements agree: $(j, A) \to_{\underline{\mathbf{D}}} (k, B)$ requires a proof that $j \equiv k$ and a map of type $\prod_{i:I} A\, i \to B\, i$.

There is an equivalence of categories between $\underline{\mathbf{Fam}}_I$ and $\underline{\mathbf{D}}$. Given an indexed container $S \mathrel{\triangleleft^\star} P$, where

$$S \colon J \to \mathrm{Set}$$

$$P\colon \prod_{j:J} S\, j \to I \to \mathrm{Set}$$

we define a generalised container over $\underline{\mathbf{D}}$ as

$$\overline{S} \coloneqq \sum_{j:J} S\, j$$
$$\overline{P}\,(j,s) \coloneqq (j, P\, j\, s).$$

Now we show that $[\![S \triangleleft^\star P]\!]^\star$ and $[\![\overline{S} \triangleleft_{\underline{\mathbf{D}}} \overline{P}]\!]_{\underline{\mathbf{D}}}$ agree on objects and on morphisms, up to the equivalence of categories.

Firstly, we show that the functors agree on objects: for $A\colon I \to \mathrm{Set}$ and $j : J$, we have the following.

$$\begin{aligned}
&[\![\overline{S} \triangleleft_{\underline{\mathbf{D}}} \overline{P}]\!]_{\underline{\mathbf{D}}}\,(j, A) \\
&= \sum_{k:J} \sum_{s:S\,k} \underline{\mathbf{D}}(\overline{P}\,(k,s), (j,A)) \\
&= \sum_{k:J} \sum_{s:S\,k} \underline{\mathbf{D}}((k, P\,k\,s), (j,A)) \\
&= \sum_{k:J} \sum_{s:S\,k} (k \equiv j) \times ((i : I) \to P\,k\,s\,i \to A\,i) \\
&\cong \sum_{s:S\,j} ((i : I) \to P\,j\,s\,i \to A\,i) && \textstyle\sum_{x:X}(x \equiv x') \text{ is contractible} \\
&= [\![S \triangleleft^\star P]\!]^\star\, A\, j
\end{aligned}$$

Secondly, we show they agree on morphisms: for $m\colon (i : I) \to X\,i \to Y\,i$, $j : J$, $s : S\,j$, and $f\colon (i : I) \to P\,j\,s\,i \to X\,i$, we have

$$\begin{aligned}
&[\![S \triangleleft^\star P]\!]^\star\, m\, j\, (s, f) \\
&= (s, m \circ^\star f) \\
&= (s, \lambda\, i\, p.m\, i\, (f\, i\, p)).
\end{aligned}$$

Now m can be turned into a morphism $\overline{m}\colon (k, X) \to_{\underline{\mathbf{D}}} (k, Y)$ for any $k : J$. If we fix k to be j as above, we have

$$\begin{aligned}
&[\![\overline{S} \triangleleft_{\underline{\mathbf{D}}} \overline{P}]\!]_{\underline{\mathbf{D}}}\, \overline{m}\, ((j,s), f) \\
&= ((j,s), m \circ_{\underline{\mathbf{D}}} f) \\
&= ((j,s), \lambda\, i\, p.m\, i\, (f\, i\, p)).
\end{aligned}$$

Therefore the two functors agree on morphisms up to the equivalence of categories between $\underline{\mathbf{Fam}}_I$ and $\underline{\mathbf{D}}$. □

1.6 Related Work

Besides the unary, n-ary, indexed, and generalised containers discussed here, other kinds of containers have been proposed in the literature. Gylterud introduced *symmetric containers*, whose shapes $S : \mathrm{Gpd}$ form a groupoid, and whose positions $P\colon S \to \mathrm{Set}$ form a family of Sets; as well as *categorical containers*, whose shapes $S : \mathrm{Cat}$ form a category, and whose positions $P\colon S \to \underline{\mathbf{Cat}}$ are represented by a functor from S to the category of small categories [Gyl11]. Altenkirch and Tian are currently working on *higher-kinded containers*, which model signature functors of type $(\underline{\mathbf{Set}} \to \underline{\mathbf{Set}}) \to (\underline{\mathbf{Set}} \to \underline{\mathbf{Set}})$, like for example for the coinductive type Bush [Alt25].

Chapter 2

Inductive & Coinductive Containers

Chapter 1 gave an introduction to containers and their closure properties. One closure property that we mentioned briefly but did not elaborate on is that 'containers are closed under initial algebras and terminal coalgebras', or in other words, 'containers are closed under least and greatest fixed points', a property first shown by Abbott et al. [AAG05]. Since many of the widely used inductive definitions such as $\mathbb{N}$ and List are fixed points, this property is essential if we are to consider containers as a semantic representation of strictly positive types.

In this chapter, we present a formalisation of the result that 'container functors are closed under initial algebras and terminal coalgebras' in `Cubical Agda`, which moreover makes no h-set assumptions. The original development of containers [AAG03; AAG04; AAG05; Abb03] uses a categorical language, where containers are presented as constructions in the internal language of LCCCs with disjoint coproducts and W-types, so-called 'Martin-Löf categories', and a standard set-theoretic metatheory is used. Abbott et al. claim in [AAG05, Section 2.1] that these developments can also be interpreted as constructions in extensional MLTT. While the work of Seely, Hofmann, and others shows that this translation can be done in principle [See84; Hof94], the construction in type theory is not actually carried out. We remedy this by providing a formalisation of existing results by Abbott et al. [AAG05]. While the aforementioned paper implicitly assumes UIP throughout, we do not assume this principle for any of the types involved in our formalisation. More precisely, the main contributions of this chapter are:

- the formalisation of the results stating 'container functors preserve initial algebras' and 'container functors preserve terminal coalgebras' (Theorems 2.3.1 and 2.3.4), and
- the generalisation of these results by proving them not in the category of Sets, but in the wild category of types.

We regard the formalisation of these results within a subsystem of HoTT as a first step towards developing container theory in HoTT, with the long-term goal of providing container semantics for higher inductive types. We discuss more on this in Sections 3.1 and 3.5.

The formalisation is written in `Cubical Agda` [VMA21], an extension of the `Agda` proof assistant which implements a cubical [CCHM18] flavour of HoTT. Although a lot of the interest around `Cubical Agda` is arguably due to its native support for the univalence axiom, our main motivation for using it is due to its treatment of equality as a path type, which restores the symmetry between inductive and coinductive reasoning in `Agda`. In intensional type theory, and therefore in vanilla `Agda`, working with inductive types is facilitated by using pattern matching and structural recursion. For coinductive types, while we do have copattern matching and some guarded corecursion, we cannot get very far when working in this setting. In particular, many equalities on coinductive types are impossible to prove in much the same way that equalities on function types cannot be proved: this requires some form of extensionality. Fortunately, `Cubical Agda` makes function extensionality provable, and many of the equalities on coinductive types that were previously impossible in vanilla `Agda` also become provable. With our formalisation, we contribute to the `agda/cubical` library [Agd25a], and hope to demonstrate that the practical developments brought to MLTT by cubical type theory extend beyond just computational univalence.

Our formalisation is available at the link in [DAL25a] in a fork of the `agda/cubical` library. We have type-checked it using versions 2.6.4.3 and 2.6.5 of `Agda`. We note that the numbering of definitions, theorems, and lemmas in the cited `Cubical Agda` file corresponds to the numbering in the published paper this chapter is based on [DAL25b].

2.1 Background

In this section, we present some background material that aids in understanding the rest of the chapter. We start by giving a brief introduction to `Cubical Agda` concepts that will be used throughout. We then review M-types, as well as the theory of containers adapted to wild categories.

The setting of this chapter is cubical type theory, although as we discuss in Section 2.3.1, we expect our proof to also hold in any type theory with function extensionality, W- and M-types, and M's coinduction principle MCoind. We note that in this chapter, since we want to generalise from h-sets to general types, we replace our use of Set (the universe of h-sets) with Type, which we use to refer to both the wild category of types as well as `Cubical Agda`'s universe of types.

2.1.1 Cubical Agda

We already introduced the proof assistant `Agda` in Section 0.2.4. This chapter is formalised in `Cubical Agda` [VMA21], which extends `Agda` with primitives from cubical type theory [CCHM18; CHM18]. While the original motivation behind this extension was arguably to allow for native support for Voevodsky's univalence ax-

iom [Voe14] and higher inductive types [LS19], we are primarily interested in `Cubical Agda`'s representation of equality. The equality type in `Cubical Agda`, also called the *path type*, restores the equivalence between bisimilarity and equality for coinductive types. In order to explain how, let us briefly introduce some of the elementary machinery of `Cubical Agda`. We note that we leave out certain important primitive operations such as hcomp if they do not appear in the rest of the chapter; the interested reader should consult [CCHM18; CHM18; VMA21] for a more complete picture of cubical type theory and its implementation in `Cubical Agda`.

The main novelty of `Cubical Agda` is the addition of an interval (pre-)type I. This type has two terms i0, i1 : I denoting the endpoints of the interval. It comes equipped with meet and join operations _∧_, _∨_ : I → I → I and a 'reversal' operation ~ : I → I, turning I into a De Morgan algebra. This allows us to internalise the usual homotopical notion of a path and take that as our definition of equality. Indeed, an equality p between two points $x, y : A$, denoted $p : x \equiv y$, is a path in A between x and y, i.e. a function $p : \mathsf{I} \to A$ such that $p\ \mathsf{i0}$ is definitionally equal to x and $p\ \mathsf{i1}$ is definitionally equal to y. To showcase some elementary constructions of paths in `Cubical Agda`, consider e.g. the proofs of reflexivity and symmetry shown below. Introducing a path variable into scope, such as the variable $i : \mathsf{I}$ in both definitions below, is known as interval abstraction or path abstraction.

```
refl : {x : A} → x ≡ x
refl {x = x} i = x
```

```
_⁻¹ : x ≡ y → y ≡ x
(p ⁻¹) i = p (~ i)
```

As in plain MLTT, there is also a notion of dependent path in `Cubical Agda` expressed by the primitive type called PathP. This type generalises the path type by considering dependent functions $(i : \mathsf{I}) \to A\ i$ instead of only non-dependent ones $\mathsf{I} \to A$. We adopt an informal approach and write $a \cong b$ for the statement that '$a : A$ is equal to $b : B$ modulo some path of types $p : A \equiv B$'. The definition of the path of types will often be omitted from the main text, as it almost always can be automatically inferred from context, but it will be included in a separate yellow text box for the interested reader.

On a related note, we also mention here that the `Cubical Agda` primitives allow us to define transport : $A \equiv B \to A \to B$. As usual we have that, for any $p : A \equiv B$, the type of dependent paths $a \cong b$ modulo p is equivalent to the type transport $p\ a \equiv b$.

One of the key advantages of `Cubical Agda`'s treatment of equality is that it renders function extensionality a triviality:

```
funExt : ((x : A) → f x ≡ g x) → f ≡ g
funExt p i x = p x i
```

A consequence of this is that we can use the types $f \equiv g$ and $(x : A) \to f\ x \equiv g\ x$ interchangeably without having to worry about introducing any bureaucracy when moving from one to the other; in `Cubical Agda`, equality of functions *is precisely* pointwise equality. In a similar way, and especially important for us, the equality type of a coinductive type *is* bisimulation, modulo copattern matching. For example, we can define id below by first introducing the path variable i, after which we are

required to construct an element of Stream A (see definition in Section 2.1.2) matching the endpoints xs at $i = 0$ and ys at $i = 1$, which we do using copattern matching. In particular, we can show that id is an equivalence, and if we assume univalence, which we do not require here, we can even show that $(xs \equiv ys) \equiv (xs \approx ys)$. To the best of our knowledge, this feature is unique to `Cubical Agda`.

```
record _≈_ (xs ys : Stream A) : Set where
  coinductive
  field
    hd≡ : hd xs ≡ hd ys
    tl≈ : tl xs ≈ tl ys

id : (xs ys : Stream A) → xs ≈ ys →
     xs ≡ ys
hd (id xs ys p i) = hd≡ p i
tl (id xs ys p i) = id (tl xs) (tl ys) (tl≈ p) i
```

2.1.2 Coinductive types and the M-type

We have already looked at inductive types in Section 0.3, and what it means for them to be strictly positive in Section 1.2. Dual to inductive types is the notion of a coinductive type, i.e. a type defined by a list of destructors. While inductive types are described by the different ways we can construct them, coinductive types are described by the ways we can break them apart. Coinductive types are typically infinite structures, for example the type of Streams, or infinite lists, shown below. This type is parameterised by a type of inputs A, and if we have a Stream, we can ask for its head (hd) and its tail (tl).

```
record Stream (A : Type) : Type where
  coinductive
  field
    hd : A
    tl : Stream A
```

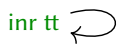
As is the case for inductive definitions, coinductive definitions also ought to be strictly positive in order to avoid inconsistencies (to be precise, in both cases we mean that the type's signature functor should be strictly positive). The type M, first studied by Abbott et al. [AAG05] and van den Berg and De Marchi [vD07] and dual to the W-type, is the type of non-well-founded, labelled trees. A tree of type M can have both finite and infinite paths. In our development, M takes two parameters S : Type and $P : S \to$ Type (note that we use Type instead of Set), and we think of them similarly as for W, i.e. S is the type of shapes of the tree, and for a given shape $s : S$, the tree has $(P\ s)$-many positions. Dually to W, M is the universal type for strictly positive coinductive types.

```
record M (S : Type) (P : S → Type) : Type where
  coinductive
  field
    shape : S
    pos : P shape → M S P
```

Example 2.1.1. If we define S and P as in Example 1.2.1, then M S P is an encoding of the conatural numbers ℕ∞.

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

Apart from having all the natural numbers as its elements, ℕ∞ also has an 'infinite number' whose predecessor is itself. This number is represented by the infinite tree shown below. This M-tree clearly has an infinite path.



◊

We will see in Section 2.3 that it is useful to have an explicit account of the coinduction principle of M, which states that any two $m_0, m_1 : \mathsf{M}\,S\,Q$ that can be related by a bisimulation are equal. In `Cubical Agda`, we can define what it means for a relation R on M to be a bisimulation using M-R below — R has to relate two elements m_0 and m_1 of M whenever their shapes are equal and their positions are related by R.

```
record M-R (R : M S Q → M S Q → Type) (m₀ m₁ : M S Q) : Type where
  field
    s-eq : shape m₀ ≡ shape m₁
    p-eq : (q₀ : Q (shape m₀)) (q₁ : Q (shape m₁))
           (q-eq : q₀ ≅ q₁) → R (pos m₀ q₀) (pos m₁ q₁)
```

↑ Above, *q-eq* is a dependent path over the path of types (λ *i* → *Q* (s-eq *i*))

We can then prove the coinduction principle using interval abstraction and copattern matching.

```
MCoind : (R : M S Q → M S Q → Type)
         (is-bisim : {m₀ m₁ : M S Q} → R m₀ m₁ → M-R R m₀ m₁)
         {m₀ m₁ : M S Q} → R m₀ m₁ → m₀ ≡ m₁
shape (MCoind R is-bisim r i) = s-eq (is-bisim r) i
pos (MCoind R is-bisim {m₀ = m₀}{m₁ = m₁} r i) q =
    MCoind R is-bisim {m₀ = pos m₀ q₀} {m₁ = pos m₁ q₁}
           (p-eq (is-bisim r) q₀ q₁ q₂) i
```

Above, q_0, q_1, and q_2 are all of the form transport ... q. The constructions of these transports use some rather technical cube algebra, and rely on the fact that if we have an element q : QQ *i* living over a path variable *i*, then we can obtain a q_0 : QQ i0 and a q_1 : QQ i1 by transporting, that are equal up to a path. We omit the details and refer the interested reader to the formalisation.

2.1.3 Wild containers

The container and container functor definitions in this section are adapted from those in Chapter 1 to the wild category of types Type. A wild category is a category as defined

in Section 0.4, except the type of morphisms need not be an h-set (i.e. it need not satisfy UIP); this is also called a precoherent higher category in [Che25]. We observe that Type has triangle and pentagon coherators, and is therefore a 2-coherent category in the sense of [Che25], and moreover its coherators are trivial [Che25, Example 2.2.5]. The definitions of container functor algebras are standard category theory definitions also adapted to this setting, with the initial algebra definition being motivated by [KvR21, Definition 5].

Definition 2.1.2. A *(unary) container* is given by a pair of types $S : \mathsf{Type}$ and $P : S \to \mathsf{Type}$, which we write as $S \triangleleft P$. ♢

Definition 2.1.3. For some (potentially infinite) indexing type I, an *I-ary container* is given by a pair $S : \mathsf{Type}$ and $\mathbf{P} : I \to S \to \mathsf{Type}$, which we write as $S \triangleleft \mathbf{P}$. ♢

As was the case in the previous chapter, we will sometimes write $P_0, P_1, \ldots$ instead of $\mathbf{P}\ \imath_0, \mathbf{P}\ \imath_1, \ldots$ to enumerate the families over S.

Unary containers are trivially I-ary containers when $I = \top$, so henceforth in this section we will only consider I-ary containers.

To every container $S \triangleleft \mathbf{P}$, we associate a wild functor which maps a family of types $\mathbf{X} : I \to \mathsf{Type}$ to a choice of shape $s : S$, and for every $i : I$ and position $\mathbf{P}\ i\ s$ associated to s, a value of type $\mathbf{X}\ i$ to be stored at that position.

Definition 2.1.4. The *container functor* associated to an I-ary container $S \triangleleft \mathbf{P}$ is the wild functor $[\![S \triangleleft \mathbf{P}]\!] : \mathsf{Type}^I \to \mathsf{Type}$, where Type^I refers to the wild category of I-indexed families of types. It has the following actions on objects and morphisms.

- Given $\mathbf{X} : I \to \mathsf{Type}$, we define $[\![S \triangleleft \mathbf{P}]\!]\,\mathbf{X} :\equiv \sum_{s:S} \left(\prod_{\imath:I} \mathbf{P}\,\imath\,s \to \mathbf{X}\,\imath \right)$.
- Given $\mathbf{X}, \mathbf{Y} : I \to \mathsf{Type}$, and a morphism $f : \prod_{\imath:I} \mathbf{X}\,\imath \to \mathbf{Y}\,\imath$, we define

$$[\![S \triangleleft \mathbf{P}]\!]\,f\,(s, g) :\equiv (s, f \circ g)$$

  for $s : S$ and $g : \prod_{\imath:I} \mathbf{P}\,\imath\,s \to \mathbf{X}\,\imath$, where $f \circ g$ is composition in Type^I. ♢

As a special case of Definition 2.1.4, given an $(I+1)$-ary container $F = S \triangleleft \mathbf{R}$, we will later need to write it in a way where we single out one component from it. We split $\mathbf{R}$ into $\mathbf{P}$ and Q and write F as having components $S : \mathsf{Type}, \mathbf{P} : I \to S \to \mathsf{Type}, Q : S \to \mathsf{Type}$, and use the notation $F = (S \triangleleft \mathbf{P}, Q)$. Given $\mathbf{X} : I \to \mathsf{Type}$ and $Y : \mathsf{Type}$, then $S, \mathbf{P}$, and Q satisfy the below.

$$[\![S \triangleleft \mathbf{P}, Q]\!](\mathbf{X}, Y) = \sum_{s:S} \left(\prod_{\imath:I} \mathbf{P}\,\imath\,s \to \mathbf{X}\,\imath \right) \times (Q\,s \to Y) \tag{2.1.5}$$

The two main results formalised in this chapter concern the initial algebra and terminal coalgebra of a container functor. We define explicitly what we mean by these in the

setting of wild categories and wild functors. We note that for (at least a naive definition of) a functor of wild categories $F : \underline{\mathbf{C}} \to \underline{\mathbf{C}}$, it is not in general the case that F-algebras form a wild category. However, this is the case for container functors. This follows from the properties of Type we mentioned at the start of the section.

Definition 2.1.6. For a (unary) container functor $[\![F]\!] : \mathsf{Type} \to \mathsf{Type}$, the *wild category of* $[\![F]\!]$*-algebras*, denoted $\mathsf{Alg}_{[\![F]\!]}$, is defined as follows.

- Objects are *algebras*: pairs $(X : \mathsf{Type}, \alpha : [\![F]\!]\, X \to X)$.[1]
- A morphism of algebras $(X, \alpha) \to (Y, \beta)$ is a function $f : X \to Y$ such that the following square commutes.

$$\begin{array}{ccc} [\![F]\!]\, X & \xrightarrow{[\![F]\!]\, f} & [\![F]\!]\, Y \\ {\scriptstyle\alpha}\downarrow & & \downarrow{\scriptstyle\beta} \\ X & \xrightarrow[f]{} & Y \end{array}$$

◊

Definition 2.1.7. The *initial algebra* of a (unary) container functor $[\![F]\!] : \mathsf{Type} \to \mathsf{Type}$ is an algebra (I, ι) such that for every algebra (X, α), $\mathsf{Alg}_{[\![F]\!]}((I, \iota), (X, \alpha))$ is contractible. ◊

The *wild category of* $[\![F]\!]$*-coalgebras* $\mathsf{Coalg}_{[\![F]\!]}$ is dual to Definition 2.1.6, where the objects, *coalgebras*, are defined as pairs $(X : \mathsf{Type}, \alpha : X \to [\![F]\!]\, X)$. The terminal coalgebra is then an object (T, τ) such that for every coalgebra (X, α), $\mathsf{Coalg}_{[\![F]\!]}((X, \alpha), (T, \tau))$ is contractible.

2.2 Setting Up

In this section, we state precisely what it is that we want to prove and start attacking the problem. We construct a candidate initial algebra and terminal coalgebra for a general container functor, which in the following section we prove to be correct. We also discuss a generalised induction principle for the inductive family Pos of finite paths in a tree.

2.2.1 Calculation of the initial algebra and terminal coalgebra

Given a container functor $[\![F]\!] : \mathsf{Type}^{I+1} \to \mathsf{Type}$, which we write as $F = (S \triangleleft \mathbf{P}, Q)$, we need to specify container functors

$$[\![A_\mu \triangleleft \mathbf{B}_\mu]\!] : \mathsf{Type}^I \to \mathsf{Type}$$
$$[\![A_\nu \triangleleft \mathbf{B}_\nu]\!] : \mathsf{Type}^I \to \mathsf{Type}$$

such that

$$[\![A_\mu \triangleleft \mathbf{B}_\mu]\!]\, \mathbf{X} \cong \mu Y.[\![F]\!](\mathbf{X}, Y),$$

[1] X is sometimes called the carrier of the algebra.

$$[\![A_\nu \triangleleft \mathbf{B}_\nu]\!]\,\mathbf{X} \cong \nu Y.[\![F]\!](\mathbf{X}, Y).$$

The symbols μ and ν above denote partial operators taking a wild functor to the carrier of its initial algebra or terminal coalgebra respectively, if they exist. More precisely, the notation $\mu Y.[\![F]\!](\mathbf{X}, Y)$ is shorthand for (the carrier of) the initial algebra of the wild functor G defined by $G\,Y \coloneqq [\![F]\!](\mathbf{X}, Y)$, and dually for ν. We also note that $\cong$ denotes an equivalence of types, which there are several different notions of in HoTT. In our formalisation, we primarily use a definition in terms of quasi-inverses [Uni13, Definition 2.4.6], i.e. a function with an explicit inverse. All statements in this paper are independent of this particular choice and can be read with any other reasonable notion of equivalence in mind.

We now illustrate how we calculate containers $(A_\mu \triangleleft \mathbf{B}_\mu)$ and $(A_\nu \triangleleft \mathbf{B}_\nu)$ in I parameters to make the above equivalences hold. Calculating A_μ and A_ν is straightforward. If we set $\mathbf{X} = \top$ in the above, we get

$$A_\mu \cong [\![A_\mu \triangleleft \mathbf{B}_\mu]\!]\,\top \cong \mu Y.[\![F]\!](\top, Y) \cong \mu Y.\sum_{s:S}(Q\,s \to Y) \cong \mu Y.[\![S \triangleleft Q]\!]\,Y \cong \mathsf{W}\,S\,Q$$
$$A_\nu \cong [\![A_\nu \triangleleft \mathbf{B}_\nu]\!]\,\top \cong \nu Y.[\![F]\!](\top, Y) \cong \nu Y.\sum_{s:S}(Q\,s \to Y) \cong \nu Y.[\![S \triangleleft Q]\!]\,Y \cong \mathsf{M}\,S\,Q.$$

The last step follows from the fact that the least (resp. greatest) fixed point of the container functor in one variable $[\![S \triangleleft Q]\!]$ is $\mathsf{W}\,S\,Q$ ($\mathsf{M}\,S\,Q$), the W-type (M-type) with shapes S and positions Q.

Calculating $\mathbf{B}_\mu : I \to \mathsf{W}\,S\,Q \to \mathsf{Type}$ and $\mathbf{B}_\nu : I \to \mathsf{M}\,S\,Q \to \mathsf{Type}$ is a bit more involved. Our reasoning below applies to both $\mathsf{W}\,S\,Q$ and $\mathsf{M}\,S\,Q$, so we consider any fixed point ϕ of the container functor $[\![S \triangleleft Q]\!]$ and construct $\mathbf{B} : I \to \phi \to \mathsf{Type}$. Being a fixed point of $[\![S \triangleleft Q]\!]$ means that ϕ consists of a carrier $\mathsf{C} : \mathsf{Type}$ together with an equivalence, $\chi : [\![S \triangleleft Q]\!]\,\mathsf{C} \cong \mathsf{C}$.

```
record FixedPoint : Type₁ where
  field
    C : Type
    χ : (Σ[ s ∈ S ] (Q s → C)) ≅ C
```

In particular, we have $\mathsf{WAlg} : \mathsf{FixedPoint}$ whose carrier is $\mathsf{W}\,S\,Q$ and $\mathsf{MAlg} : \mathsf{FixedPoint}$ whose carrier is $\mathsf{M}\,S\,Q$ (for the χ components, we refer the reader to the formalisation).

If $[\![\mathsf{C} \triangleleft \mathbf{B}]\!]\mathbf{X}$ is to be a fixed point of $[\![F]\!](\mathbf{X}, -)$, then we need the following isomorphism to hold.

$$[\![F]\!](\mathbf{X}, [\![\mathsf{C} \triangleleft \mathbf{B}]\!]\,\mathbf{X}) \cong [\![\mathsf{C} \triangleleft \mathbf{B}]\!]\,\mathbf{X} \tag{2.2.1}$$

By massaging the left hand side of this isomorphism, we can write it as a container functor in terms of only $\mathbf{X}$.

$$\sum_{s:S}\left(\left(\prod_{\imath}\mathbf{P}\,\imath\,s \to \mathbf{X}\,\imath\right) \times (Q\,s \to [\![\mathsf{C} \triangleleft \mathbf{B}]\!]\,\mathbf{X})\right) \qquad \text{using Equation 2.1.5}$$
$$= \sum_{s:S}\left(\left(\prod_{\imath}\mathbf{P}\,\imath\,s \to \mathbf{X}\,\imath\right) \times \left(Q\,s \to \sum_{c:\mathsf{C}}\left(\prod_{\imath}\mathbf{B}\,\imath\,c \to \mathbf{X}\,\imath\right)\right)\right) \qquad \text{definition of } [\![\_]\!]$$

$$\cong \sum_{s:S}\Big(\Big(\prod_{\imath} \mathbf{P}\ \imath\ s \to \mathbf{X}\ \imath\Big) \times \sum_{f\,:\,Q\ s \to C} \Big(\prod_{q:Q\ s}\prod_{\imath} \mathbf{B}\ \imath\ (f\ q) \to \mathbf{X}\ \imath\Big)\Big) \qquad \text{distributivity of } \Pi \text{ over } \Sigma$$

$$\cong \sum_{(s,f)\,:\,\sum_{s:S}(Q\ s \to C)} \Big(\prod_{\imath}\Big(\mathbf{P}\ \imath\ s + \sum_{q:Q\ s} \mathbf{B}\ \imath\ (f\ q)\Big) \to \mathbf{X}\ \imath\Big) \qquad \text{commutativity of } \times \text{ and } (A+B) \to C \cong (A \to C) \times (B \to C)$$

$$= \Big[\!\Big[(s,f) : \sum_{s:S}(Q\ s \to C) \triangleleft \Big(\lambda\ \imath.\ \mathbf{P}\ \imath\ s + \sum_{q:Q\ s} \mathbf{B}\ \imath\ (f\ q)\Big)\Big]\!\Big]\ \mathbf{X} \qquad \text{definition of } [\![\_]\!]$$

The induced isomorphism (2.2.1) can then be written as

$$\Big[\!\Big[(s,f) : \sum_{s:S}(Q\ s \to C) \triangleleft \Big(\lambda\ \imath.\ \mathbf{P}\ \imath\ s + \sum_{q:Q\ s} \mathbf{B}\ \imath\ (f\ q)\Big)\Big]\!\Big]\ \mathbf{X} \cong [\![C \triangleleft \mathbf{B}]\!]\ \mathbf{X}.$$

We already have the isomorphism $\chi : \sum_{s:S}(Q\ s \to C) \cong C$ on shapes. We will also need the below isomorphism on positions for $\imath : I$ and $c : C$. We call ξ the map in the inverse direction of χ and use the notation $(\phi\ \xi_0)$ and $(\phi\ \xi_1)$ for its first and second projections, so that $(\phi\ \xi_0)\ c : S$ and $(\phi\ \xi_1)\ c : Q\ ((\phi\ \xi_0)\ c) \to C$.

$$\Big(\mathbf{P}\ \imath\ ((\phi\ \xi_0)\ c) + \sum_{q:Q\ ((\phi\ \xi_0)\ c)} \mathbf{B}\ \imath\ ((\phi\ \xi_1)\ c\ q)\Big) \cong \mathbf{B}\ \imath\ c$$

We use this as our definition of $\mathbf{B}$, which we hereafter call Pos, as an inductive family over C. In our code, Pos is also parameterised by a fixed point ϕ.

```
data Pos (ϕ : FixedPoint) (ı : I) : ϕ .C → Type where
  here : {c : ϕ .C} → P ı ((ϕ χ⁻¹₀) c) → Pos ϕ ı c
  below : {c : ϕ .C} (q : Q ((ϕ χ⁻¹₀) c)) → Pos ϕ ı ((ϕ χ⁻¹₁) c q) → Pos ϕ ı c
```

It turns out that Pos works for both the initial and the terminal case: we set $\mathbf{B}_\mu$ = Pos WAlg and $\mathbf{B}_\nu$ = Pos MAlg. It is not immediately clear that choosing $\mathbf{B}_\nu$ to be an inductive (and not coinductive) family over M S Q would be the right choice in the coinductive case, so we explain why this works in more detail.

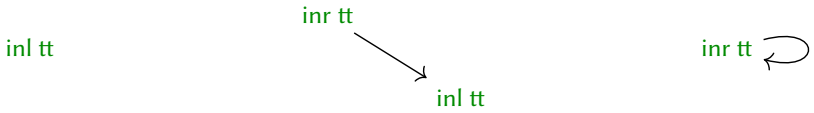
Intuitively, we can think of Pos as the type of finite paths through a W or M tree. To see this more clearly, we look at Pos specified to ϕ = MAlg, $I = \top$, and $\mathbf{P}$ tt $s := \top$ (which implies we only consider unary containers), which would be equivalent to PosM below.

```
data PosM : M S Q → Type where
  here : {m : M S Q} → PosM m
  below : {m : M S Q} (q : Q (shape m)) → PosM ((pos m) q) → PosM m
```

Now, as an example, recall from Example 2.1.1 that if we set S = S and Q = P, where

$$\begin{aligned} \mathsf{S} &= \top + \top \\ \mathsf{P}\ (\mathsf{inl}\ \_) &= \bot \\ \mathsf{P}\ (\mathsf{inr}\ \_) &= \top, \end{aligned}$$

we have that M S P ≅ ℕ∞. Also recall that the M trees encoding 0, 1, and ∞ respectively of type ℕ∞ are as follows.



Now we look at the elements of PosM (M S P), or equivalently PosM ℕ∞, for the elements 0, 1, and ∞ of ℕ∞. For the first tree (encoding 0), PosM would consist solely of the element here , because we cannot construct anything via below, since P (inl tt) is empty. For the second tree (encoding 1), PosM consists of here and below tt here, since P (inr tt) is now ⊤. For the third tree (encoding ∞), PosM consists of here, below tt here, below tt (below tt here), and so on, ad infinitum. Although M trees can have infinite paths, like in the third case, any position (i.e. where data is stored in the tree, even though this example does not involve payloads) is obtained via a finite path, and since PosM encodes exactly the finite paths, it is precisely what is required. We verify this is actually the case in Section 2.3.

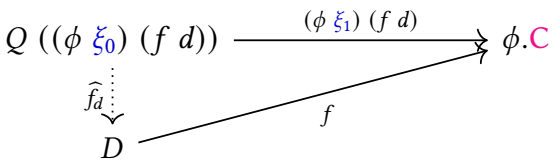
2.2.2 Generalised induction principle for Pos

We take the opportunity to mention the induction principle for Pos, which will come in useful later. In general, given a fixed point ϕ, an index $\iota : I$, and a family of types $A : (c : \phi\ .\mathsf{C}) \to \mathsf{Pos}\ \phi\ \iota\ c \to \mathsf{Type}$ equipped with

- $h : \{c : \phi\ .\mathsf{C}\}\ (p : P\ \iota\ ((\phi\ \xi_0)\ c)) \to A\ c\ (\mathsf{here}\ p)$
- $b : \{c : \phi\ .\mathsf{C}\}\ (q : Q\ ((\phi\ \xi_0)\ c))\ (p : \mathsf{Pos}\ \phi\ \iota\ ((\phi\ \xi_1)\ c\ q)) \to A\ ((\phi\ \xi_1)\ c\ q)\ p \to A\ c\ (\mathsf{below}\ q\ p)$

induces, in the obvious way, a dependent function $(c : \phi\ .\mathsf{C})\ (p : \mathsf{Pos}\ \phi\ \iota\ c) \to A\ c\ p$. In `Cubical Agda`, this is precisely the induction principle we get from performing a standard pattern matching. In practice, however, this induction principle is quite limited. The primary difficulty we run into is in the case where A is only defined over $(d : D)$ and $\mathsf{Pos}\ \phi\ \iota\ (f\ d)$ for some fixed function $f : D \to \phi\ .\mathsf{C}$. In this case, the induction principle above does not apply since A is not defined over all of $\phi\ .\mathsf{C}$. This is analogous to how path induction does not apply to paths with both endpoints fixed. There are special cases when the induction principle is still applicable: for instance, when f is a retraction. In fact, we only need f to satisfy a weaker property, namely the following.

Definition 2.2.2. Given a fixed point ϕ, a function $f : D \to \phi\ .\mathsf{C}$ is called a *ϕ-retraction* if for any $d : D$, the lift $\widehat{f_d}$ in the diagram below exists.



◊

What f being a ϕ-retraction means informally is that for every $d : D$, when we look

at the 'position' part of $f\ d$ (i.e. $(\phi\ \xi_1)\ (f\ d)$), it is still of the form $f\ d'$ for some other $d' : D$. We see an application of this later in Lemma 2.3.11.

Lemma 2.2.3 (Generalised Pos induction)**.** *Let ϕ be a fixed point, $\iota : I$ an index, and $f : D \to \phi\ .C$ a ϕ-retraction. Let $A : (d : D) \to \mathsf{Pos}\ \phi\ \iota\ (f\ d) \to \mathsf{Type}$ be a dependent type equipped with*

- $h : \{d : D\}\ (p : P\ \iota\ ((\phi\ \xi_0)\ (f\ d))) \to A\ d\ (\mathsf{here}\ p)$
- $b : \{d : D\}\ (q : Q\ ((\phi\ \xi_0)\ (f\ d)))\ (p : \mathsf{Pos}\ \phi\ \iota\ ((\phi\ \xi_1)\ (f\ d)\ q)) \to$
  $A\ (\widehat{f_d}\ q)\ \widetilde{p} \to A\ d\ (\mathsf{below}\ q\ p)$

where $\widetilde{p}$ is p transported along the witness of the fact the diagram in Definition 2.2.2 commutes. This data induces a dependent function $(d : D)\ (p : \mathsf{Pos}\ \phi\ \iota\ (f\ d)) \to A\ d\ p$.

Proof sketch. The induction principle follows immediately from the usual induction principle for Pos but with the family $\widehat{A} : (c : \phi\ .C) \to \mathsf{Pos}\ \phi\ \iota\ c \to \mathsf{Type}$ defined by

$$\widehat{A}\ c\ p \coloneqq (d : D)(t : c \equiv f\ d) \to A\ d\ \widetilde{p}$$

where $\widetilde{p}$ denotes the result of transporting p along $t : c \equiv f\ d$. We obtain the appropriate statement by setting $c \coloneqq f\ d$. □

2.3 Fixed Points

Let us now show that the constructions from Section 2.2 are correct: $[\![\mathsf{W}\ S\ Q \triangleleft \mathsf{Pos}\ \mathsf{WAlg}]\!]\ \mathbf{X}$ is the initial $[\![F]\!](\mathbf{X}, -)$-algebra carrier, and $[\![\mathsf{M}\ S\ Q \triangleleft \mathsf{Pos}\ \mathsf{MAlg}]\!]\ \mathbf{X}$ is the terminal $[\![F]\!](\mathbf{X}, -)$-coalgebra carrier. The proofs in this section mostly follow those given in [AAG05], but in the more general (UIP-free) setting of Type instead of Set.

We start off by showing that $[\![\mathsf{W}\ S\ Q \triangleleft \mathsf{Pos}\ \mathsf{WAlg}]\!]\ \mathbf{X}$ is the initial $[\![S \triangleleft \mathbf{P}, Q]\!](\mathbf{X}, -)$-algebra carrier. This proof is relatively straightforward.

Theorem 2.3.1. *Let $F = (S \triangleleft \mathbf{P}, Q)$ be a container in $Ind + 1$ parameters with $S : \mathsf{Type}, \mathbf{P} : Ind \to S \to \mathsf{Type}, Q : S \to \mathsf{Type}$. For any fixed $\mathbf{X} : Ind \to \mathsf{Type}$, the type $[\![\mathsf{W}\ S\ Q \triangleleft \mathsf{Pos}\ \mathsf{WAlg}]\!]\ \mathbf{X}$ is the carrier of the initial algebra of $[\![F]\!](\mathbf{X}, -) : \mathsf{Type} \to \mathsf{Type}$, i.e.*

$$[\![\mathsf{W}\ S\ Q \triangleleft \mathsf{Pos}\ \mathsf{WAlg}]\!]\ \mathbf{X} \cong \mu Y.[\![F]\!](\mathbf{X}, Y).^{2}$$

Proof of Theorem 2.3.1. We write W for $\mathsf{W}\ S\ Q$ and $\mathsf{Pos}\mu$ for $\mathsf{Pos}\ \mathsf{WAlg}$. We construct an $[\![F]\!](\mathbf{X}, -)$-algebra with carrier $[\![\mathsf{W} \triangleleft \mathsf{Pos}\mu]\!]\ \mathbf{X}$ by defining a morphism

$$\mathsf{into} : [\![F]\!](\mathbf{X}, [\![\mathsf{W} \triangleleft \mathsf{Pos}\mu]\!]\ \mathbf{X}) \to [\![\mathsf{W} \triangleleft \mathsf{Pos}\mu]\!]\ \mathbf{X}$$

by induction on $\mathsf{Pos}\mu$ as follows, where $s : S, f : Q\ s \to \mathsf{W}, g : (\iota : Ind) \to \mathbf{P}\ \iota\ s \to \mathbf{X}\ \iota$, and $h : (\iota : Ind)(q : Q\ s) \to \mathsf{Pos}\mu\ \iota\ (f\ q) \to \mathbf{X}\ \iota$.[3]

[2] We use the notation $\iota : Ind$ for indices to distinguish them from interval variables $i : I$.

[3] We repackage the type of the input of into via Equation 2.1.5 and distributivity of functions over Σ. This also applies for the definition of out in Theorem 2.3.4.

```
fst (into ((s , f) , g , h)) = sup-W s f
snd (into ((s , f) , g , h)) ι (here p) = g ι p
snd (into ((s , f) , g , h)) ι (below q b) = h ι q b
```

Then $(\llbracket \mathsf{W} \triangleleft \mathsf{Pos}\mu \rrbracket\, \mathbf{X}, \mathsf{into})$ is an $\llbracket F \rrbracket(\mathbf{X}, -)$-algebra. Now for any algebra (Y, α), we need to define $\overline{\alpha} : \llbracket \mathsf{W} \triangleleft \mathsf{Pos}\mu \rrbracket\, \mathbf{X} \to Y$ uniquely such that the below diagram commutes.

$$\begin{array}{ccc} \llbracket F \rrbracket(\mathbf{X}, \llbracket \mathsf{W} \triangleleft \mathsf{Pos}\mu \rrbracket\, \mathbf{X}) & \xrightarrow{\mathsf{into}} & \llbracket \mathsf{W} \triangleleft \mathsf{Pos}\mu \rrbracket\, \mathbf{X} \\ {\scriptstyle \llbracket F \rrbracket(\mathbf{X}, \overline{\alpha})}\downarrow & & \downarrow{\scriptstyle \overline{\alpha}} \\ \llbracket F \rrbracket(\mathbf{X}, Y) & \xrightarrow[\alpha]{} & Y \end{array} \tag{2.3.2}$$

We define $\overline{\alpha} : \sum_{w : \mathsf{W}} ((\iota : \mathit{Ind}) \to \mathsf{Pos}\mu\ \iota\ w \to \mathbf{X}\ \iota) \to Y$ by induction on W.[4]

```
ᾱ : Σ[ w ∈ W S Q ] ((ι : Ind) → Posμ ι w → X ι) → Y
ᾱ (sup-W s f , k) = α (s , g , λ q → ᾱ (f q , λ ι → h ι q))
  where
    g : (ι : Ind) → P ι s → X ι
    g ι p = k ι (here p)

    h : (ι : Ind) → (q : Q s) → Posμ ι (f q) → X ι
    h ι q b = k ι (below q b)
```

That (2.3.2) commutes then follows definitionally.

The only thing left to show is that $\overline{\alpha}$ is unique. We assume there is another arrow $\tilde{\alpha} : \llbracket \mathsf{W} \triangleleft \mathsf{Pos}\mu \rrbracket\, \mathbf{X} \to Y$ making (2.3.2) commute, i.e.

$$\tilde{\alpha} \circ \mathsf{into} \equiv \alpha \circ \llbracket F \rrbracket(\mathbf{X}, \tilde{\alpha}), \tag{2.3.3}$$

and prove that for $w : \mathsf{W}, k : \mathsf{Pos}\mu\ \iota\ w \to \mathbf{X}\ \iota$, we have $\tilde{\alpha}(w, k) \equiv \overline{\alpha}(w, k)$. By induction on W, we have to show that for $s : S$ and $f : Q\ s \to \mathsf{W}$, we have $\tilde{\alpha}(\mathsf{sup\text{-}W}\ s\ f, k) \equiv \overline{\alpha}(\mathsf{sup\text{-}W}\ s\ f, k)$. This follows easily from $\overline{\alpha}$'s definition, assumption (2.3.3), and our inductive hypothesis. □

Next, we show that $\llbracket \mathsf{M}\ S\ Q \triangleleft \mathsf{Pos}\ \mathsf{MAlg} \rrbracket\, \mathbf{X}$ is the terminal $\llbracket S \triangleleft \mathbf{P}, Q \rrbracket(\mathbf{X}, -)$-coalgebra carrier. This proof is significantly more challenging than the previous one, both theoretically in that we use a modified version of the induction principle for Pos, and also technically in that we have to go through several workarounds for `Cubical Agda` to accept our proof. It also requires us to use a considerable amount of path algebra to prove coherences that are not needed when assuming UIP, which appears to be implicitly assumed in the original proof.

Theorem 2.3.4. *Let $F = (S \triangleleft \mathbf{P}, Q)$ be a container in $\mathit{Ind} + 1$ parameters with $S : \mathsf{Type}, \mathbf{P} : \mathit{Ind} \to S \to \mathsf{Type}$, and $Q : S \to \mathsf{Type}$. For any fixed $\mathbf{X} : \mathit{Ind} \to \mathsf{Type}$, the type $\llbracket \mathsf{M}\ S\ Q \triangleleft \mathsf{Pos}\ \mathsf{MAlg} \rrbracket\, \mathbf{X}$ is the carrier of the terminal coalgebra of $\llbracket F \rrbracket(\mathbf{X}, -) : \mathsf{Type} \to \mathsf{Type}$, i.e.*

$$\llbracket \mathsf{M}\ S\ Q \triangleleft \mathsf{Pos}\ \mathsf{MAlg} \rrbracket\, \mathbf{X} \cong \nu Y. \llbracket F \rrbracket(\mathbf{X}, Y).$$

[4] Technically, this definition raises a termination checking error, but this is easily fixed in the actual code by defining the uncurried version first then writing $\overline{\alpha}$ in terms of it.

Before we prove Theorem 2.3.4, we spell out what it is we need to show. We write M for M S Q and Posν for Pos MAlg. First, we construct an $[\![F]\!](\mathbf{X}, -)$-coalgebra with carrier $[\![\mathsf{M} \triangleleft \mathsf{Pos}\nu]\!]\,\mathbf{X}$ by defining

$$\mathsf{out} : [\![\mathsf{M} \triangleleft \mathsf{Pos}\nu]\!]\,\mathbf{X} \to [\![F]\!](\mathbf{X}, [\![\mathsf{M} \triangleleft \mathsf{Pos}\nu]\!]\,\mathbf{X})$$

roughly as into^{-1}, where into is the function from Theorem 2.3.1.

$$\begin{aligned}&\mathsf{out}\ (m\ ,\ k) = (\mathsf{shape}\ m\ ,\ \mathsf{pos}\ m)\ ,\\ &\qquad((\lambda\ \imath\ p \to k\ \imath\ (\mathsf{here}\ p))\ ,\ (\lambda\ \imath\ q\ b \to k\ \imath\ (\mathsf{below}\ q\ b)))\end{aligned}$$

So $([\![\mathsf{M} \triangleleft \mathsf{Pos}\nu]\!]\,\mathbf{X}, \mathsf{out})$ is an $[\![F]\!](\mathbf{X}, -)$-coalgebra. For any coalgebra (Y, β), we need to define $\overline{\beta} : Y \to [\![\mathsf{M} \triangleleft \mathsf{Pos}\nu]\!]\,\mathbf{X}$ uniquely, such that the below diagram commutes.

$$\begin{array}{ccc} Y & \xrightarrow{\quad\beta\quad} & [\![F]\!](\mathbf{X}, Y) \\ {\scriptstyle\overline{\beta}}\downarrow & & \downarrow{\scriptstyle [\![F]\!](\mathbf{X},\overline{\beta})} \\ [\![\mathsf{M} \triangleleft \mathsf{Pos}\nu]\!]\,\mathbf{X} & \xrightarrow[\mathsf{out}]{} & [\![F]\!](\mathbf{X}, [\![\mathsf{M} \triangleleft \mathsf{Pos}\nu]\!]\,\mathbf{X}) \end{array} \tag{2.3.5}$$

To this end, from now on we fix $\beta : Y \to [\![F]\!](\mathbf{X}, Y)$ with the following components.

$$\begin{aligned}&\beta s : Y \to S\\ &\beta g : (y : Y)\ (\imath : \mathit{Ind}) \to \mathbf{P}\ \imath\ (\beta s\ y) \to \mathbf{X}\ \imath\\ &\beta h : (y : Y) \to Q\ (\beta s\ y) \to Y\end{aligned}$$

To prove Theorem 2.3.4 we (i) construct $\overline{\beta} : Y \to \sum_{m:\mathsf{M}} ((\imath : \mathit{Ind}) \to \mathsf{Pos}\nu\ \imath\ m \to \mathbf{X}\ \imath)$ such that (2.3.5) commutes and (ii) show that this $\overline{\beta}$ is unique. This will be the content of Lemmas 2.3.6 and 2.3.8.

Lemma 2.3.6. *There is a function* $\overline{\beta} : Y \to \sum_{m:\mathsf{M}} ((\imath : \mathit{Ind}) \to \mathsf{Pos}\nu\ \imath\ m \to \mathbf{X}\ \imath)$ *making* (2.3.5) *commute.*

Proof. We will define $\overline{\beta}$ by

$$\begin{aligned}&\overline{\beta} : Y \to \Sigma[\ m \in \mathsf{M}\ ]\ ((\imath : \mathit{Ind}) \to \mathsf{Pos}\nu\ \imath\ m \to X\ \imath)\\ &\overline{\beta}\ y = \overline{\beta}_1\ y\ ,\ \overline{\beta}_2\ y\end{aligned}$$

where $\overline{\beta}_1 : Y \to \mathsf{M}$ is defined by coinduction on M and $\overline{\beta}_2 : (y : Y)\ (\imath : \mathit{Ind}) \to \mathsf{Pos}\nu\ \imath\ (\overline{\beta}_1\ y) \to \mathbf{X}\ \imath$ is defined by induction on Posν as follows.

$$\begin{aligned}&\overline{\beta}_1 : Y \to \mathsf{M}\\ &\mathsf{shape}\ (\overline{\beta}_1\ y) = \beta s\ y\\ &\mathsf{pos}\ (\overline{\beta}_1\ y) = \overline{\beta}_1 \circ (\beta h\ y)\end{aligned}$$

$$\begin{aligned}&\overline{\beta}_2 : (y : Y)\ (\imath : \mathit{Ind}) \to \mathsf{Pos}\nu\ \imath\ (\overline{\beta}_1\ y) \to X\ \imath\\ &\overline{\beta}_2\ y\ \imath\ (\mathsf{here}\ p) = \beta g\ y\ \imath\ p\\ &\overline{\beta}_2\ y\ \imath\ (\mathsf{below}\ q\ p) = \overline{\beta}_2\ (\beta h\ y\ q)\ \imath\ p\end{aligned}$$

This construction makes (2.3.5) commute by definition. □

To show that $\overline{\beta}$ is unique, we assume there is another arrow $\tilde{\beta} : Y \to [\![\mathsf{M} \triangleleft \mathsf{Pos}\nu]\!]\,\mathbf{X}$ making the above diagram commute, i.e.

$$\mathsf{out} \circ \tilde{\beta} \equiv [\![F]\!](\mathbf{X}, \tilde{\beta}) \circ \beta, \tag{2.3.7}$$

then show that $\tilde{\beta} \equiv \overline{\beta}$. Naming $\tilde{\beta}$'s first and second projections $\tilde{\beta}_1$ and $\tilde{\beta}_2$, assumption (2.3.7) can be split up into the paths shown below. We remark that all but the first one of these paths are dependent paths. In what follows, we fix $y : Y$.

$$comm_1\ y : \mathsf{shape}\ (\tilde{\beta}_1\ y) \equiv \beta s\ y$$
$$comm_2\ y : \mathsf{pos}\ (\tilde{\beta}_1\ y) \cong (\lambda\ q \to \tilde{\beta}_1\ (\beta h\ y\ q))$$

↑ dependent path over $(\lambda\ i \to Q\ (comm_1\ y\ i) \to \mathsf{M})$

$$comm_3\ y : (\lambda\ \imath\ p \to \tilde{\beta}_2\ y\ \imath\ (\mathsf{here}\ p)) \cong \beta g\ y$$

↑ dependent path over $(\lambda\ i \to (\imath : Ind) \to P\ \imath\ (comm_1\ y\ i) \to X\ \imath)$

$$comm_4\ y : (\lambda\ \imath\ q\ b \to \tilde{\beta}_2\ y\ \imath\ (\mathsf{below}\ q\ b)) \cong (\lambda\ \imath\ q\ b \to \tilde{\beta}_2\ (\beta h\ y\ q)\ \imath\ b)$$

↑ dependent path over $(\lambda\ i \to (\imath : Ind)(q : Q\ (comm_1\ y\ i)) \to \mathsf{Pos}\nu\ \imath\ (comm_2\ y\ i\ q) \to X\ \imath)$

These equations simply express the fact that for $\tilde{\beta}$ to make diagram (2.3.5) commute, $\tilde{\beta}_1$ and $\tilde{\beta}_2$ have to be defined in the same way component-wise as $\overline{\beta}_1$ and $\overline{\beta}_2$, up to a path.

Lemma 2.3.8. *The function* $\overline{\beta} : Y \to \sum_{m:\mathsf{M}} ((\imath : Ind) \to \mathsf{Pos}\nu\ \imath\ m \to \mathbf{X}\ \imath)$ *from Lemma 2.3.6 is unique. In other words, under the assumption that* $comm_1$*–*$comm_4$ *above exist, we can construct the following paths.*

$$\mathsf{fstEq} : (y : Y) \to \tilde{\beta}_1\ y \equiv \overline{\beta}_1\ y$$
$$\mathsf{sndEq} : (y : Y) \to \tilde{\beta}_2\ y \cong \overline{\beta}_2\ y$$

↑ dependent path over $(\lambda i \to (\imath : Ind) \to \mathsf{Pos}\ \imath\ (\mathsf{fstEq}\ y\ i)\ X\ \imath)$

Proof of Lemma 2.3.8, part 1: construction of fstEq. Recall M's coinduction principle MCoind from Section 2.1. Using this, we can prove fstEq in a rather straightforward manner. To apply MCoind, we need to construct a binary relation R on M. We construct it as an inductive family that relates precisely those terms we need to prove equal, i.e. $\tilde{\beta}_1\ y$ and $\overline{\beta}_1\ y$.

```
data R : M → M → Type where
  R-intro : (y : Y) → R (β̃₁ y) (β̄₁ y)
```

We then prove that it is a bisimulation using copattern matching.

```
isBisimR : {m₀ m₁ : M} → R m₀ m₁ → M-R R m₀ m₁
s-eq (isBisimR (R-intro y)) = comm₁ y
p-eq (isBisimR (R-intro y)) q₀ q₁ q-eq = transport ... (R-intro (βh y q₁))
```

↑ Here, the second goal is of type R (pos ($\tilde{\beta}_1$ y q_0)) ($\overline{\beta}_1$ (βh y q_1)) while R-intro (βh y q_1) is of type R ($\tilde{\beta}_1$ (βh y q_1)) ($\overline{\beta}_1$ (βh y q_1)). This mismatch is adjusted using $comm_2$. Explicitly, we transport over the path of types ($\lambda\, i \to$ R ($comm_2$ y ($\sim i$) (q-eq ($\sim i$)))).

This allows us to finish the construction of fstEq.

fstEq y = MCoind R isBisimR (R-intro y) □

Before we continue with the construction of sndEq, we briefly discuss some of the finer points concerning the construction of fstEq. Because we used MCoind and isBisimR to construct fstEq, its definition is somewhat opaque. Fortunately, the construction is well-behaved on shape and thus the action of shape on (fstEq y) computes definitionally to $comm_1$ y. This means that the action of pos on (fstEq y) can be viewed as a dependent path pos ($\tilde{\beta}_1$ y) $\cong$ $\overline{\beta}_1 \circ (\beta h\ y)$ over the path of types ($\lambda\ i \to Q$ ($comm_1$ y i) $\to$ M). There is another canonical element of this type obtained by simply composing $comm_2$ with a corecursive call of fstEq — let us call it fstEqPos. It is defined as the composition of paths

$$\text{pos}\ (\tilde{\beta}_1\ y) \overset{comm_2\ y}{\rightsquigarrow} (\lambda\ q \to \tilde{\beta}_1\ (\beta h\ y\ q)) \xrightarrow{\text{fstEq} \circ (\beta h\ y)} (\lambda\ q \to \overline{\beta}_1\ (\beta h\ y\ q)) \tag{2.3.9}$$

where the squiggly arrow indicates that ($comm_2$ y) is a dependent path. We can now ask whether pos on (fstEq y) computes to (fstEqPos y) (which in essence just says that fstEq satisfies the obvious coinductive computational rule). This would be entirely trivial if we had assumed UIP, since we would be comparing equalities on h-sets, but now becomes something we cannot take for granted. Fortunately, it turns out we can still prove it.

Lemma 2.3.10. *For all $y : Y$, we have* fstEqPos $y \equiv (\lambda\ i \to$ pos (fstEq y i)).

Proof sketch of Lemma 2.3.10. The lemma is proved by abstracting and applying function extensionality and path induction on $comm_1$. In this special case, i.e. when $comm_1$ y = refl, one can simplify the instances of isBisimR and MCoind used in the construction of fstEq. We omit the details which are just technical path algebraic manipulations and refer the reader to the formalisation. □

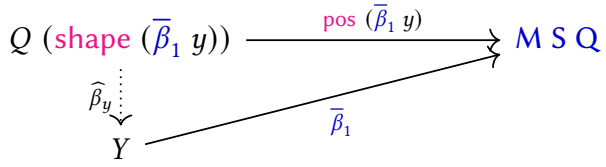
One may reasonably ask why this is a lemma and not simply part of the *definition* of fstEq. We discuss this in Section 2.3.1.

Let us now move on to the construction of sndEq. We construct sndEq using Lemma 2.2.3 which requires the following fact.

Lemma 2.3.11. $\overline{\beta}_1 : Y \to$ M *is an* MAlg-retraction.

Proof. For each $y : Y$, we need to construct a function $\widehat{\beta}_y : Q$ (shape ($\overline{\beta}_1$ y)) $\to Y$ such

that the below diagram commutes.



By unfolding the definition of $\overline{\beta}_1\, y$, this equality can be stated as follows for each $q : Q\ (\beta s\ y)$.

$$\overline{\beta}_1\ (\widehat{\beta}_y\ q) \equiv_{\mathsf{M\,S\,Q}} \overline{\beta}_1\ (\beta h\ y\ q) \tag{2.3.12}$$

Defining $\widehat{\beta}_y = \beta h\ y$ makes (2.3.12) hold definitionally. □

Finally, we are ready to construct sndEq and thereby finish the proof of Lemma 2.3.8.

Proof of Lemma 2.3.8, part 2: construction of sndEq. For ease of notation, we define, for each $\imath : \mathit{Ind}$ and $y : Y$, the function

$$\begin{aligned}&\mathsf{tr}\ y \colon \mathsf{Pos}\nu\ \imath\ (\overline{\beta}_1\ y) \to \mathsf{Pos}\nu\ \imath\ (\tilde{\beta}_1\ y)\\ &\mathsf{tr}\ y \coloneqq \mathsf{transport}\ (\lambda\ i \to \mathsf{Pos}\nu\ \imath\ ((\mathsf{fstEq}\ y)^{-1}\ i))\end{aligned}$$

For the construction of sndEq, we first note that, by function extensionality and the interchangeability of dependent paths and transports, constructing sndEq is equivalent to showing that

$$\tilde{\beta}_2\ y\ \imath\ (\mathsf{tr}\ y\ t) \equiv \overline{\beta}_2\ y\ \imath\ t \tag{2.3.13}$$

for $\imath : \mathit{Ind}$ and $t : \mathsf{Pos}\nu\ \imath\ (\overline{\beta}_1\ y)$. In light of Lemma 2.3.11, we may apply Lemma 2.2.3 in order to induct on t. When t is of the form here p, there is not much to show. Indeed, by translating this instance of (2.3.13) back into the language of dependent paths, we see that the data is given precisely by comm_3.

When t is of the form below $q\ p$, we may assume inductively that we have a path

$$\tilde{\beta}_2\ (\beta h\ y\ q)\ \imath\ (\mathsf{tr}\ (\beta h\ y\ q)\ p) \equiv \overline{\beta}_2\ (\beta h\ y\ q)\ \imath\ p \tag{2.3.14}$$

and the goal is to show that

$$\tilde{\beta}_2\ y\ \imath\ (\mathsf{tr}\ y\ (\mathsf{below}\ q\ p)) \equiv \overline{\beta}_2\ y\ \imath\ (\mathsf{below}\ q\ p). \tag{2.3.15}$$

The RHS of (2.3.15) is, by definition, equal to the RHS of (2.3.14). By commuting transports with below and using comm_4, we can rewrite the LHS of (2.3.14) to a term of the form $\tilde{\beta}_2\ y\ \imath\ (\mathsf{below}\ (\mathsf{transport}\ \ldots\ q)\ (\mathsf{transport}\ \ldots\ p))$. Commuting transports with below in the LHS of (2.3.15), we get a term of the same form, albeit with transports over slightly different families. Thus, it remains to equate these families. We spare the reader the technical details and simply point out that this task, after some path algebra, boils down to precisely Lemma 2.3.10. This concludes the proof of Lemma 2.3.8 and thus also of Theorem 2.3.4. □

Example 2.3.16. For a concrete example of Theorems 2.3.1 and 2.3.4, consider S, P_0, and P_1 as defined in Example 1.3.8. Then for a fixed $X : \mathsf{Type}$,

$$F_{\mathsf{List}} = [\![\mathsf{S} \triangleleft \mathsf{P}_0, \mathsf{P}_1]\!](-, X) = \top + (- \times X) : \mathsf{Type} \to \mathsf{Type},$$

the signature functor for List, has an initial algebra with carrier $[\![\mathbb{N} \triangleleft \mathsf{Fin}]\!]\, X$, and it has a terminal coalgebra with carrier $[\![\mathbb{N}\infty \triangleleft \mathsf{Cofin}]\!]\, X$. $\mathbb{N}\infty$ is defined as in Example 2.1.1 and Cofin is the inductive type family over $\mathbb{N}\infty$ of finite and (countably) infinite sets. $\mathbb{N} \triangleleft \mathsf{Fin}$ is the container representation of lists while $\mathbb{N}\infty \triangleleft \mathsf{Cofin}$ is the container representation of colists (the type of finite and infinite lists). ◊

2.3.1 The absence of UIP and Agda's termination checker

One of the key contributions of this chapter is the fact we were able to formalise Lemma 2.3.8 without using UIP. Our main difficulty was proving the technical Lemma 2.3.10 which essentially says that fstEq is coinductively defined in the obvious manner. In theory, `Cubical Agda` should allow us to define fstEq as

```
shape (fstEq y i) = comm₁ y i
pos (fstEq y i) = fstEqPos i
```

where we recall that fstEqPos is defined by coinductively calling fstEq as in (2.3.9). This construction would make Lemma 2.3.10 hold definitionally, without requiring any form of UIP. There are, however, two difficulties that come up with this. Firstly, `Agda` does not accept this definition and raises a termination checking error. We believe this to be an issue with `Cubical Agda`'s current termination checker. Generally speaking, in order to check whether a corecursive function terminates, `Agda` needs to ensure its output can be produced in a finite amount of steps. We call such functions *productive*. In the cases when it is not obvious from the structure of the code that it is productive, e.g. if we make a corecursive call and do something else with it before returning, rather than returning directly, `Agda` usually raises a termination error. While this is justified in general, composing productive calls using `Cubical Agda`'s primitive path composition function, hcomp, should be productive, but `Agda` still raises an error. This was raised as a GitHub issue [Alt20].

If the first issue were to be resolved, our proof of Theorem 2.3.4 could be made significantly shorter, as we would not need to use M's coinduction principle MCoind in the definition of fstEq. Nevertheless, there is another drawback with such a construction of fstEq, namely that it relies heavily on the intricacies of cubical type theory (specifically when we introduce a path variable i and then copattern match). As a result, we could not expect to reproduce such a proof in other non-cubical type theories. Therefore, from a mathematical perspective, the issue we faced with the termination checker may have been a blessing in disguise. Our original motivation behind formalising Lemma 2.3.8 was to support the claim by Abbott et al. [AAG05] that the theory of containers can be understood in *type theory*. Here, we aim to interpret *type theory* as generally as possible, rather than restricting ourselves to cubical type theory. Since we had to define fstEq using MCoind rather than the `Cubical Agda`-specific construction above, our proof should hold in any type theory having function extensionality (e.g. setoid type theory

[ABKT19]), W-types (M-types can be derived from inductive types, see [ACS15]), and coinduction for M-types. The fact that the authors of [AAG05] never mention any results akin to Lemma 2.3.10 suggests that they worked under a tacit assumption of UIP, which is further evidence that our formalisation indeed is a generalisation of previous results.

2.4 Conclusion & Related Work

In this chapter, we presented a formalisation of the following results on containers, doing so without making any h-set assumptions, thereby generalising the original results from the category of sets to the wild category of types.

- $[\![\mathsf{W}\ S\ Q \triangleleft \mathsf{Pos}\ \mathsf{WAlg}]\!]\ \mathbf{X}$ is (the carrier of) the initial $[\![S \triangleleft \mathbf{P}, Q]\!](\mathbf{X}, -)$-algebra, and
- $[\![\mathsf{M}\ S\ Q \triangleleft \mathsf{Pos}\ \mathsf{MAlg}]\!]\ \mathbf{X}$ is (the carrier of) the terminal $[\![S \triangleleft \mathbf{P}, Q]\!](\mathbf{X}, -)$-coalgebra.

While the first proof was straightforward, the second proof needed more careful consideration. In particular, it employed a modified version of Pos's induction principle and required various workarounds for `Cubical Agda` to accept our proof. However, having done these workarounds means that our construction should hold not only in *cubical* type theory, but in any type theory with function extensionality, W- (and M-) types, and MCoind.

Similar results have been formalised in `Lean` by Avigad et al. [ACH19], who utilised a variation on bounded natural functors in their formalisation, although to our knowledge our formalisation is the first one not relying on UIP. Vezzosi [Vez17] wrote about the semantics of allowing path abstraction in copattern matching definitions in `Cubical Agda`. Ahrens et al. [ACS15] formalised M-types in `Cubical Agda` as limits of chains, so that their definition does not use the coinductive keyword. Since our goal was not only to establish the theoretical results in a more general setting, but also to make use of `Cubical Agda`'s support for proving equalities on coinductives, we chose to use native coinductive types and defined M as shown in Section 2.1.2. Nevertheless, it is possible that using their definition might have circumvented the issues we faced with the termination checker, due to avoiding the use of native coinductive types, although their approach relies on the univalence axiom, while ours does not.

Having formalised these results on fixed points of containers without making h-set assumptions suggests that equivalent results should hold for the more general groupoid (or symmetric) containers and categorified containers [Gyl11; Alt24]. These more general kinds of containers are of interest as they can describe a larger class of types, such as the type of multisets, which are not covered by h-set based container definitions. Their theory is however not fully worked out yet, so we leave this for future work.

Chapter 3

A Container Model of Type Theory

One of our long-term research goals is to extend the theory of containers, which already provides semantics for simple inductive types and inductive families, to the more general QIITs. When we started looking into this, it became clear that as a prerequisite to this approach, we need a way to model type theory via containers. We start by motivating these two problems and explaining why the latter is a prerequisite to the former. Most of this chapter focusses on presenting the container model of type theory first proposed by Altenkirch and Kaposi, using groupoid categories with families (GCwFs) as our notion of model, which has been fully formalised in `Cubical Agda`. At the end of this chapter, we present some conjectures on obtaining QIIT semantics via generalised containers as a starting point for future work.

3.1 Motivation and Related Work

Providing semantics for HITs in HoTT continues to be an open problem. Despite there already being a lot of work in the literature making use of HITs (e.g. [LS13; Bru16; LM23]), they still lack a general specification and theoretical foundations. Several approaches have been proposed in the literature. For example, Coquand, Huber, and Mörtberg extend cubical type theory with a syntax for HITs [CHM18]; Lumsdaine and Shulman introduce the notion of 'cell monads with parameters' to model HITs [LS19]; Cavallo and Harper give a schema for HITs that are formulated in terms of cubical type theory's path type [CH19]. However, these contributions either work in a subsytem of HoTT (e.g. cubical type theory), or they describe specific examples as opposed to giving a general treatment of HITs. This means that we still do not have a precise definition of what a HIT is in general.

As a first step towards closing this gap, we look at giving semantics for set-truncated HITs with induction-induction, also known as QIITs. These types generalise dependent

types in two ways. Firstly, they not only allow the use of point constructors, but also path constructors up to first order equalities (i.e. these types are h-sets hence their paths are h-props). 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. More precisely, we can simultaneously define a type $A : \mathsf{Type}$ with a family $B\colon A \to \mathsf{Type}$ over A, where the constructors of A can refer to B. This means we can refer to a family (e.g. a predicate) over A when defining A itself.

The QIIT of the syntax of a very basic type theory is given below as an example. It is a *quotient* type due to eq being an equality constructor, and it is an *inductive-inductive* type due to the constructor eq of Con referring to the constructor σ of Ty, which in turn is a type family being defined over Con.[1]

```
data Con : Set
data Ty : Con → Set

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

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

Simple inductive types and inductive families already enjoy fully developed semantics in HoTT. Dybjer 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 [Dyb96b]. His approach works in extensional type theory but raises issues in intensional type theory (e.g., in this setting there are multiple representations of zero : $\mathbb{N}$, see [McB10]), although this was later fixed by Hugunin [Hug21]. These results establish W-types as the universal type for simple inductive types. Abbott et al. show that (n-ary) containers are a normal form for simple inductive types [AAG05], and use them to generalise Dybjer's result to nested inductive types and coinductive types, while providing a more complete description of the categorical infrastructure involved. Later, Altenkirch and Morris introduce indexed containers as a normal form for inductive families [AM09], and show that any inductive family can be represented as an indexed W-type, thereby identifying a universal type for inductive families. Sattler shows that indexed W-types can be reduced to W-types

[1] We note that this is written in 'pseudo-`Agda`' code for two reasons. Firstly, `Cubical Agda` does not (at the time of writing) accept the constructor eq, due to it referencing σ which is not yet defined. We could in theory get `Agda` to accept an encoding of this QIIT — more details on such an encoding can be found in von Raumer's thesis [vRau19] and Sestini's thesis [Ses23]. Secondly, for this type to be a *Q*IIT, we need to ensure that Con is an h-set and Ty is a family of h-sets. In the example, we do so by declaring the types of Con and Ty Γ to be Set instead of Type, but in reality, writing this in `Agda` would not enforce this. To achieve this in practice, we would have to add more equality constructors stating that any two equalities on any two elements of the declared types are equal.

[Sat15], showing that W-types are enough to represent both simple inductive types and inductive families. Once again, these results are stated in extensional type theory, but they can also be translated into intensional type theory as long as we have function extensionality. This is dealt with formally in [AGS12].

A similar well-established theoretical foundation for IITs and QIITs in HoTT has not yet been found. Probably the most up-to-date developments on this can be found in Kovacs's thesis [Kov22]. The first obstacle we face in formalising IITs is that due to the dependency we allow between constructors of different sorts, we do not have a way of expressing IITs as 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. remedy this for the even more general case of QIITs [ACD+18]. They specify a way to still represent QIITs as initial algebras, but instead of being initial F-algebras for some functor F, they are now initial 'dependent dialgebras'. They generalise the usual functorial semantics of inductive types to QIITs by starting off with a category of the sorts of a QIIT, and adding one constructor at a time, where the n^{th} constructor is represented by a pair of functors L_n (the left hand side, or the arguments) and R_n (the right hand side, or the target type, which can either be a sort or an equality). At the end of this process, once all the constructors are added, we end up with a category of dependent dialgebras, whose initial object corresponds to the QIIT. 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: it tells us that *if* a QIIT specification has an initial algebra, then we can describe it in a specific way. However, it doesn't tell us *which* QIIT specifications have initial algebras. This is precisely the problem we aim to tackle, namely, we want to provide a canonical way to represent QIIT specifications that admit an initial algebra, i.e. the strictly positive ones. The way this was achieved for simple inductive types and inductive families was using containers. We take inspiration from this and aim to 'containerify' the semantics given in [ACD+18], i.e. restrict the functors representing the constructors being added to some form of container functors (as well as adding some other restrictions), thereby only allowing strictly positive QIIT definitions.

Since a constructor is an expression in type theory and we want to express each constructor as a container, we need to be able to interpret any expression in type theory as some kind of 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 [AK21], but the details of this have not yet been worked out. Von Glehn has presented a polynomial functor model of type theory using comprehension categories as notion of model [vGle15]. The same model was also presented by Atkey [Atk20] and Kovács [Kov20] using categories with families (CwFs) as notion of model. We construct an alternative container model of type theory using CwFs, which has the same contexts and substitutions as the latter two but different types and terms.

3.2 Categories with Families

In this section, we state the original definition of a CwF and point out coherence issues that arise when using this definition in intensional type theory.

Developing a notion of semantics of an algebraic theory such as MLTT is useful for various reasons. One important reason is that having rigorous semantics provides us with a foundation on which we can prove properties about our theory. 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. Many different notions of model of type theory have been proposed over the years, such as Cartmell's contextual categories [Car86], Jacobs's comprehension categories [Jac93], and Dybjer's CwFs [Dyb96a].

We present here the notion of model that we are interested in, namely CwFs. The main reason we choose CwFs over other notions of model is that we would like to work *in* type theory, and CwFs can be described fairly straightforwardly in this setting over other notions which are more categorical. Indeed, one intuition for the definition of a CwF is the following. If we write down the intrinsic syntax of dependent type theory as a QIIT, algebras of this signature correspond to CwFs [AK16]. By 'intrinsic syntax' we mean the syntax written directly in type theory as an inductive type, as opposed to extrinsic syntax, where we'd have a notion of pre-terms, pre-types, and pre-contexts, and a typing relation such that the well-typed terms are those pre-terms for which there exist a pre-context and a pre-type making the relation hold. Our definition and discussion are adapted from [Dyb96a], [Hof97], and [Kap13].

Definition 3.2.1. A *category with families* (CwF) consists of:

- A category $\underline{\mathbf{C}}$ whose objects interpret contexts ($\Gamma, \Delta, \dots$) and whose morphisms interpret context substitutions ($\Delta \xrightarrow{\gamma} \Gamma, \dots$), having a terminal object $\diamond$ interpreting the empty context.
- A presheaf
$$\mathrm{Ty} : \underline{\mathbf{C}}^{\mathrm{op}} \to \underline{\mathbf{Set}}$$
whose object part interprets types and whose morphism part interprets type substitution. For a substitution $\gamma : \Delta \to \Gamma$, $\mathrm{Ty}\ \gamma : \mathrm{Ty}\ \Gamma \to \mathrm{Ty}\ \Delta$, and for $A : \mathrm{Ty}\ \Gamma$, we write $\mathrm{Ty}\ \gamma\ A$ as $A[\gamma]$ (the type A substituted by γ).
- A presheaf
$$\mathrm{Tm} : (\textstyle\int \mathrm{Ty})^{\mathrm{op}} \to \underline{\mathbf{Set}}$$
whose object part interprets terms and whose morphism part interprets term substitution. For a substitution $\gamma : \Delta \to \Gamma$ and types $A : \mathrm{Ty}\ \Gamma$ and $B : \mathrm{Ty}\ \Delta$, $\mathrm{Tm}\ \gamma : \mathrm{Tm}\ (\Gamma, A) \to \mathrm{Tm}\ (\Delta, B)$, and for $a : \mathrm{Tm}\ (\Gamma, A)$, we write $\mathrm{Tm}\ \gamma\ a$ as $a[\gamma]$ (the term a substituted by γ).
- A context comprehension operation which associates to every context Γ and type $A : \mathrm{Ty}\ \Gamma$, a context $\Gamma.A : |\underline{\mathbf{C}}|$ and
  - a morphism $\mathbf{p} : \Gamma.A \to \Gamma$ in $\underline{\mathbf{C}}$,
  - a term $\mathbf{q} : \mathrm{Tm}\ (\Gamma.A, A[\mathbf{p}])$,

– and a function $\langle \_, \_ \rangle \colon \sum_{\gamma \colon \Delta \to \Gamma} \mathrm{Tm}\,(\Delta, A[\gamma]) \to (\Delta \to \Gamma.A)$
expressing that

$$\Delta \to \Gamma.A \cong \sum_{\gamma \colon \Delta \to \Gamma} \mathrm{Tm}\,(\Delta, A[\gamma])$$

is a natural isomorphism. ◊

Intuitively, we can think of contexts as lists of terms, and substitutions as assignments of the terms in the target context possibly using terms in the source context.

Example 3.2.2. Say we have contexts

$$\Delta \coloneqq \{b : \mathsf{Bool}, n : \mathbb{N}, x : \mathsf{Fin}\, n\}$$
$$\Gamma \coloneqq \{m : \mathbb{N}, y : \mathsf{Fin}\, m\}.$$

Then a substitution $\gamma \colon \Delta \to \Gamma$ might look like

$$\gamma \coloneqq \langle \langle \epsilon, \mathsf{suc}\, n \rangle, \mathsf{if}\ (b \equiv \mathsf{true})\ \mathsf{then}\ (\mathsf{fs}\, x)\ \mathsf{else}\ \mathsf{fz} \rangle,$$

where $\epsilon \colon \Delta \to \diamond$ is the unique morphism into the empty context. The substitution γ gives an assignment to the variables in Γ using those in $\Delta - m$ is assigned the value $\mathsf{suc}\, n$ and if b is true, then y is assigned the value $\mathsf{fs}\, x : \mathsf{Fin}\,(\mathsf{suc}\, n)$, otherwise it is assigned the value $\mathsf{fz} : \mathsf{Fin}\,(\mathsf{suc}\, n)$. ◊

3.2.1 Coherence issues

The current state of the literature on models of type theory using CwFs as their notion of model presents an issue when we do not assume UIP, or in other words unless we work in an extensional setting. Several of these models, including the set model (or standard model) and the presheaf model, do not precisely fit Definition 3.2.1 of a CwF. We illustrate this issue via the set model below.

Example 3.2.3. The set model of type theory as a CwF is defined in extensional type theory as follows.

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

$$\mathrm{Ty}\ \Gamma \colon \mathrm{Set}$$
$$\mathrm{Ty}\ \Gamma \coloneqq \Gamma \to \mathrm{Set}$$

$$\mathrm{Ty}\,(\Delta \xrightarrow{\sigma} \Gamma) \colon \mathrm{Ty}\ \Gamma \to \mathrm{Ty}\ \Delta$$
$$\mathrm{Ty}\,(\Delta \xrightarrow{\sigma} \Gamma)\, A \coloneqq A \circ \sigma,$$

with $A \circ \sigma \colon \mathrm{Ty}\ \Delta = \Delta \to \mathrm{Set}$.

- The presheaf $\mathrm{Tm} : (\int \mathrm{Ty})^{\mathrm{op}} \to \underline{\mathbf{Set}}$ is defined on objects as

$$\mathrm{Tm}\,(\Gamma, A) : \mathrm{Set}$$
$$\mathrm{Tm}\,(\Gamma, A) \coloneqq \prod_{\gamma:\Gamma} A\,\gamma.$$

  Before we define Tm on morphisms, recall the objects and morphisms of the category $(\int \mathrm{Ty})^{\mathrm{op}}$:
  - Objects are pairs $(\Gamma : \mathrm{Set}, \mathrm{Ty}\ \Gamma)$.
  - A morphism $(\Gamma, A) \to (\Delta, B)$ is a function $f : \Delta \to \Gamma$ and a proof $p : \mathrm{Ty}\ f\,A \equiv B$, and by the definition of Ty on morphisms, this reduces to $p : A \circ f \equiv B$.

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

$$\mathrm{Tm}\,((\Gamma, A) \xrightarrow{(f,p)} (\Delta, B)) : \mathrm{Tm}\,(\Gamma, A) \to \mathrm{Tm}\,(\Delta, B)$$
$$\mathrm{Tm}\,((\Gamma, A) \xrightarrow{(f,p)} (\Delta, B))\,a \coloneqq \lambda\,\delta.a\,(f\,\delta),$$

  where $a : \prod_{\gamma:\Gamma} A\,\gamma$, and $\delta : \Delta$. Then $a\,(f\,\delta)$ is of type $A\,(f\,\delta) = (A \circ f)\,\delta \equiv B\,\delta$ by p, as required.

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

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

  with $a\,(f\,\delta) : (A \circ f)\,\delta$ as required.
- The context comprehension operation associates to a $\Gamma : \mathrm{Set}$ and an $A : \mathrm{Ty}\ \Gamma$ the set $\Gamma.A$ of pairs $(\rho : \Gamma, u : A\,\rho)$. Then $\mathbf{p} : \Gamma.A \to \Gamma$ is defined by the first projection out of $\Gamma.A$, and $\mathbf{q} : \mathrm{Tm}\,(\Gamma.A, A \circ \mathbf{p})$ by the second projection. ◊

The problem with Example 3.2.3 when not assuming UIP is that Ty Γ, defined as the collection of families of h-sets over Γ, is no longer an h-set, which it is required to be in order to fit within Definition 3.2.1. For the collection $\Gamma \to \mathrm{Set}$ to be an h-set, the universe of h-sets Set needs to be an h-set. If we do not assume univalence nor UIP, whether Set is an h-set is undecided. When working in HoTT and assuming univalence, Set is provably a groupoid and therefore not an h-set [Uni13, Theorem 7.1.11].

If we want to present the set model of type theory in a setting that is compatible with HoTT, we cannot assume UIP. There are two directions we can take to get around the absence of UIP.

- **Approach A**: We coerce the set model to fit into Definition 3.2.1 by changing the definition of Ty Γ, for example by use of an encoding, so that Ty Γ becomes an h-set and we eliminate the issue.
- **Approach B**: We accept the higher h-level of Ty Γ as a groupoid, augment Definition 3.2.1 with some extra coherence laws, prove these coherence laws, and present the set model as an instance of this new structure.

The fundamental underlying issue with defining CwFs as in Definition 3.2.1 in HoTT is that 1-categories are not a suitable formalism to use to talk about a theory where equalities are *data* and not *propositions*, i.e. where we have proof relevance. Instead, ∞-categories do a much better job in this setting, since equality in such a theory behaves more like an isomorphism in an ∞-category than a law in a 1-category. This viewpoint motivates the definition of a model of type theory involving an ∞-category instead of a 1-category, which is referred to in the literature as ∞-CwFs. Kraus develops a notion of an ∞-CwF in two-level type theory [Kra21]. Uemura uses a different notion of model, namely natural models [Awo18], and shows that the initial ∞-natural model is an h-set, although the proof is not done in type theory [Uem22].

In this thesis, we are concerned with formulating a container model of type theory, and we do this by taking **Approach B** outlined above. For our purposes, since the container model involves types that are at most groupoids, we need not consider ∞-CwFs, but rather we can simply go one h-level higher than CwFs and consider *groupoid* CwFs. We work in the setting of HoTT and assume univalence.

It is also possible to take **Approach A** and present a version of the container model that fits into Definition 3.2.1, without assuming neither UIP nor univalence. This would involve encoding containers via an inductive-recursive universe $\mathsf{U} : \mathsf{Type}, \mathsf{El} : \mathsf{U} \to \mathsf{Type}$, and defining types as *codes* for containers (elements of U) so that their collection forms an h-set. This takes care of the coherence issue explained above, and brings the h-level of $\mathsf{Ty}\ \Gamma$ down to being an h-set. However, in the absence of univalence, another issue crops up whereby we can only prove the functor laws of Ty up to isomorphism and not equality. To circumvent this, we define types as also being augmented by a substitution so that type substitution is 'delayed', which lets us prove the functor laws for Ty strictly without making use of univalence. We do not focus on this approach here and leave writing out the details of this for future work.

3.3 Groupoid Categories with Families

In this section, we explain in detail the structure and coherence laws that need to be added to a CwF for it to accomodate for groupoids of types, which we call a groupoid CwF (GCwF), and then we define the GCwF of containers.

There has been some recent work done on models of type theory that allow for groupoids of types. Chen studies CwFs whereby the collection of substitutions do not have to form an h-set (i.e. the base category need not have hom-*sets* but it can have any hom-type) [Che25]. In particular, our GCwF definition (Definition 3.3.3) is a simple example of what Chen calls '2-coherent wild CwFs' (and we also borrow the terminology 'type structure' and 'term structure' from this preprint). Altenkirch et al. define GCwFs and show that the initial GCwF (with Π-types and a base family) is set-truncated [AKX26]. Our own GCwF definition (Definition 3.3.3) should be equivalent to that given by Altenkirch et al., and the reason we define our own is that their formalised definition in `Agda` involves some encoding techniques we did not need that would have made our container GCwF more opaque. Van der Weide tackles the coherence issues mentioned in the previous section for comprehension categories [vdWei25].

Just like CwFs, our GCwF definition will start with a base category $\underline{\mathbf{C}}$ of contexts and substitutions. For interpreting types, we want to replace the presheaf $\mathrm{Ty} : \underline{\mathbf{C}}^{\mathrm{op}} \to \underline{\mathbf{Set}}$ from CwFs by a presheaf going into the bicategory $\underline{\mathbf{Gpd}}$, whose 0-cells are groupoids, 1-cells are functors, and 2-cells are natural transformations. Since the codomain of Ty is no longer the 1-category $\underline{\mathbf{Set}}$ but is now a bicategory, we need to adjust our notion of presheaf. Namely, we need to take into consideration that the functor laws stating preservation of identity and associativity are no longer propositions (equalities on h-sets) but h-sets (equalities on groupoids). Therefore, these functor laws need to abide by some extra coherence laws that did not need to be mentioned in the original CwF definition, since in that case they were properties and not data.

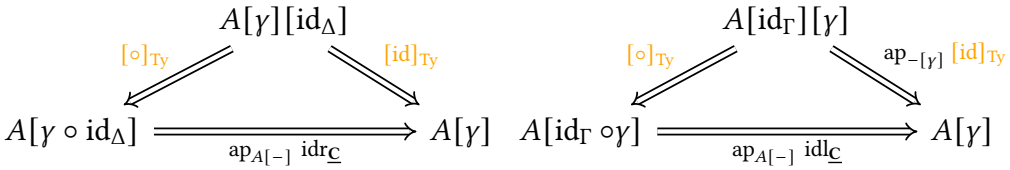
What we need is Ty to be a pseudofunctor in the sense of Ahrens et. al [AFM+21], in the special case where the source category is a 1-category instead of a bicategory. However, in our actual definition(s), we interpret $\underline{\mathbf{Gpd}}$ synthetically, i.e. we mean hGpd, the universe of 1-types. This makes our definition(s) more succinct. We spell out precisely what structure Ty needs to carry in the definition below.

Definition 3.3.1. A *type structure* on a 1-category $\underline{\mathbf{C}}$ consists of:

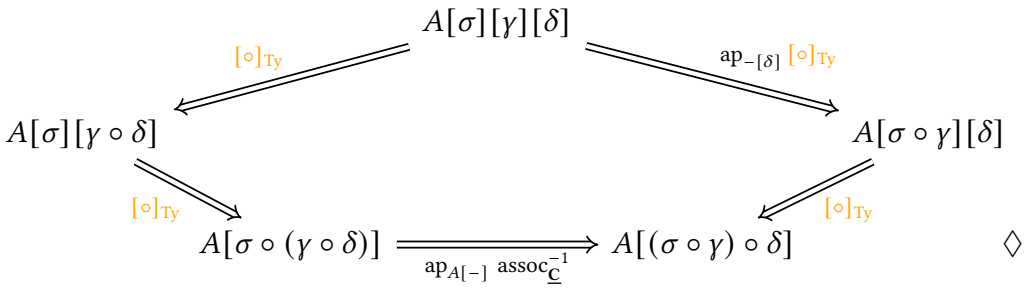
- A map on objects $\mathrm{Ty}_0 : |\underline{\mathbf{C}}| \to \mathrm{hGpd}$.
- A contravariant map on morphisms $\mathrm{Ty}_1 : \underline{\mathbf{C}}(\Gamma, \Delta) \to \mathrm{Ty}_0\ \Delta \to \mathrm{Ty}_0\ \Gamma$.

  In what follows, we use the notation Ty for Ty_0 and $-[\delta]$ for $\mathrm{Ty}_1\ \delta$.

- For an object Γ in $\underline{\mathbf{C}}$ and a groupoid $A : \mathrm{Ty}\ \Gamma$, an equality
$$[\mathrm{id}]_{\mathrm{Ty}} : A[\mathrm{id}_\Gamma] \equiv_{\mathrm{Ty}\ \Gamma} A.$$
- For morphisms $\Delta \xrightarrow{\gamma} \Gamma \xrightarrow{\sigma} \Sigma$ in $\underline{\mathbf{C}}$ and a groupoid $A : \mathrm{Ty}\ \Sigma$, an equality
$$[\circ]_{\mathrm{Ty}} : A[\sigma][\gamma] \equiv_{\mathrm{Ty}\ \Delta} A[\sigma \circ \gamma].$$
- For a morphism $\gamma : \Delta \to \Gamma$ in $\underline{\mathbf{C}}$ and a groupoid $A : \mathrm{Ty}\ \Gamma$, the below commutative diagrams on equalities, which we call $[\mathrm{id}]_{\mathrm{Ty}}$-cohL and $[\mathrm{id}]_{\mathrm{Ty}}$-cohR respectively, known as the **triangulators**:



- For morphisms $\Xi \xrightarrow{\delta} \Delta \xrightarrow{\gamma} \Gamma \xrightarrow{\sigma} \Sigma$ in $\underline{\mathbf{C}}$ and a groupoid $A : \mathrm{Ty}\ \Sigma$, the below commutative diagram on equalities, called $[\circ]_{\mathrm{Ty}}$-coh, known as the **pentagonator**:



Recall from Example 3.2.3 that the coherence issues with the original CwF definition arise when interpreting types. We note that no coherence issues arise when interpreting terms, since the collection of terms in the case of the set model (as well as the presheaf model and, as we will see, the container model) is in fact an h-set. This suggests that we should not have to make any adjustments to the interpretation of terms in our GCwF definition. While we won't have to add any coherence laws for Tm like we did for Ty, we still need to be careful since the definition of Tm depends on Ty and the structure required on Ty has now changed.

The original CwF definition requires a presheaf $\mathrm{Tm} : (\int \mathrm{Ty})^{\mathrm{op}} \to \underline{\mathbf{Set}}$, where $\int \mathrm{Ty}$ is the category of elements over Ty, and Ty is a presheaf. For our GCwF definition, we now require Ty to be a type structure on the base category $\underline{\mathbf{C}}$ as in Definition 3.3.1, which makes the notion of 'a category of elements over Ty' imprecise. This is easily corrected by simply looking at the 'presheaf' part of Ty (i.e. ignoring the triangulators and pentagonator laws in Definition 3.3.1) and considering the category of elements over it. We spell out precisely the structure required on Tm in the definition below.

Definition 3.3.2. A *term structure* on a 1-category $\underline{\mathbf{C}}$ and a type structure Ty consists of:

- A map on objects
$$\mathrm{Tm}_0 : \sum_{\Gamma : |\underline{\mathbf{C}}|} \mathrm{Ty}_0\ \Gamma \to \mathrm{Set}.$$
- For $\Gamma, \Delta : |\underline{\mathbf{C}}|$ and $B : \mathrm{Ty}_0\ \Delta$, a contravariant map on morphisms
$$\mathrm{Tm}_1^B : \prod_{\delta : \underline{\mathbf{C}}(\Gamma,\Delta)} \mathrm{Tm}_0\ (\Delta,\ B) \to \mathrm{Tm}_0\ (\Gamma, \mathrm{Ty}_1\ \delta\ B).$$

  In what follows, we use the notation Tm for Tm_0 and $-[\delta]$ for $\mathrm{Tm}_1\ \delta$, as well as the previously mentioned notation (in Definition 3.3.1) for type structures. As a convention, we use upper case letters to refer to types and lower case letters to refer to terms, so that $B[\delta]$ will refer to $\mathrm{Ty}_1\ \delta\ B$ while $b[\delta]$ will refer to $\mathrm{Tm}_1^B\ \delta\ b$.

- For $\Gamma : |\underline{\mathbf{C}}|$, $A : \mathrm{Ty}\ \Gamma$, and $a : \mathrm{Tm}\ (\Gamma, A)$, an equality
$$[\mathrm{id}]_{\mathrm{Tm}} : a[\mathrm{id}_\Gamma] \approx a.$$

↑ $[\mathrm{id}]_{\mathrm{Tm}}$ is a dependent equality over $[\mathrm{id}]_{\mathrm{Ty}}$

- For $\Delta \xrightarrow{\gamma} \Gamma \xrightarrow{\sigma} \Sigma$ in $\underline{\mathbf{C}}$, $B : \mathrm{Ty}\ \Sigma$, and $b : \mathrm{Tm}\ (\Sigma, B)$, an equality
$$[\circ]_{\mathrm{Tm}} : b[\sigma][\gamma] \approx b[\sigma \circ \gamma].$$

↥ $[\circ]_{\mathrm{Tm}}$ is a dependent equality over $[\circ]_{\mathrm{Ty}}$ ◊

Note that the equalities $[\mathrm{id}]_{\mathrm{Tm}}$ and $[\circ]_{\mathrm{Tm}}$ in Definition 3.3.2 are dependent on the equalities $[\mathrm{id}]_{\mathrm{Ty}}$ and $[\circ]_{\mathrm{Ty}}$ from Definition 3.3.1 respectively, hence why we use the notation $\approx$ instead of $\equiv$.

Lastly, our GCwF definition requires a context comprehension operation, which remains unchanged from the one in the original CwF definition. We are now ready to give the definition of a GCwF.

Definition 3.3.3. A *groupoid category with families* (GCwF) consists of:

- A category $\underline{\mathbf{C}}$ whose objects interpret contexts and whose morphisms interpret context substitutions, having a terminal object interpreting the empty context.
- A type structure over $\underline{\mathbf{C}}$
$$\mathrm{Ty} : \underline{\mathbf{C}}^{\mathrm{op}} \to \underline{\mathbf{Gpd}}$$
whose object map interprets types and whose morphism map interprets type substitution.
- A term structure over $\underline{\mathbf{C}}$ and Ty
$$\mathrm{Tm} : (\textstyle\int \mathrm{Ty})^{\mathrm{op}} \to \underline{\mathbf{Set}}$$
whose object part interprets terms and whose morphism part interprets term substitution.
- A context comprehension structure consisting of
  - a context extension operation
$$\_.\_ : \prod_{\Gamma : |\underline{\mathbf{C}}|} \mathrm{Ty}\ \Gamma \to |\underline{\mathbf{C}}|$$
  - for $\Gamma : |\underline{\mathbf{C}}|$ and $A : \mathrm{Ty}\ \Gamma$, a substitution $\mathbf{p}_A : \underline{\mathbf{C}}(\Gamma.A, \Gamma)$
  - for $\Gamma : |\underline{\mathbf{C}}|$ and $A : \mathrm{Ty}\ \Gamma$, a term $\mathbf{q}_A : \mathrm{Tm}\ (\Gamma.A, A[\mathbf{p}])$
  - for $\Gamma, \Delta : |\underline{\mathbf{C}}|$ and $A : \mathrm{Ty}\ \Gamma$, an operation
$$\langle \_, \_ \rangle : \prod_{\gamma : \underline{\mathbf{C}}(\Delta,\Gamma)} \mathrm{Tm}\ (\Delta, A[\gamma]) \to \underline{\mathbf{C}}(\Delta, \Gamma.A)$$

  such that we have the following equalities.

$$* \ \mathbf{p}\beta : \mathbf{p}_A \circ \langle \gamma, a \rangle \equiv \gamma \qquad \forall \Delta \xrightarrow{\gamma} \Gamma \text{ in } \underline{\mathbf{C}}, A : \mathrm{Ty}\ \Gamma, a : \mathrm{Tm}\ (\Delta, A[\gamma])$$

$$* \ \mathbf{q}\beta : \mathbf{q}_A[\langle \gamma, a \rangle] \approx a \qquad \forall \Delta \xrightarrow{\gamma} \Gamma \text{ in } \underline{\mathbf{C}}, A : \mathrm{Ty}\ \Gamma, a : \mathrm{Tm}\ (\Delta, A[\gamma])$$

$$* \ \langle \eta \rangle : \langle \mathbf{p}_A \circ \theta, \mathbf{q}_A[\theta] \rangle \equiv \theta \qquad \forall A : \mathrm{Ty}\ \Gamma, \Delta \xrightarrow{\theta} \Gamma.A$$

$$* \ \langle \circ \rangle : \langle \sigma, a \rangle \circ \gamma \approx \langle \sigma \circ \gamma, a[\gamma] \rangle \qquad \forall \Delta \xrightarrow{\gamma} \Gamma \xrightarrow{\sigma} \Sigma \text{ in } \underline{\mathbf{C}}, A : \mathrm{Ty}\ \Sigma, a : \mathrm{Tm}\ (\Gamma, A[\sigma])$$

↥ $\mathbf{q}\beta$ is a dependent equality over $[\circ]_{\mathrm{Ty}}$ and $\mathbf{p}\beta$

↥ $\langle \circ \rangle$ is a dependent equality over $[\circ]_{\mathrm{Ty}}$ ◊

Intuitively, the equalities $\mathbf{p}\beta$, $\mathbf{q}\beta$, $\langle\eta\rangle$, and $\langle\circ\rangle$ in Definition 3.3.3 ensure that for $\Gamma : |\underline{\mathbf{C}}|$ and $A : \mathrm{Ty}\ \Gamma$, $\Gamma.A$ is a representation of the presheaf

$$\sum_{\gamma\,:\,\underline{\mathbf{C}}(-,\Gamma)} \mathrm{Tm}\,(-, A[\gamma]) : \underline{\mathbf{C}}^{\mathrm{op}} \to \underline{\mathbf{Set}}.$$

3.4 The GCwF of Containers

We now present an instance of the above definition: the GCwF of containers, based on the container model presented in an abstract by Altenkirch and Kaposi [AK21]. All the constructions shown in this section, as well as Definition 3.3.3, are fully formalised in `Cubical Agda` and can be found at the link in [Dam25]. Our formalisation is based on the one accompanying Altenkirch and Kaposi's abstract [AK21]. We used their (incomplete) formalisation as a starting point for ours. While their formalisation included the main components of the container model, like the definitions of contexts, types, terms, and some of the associated proofs, many other proofs were omitted. In particular, the coherence laws required for a GCwF were not stated nor proved, and proofs relating to terms and context comprehension were incomplete or omitted. We remedy this by completing all the necessary constructions and proofs, which required us to set things up differently than they did to facilitate later proofs.

3.4.1 Contexts and substitutions

We define the category of contexts and substitutions to be $\underline{\mathbf{Cont}}$, as defined in Definition 1.3.9, where contexts are containers

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

and substitutions are container morphisms.

```
record Sub (Γ Δ : SetCon) : Type where
  constructor _◁_
  field
    s : Γ .S → Δ .S
    p : {sΓ : Γ .S} → Δ .P (s sΓ) → Γ .P sΓ
```

Note: We use slightly different notation here for container morphisms than that introduced in Definition 1.3.9. Namely, we now call the shape map `s` instead of u and the position map `p` instead of f, and the first argument of the position map is now implicit.

In this chapter we will refer to $\underline{\mathbf{Cont}}$ as the category of *set*-containers to differentiate them from *generalised* containers. Recall that every set-container $S \triangleleft P$ has an associated functor $[\![S \triangleleft P]\!] : \underline{\mathbf{Set}} \to \underline{\mathbf{Set}}$, as defined in Definition 1.3.5.

We also recall that associativity of composition ass and the left and right identity laws idl and idr hold by definition, i.e. they are provable by refl.

Cont has a terminal object $\top \triangleleft (\lambda \_ \to \top)$.

3.4.2 Types and type substitution

The type structure $\mathrm{Ty} : \underline{\mathbf{Cont}}^{\mathrm{op}} \to \underline{\mathbf{Gpd}}$ is defined as follows.

- (Objects) For a set-container Γ, $\mathrm{Ty}\ \Gamma$ is the groupoid of generalised containers over the category of elements over $[\![\Gamma]\!]$, denoted $\int [\![\Gamma]\!]$, as defined in Definition 1.5.1. If $A : \mathrm{Ty}\ \Gamma$, then $A = S_A \triangleleft_{\int [\![\Gamma]\!]} P_A$ with $S_A : \mathrm{Set}$, $P_A : S_A \to \left| \int [\![\Gamma]\!] \right|$, where we break apart the latter into 3 components as shown below.

$$
\begin{aligned}
&S_A : \mathrm{Set} \\
&P_A^X : S_A \to \mathrm{Set} \\
&P_A^s : S_A \to S_\Gamma \\
&P_A^f : \prod_{s : S_A} P_\Gamma\ (P_A^s\ s) \to P_A^X\ s
\end{aligned}
$$

  In our formalisation, we represent the above 4 components as the 3 components S, P, and Q below. These distinct formulations of a generalised container are equivalent; the above is the fibred way while the below is the indexed way of expressing the same data.[2]

```
record GenCon (Γ : SetCon) : Type₁ where
  no-eta-equality
  field
    S : Γ .S → Type
    P : {sΓ : Γ .S} → S sΓ → Type
    Q : {sΓ : Γ .S} (s : S sΓ) → Γ .P sΓ → P s
    isSetS : {sΓ : Γ .S} → isSet (S sΓ)
    isSetP : {sΓ : Γ .S} {s : S sΓ} → isSet (P s)
```

  So the object part of Ty is defined as GenCon, and if $A : \mathrm{Ty}\ \Gamma$, then by definition $A : \mathrm{GenCon}\ \Gamma$.
- (Morphisms) Given a substitution $\gamma : \Delta \to \Gamma$ and a type $A : \mathrm{Ty}\ \Gamma$, we define $A[\gamma] : \mathrm{Ty}\ \Delta$ as $A[\gamma] \coloneqq S_{A[\gamma]} \triangleleft_{\int [\![\Delta]\!]} P_{A[\gamma]}$, where

$$
\begin{aligned}
&S_{A[\gamma]} : \mathrm{Set} \\
&P_{A[\gamma]}^X : S_{A[\gamma]} \to \mathrm{Set} \\
&P_{A[\gamma]}^s : S_{A[\gamma]} \to S_\Delta
\end{aligned}
$$

[2] To see precisely how they are equivalent: given an indexed generalised container S, P, Q, we define $S_A \coloneqq \sum_{s\Gamma : S_\Gamma} S\ s\Gamma$, $P_A^X\ (s\Gamma, s) \coloneqq P\ \{s\Gamma\}\ s$, $P_A^s\ (s\Gamma, s) \coloneqq s\Gamma$, and $P_A^f\ (s\Gamma, s)\ p \coloneqq Q\ \{s\Gamma\}\ s\ p$. Given a fibred generalised container S_A, P_A^X, P_A^s, P_A^f, we set $S\ s\Gamma \coloneqq \sum_{sA : S_A} (P_A^s\ sA \equiv s\Gamma)$, $P\ \{s\Gamma\}\ (sA, e) \coloneqq P_A^X\ sA$, and $Q\ \{s\Gamma\}\ (sA, e)\ p\Gamma \coloneqq P_A^f\ sA\ p\Gamma$, with the last definition typechecking up to e.

$$P^f_{A[\gamma]} : \prod_{s : S_{A[\gamma]}} P_\Delta\,(P^s_{A[\gamma]}\,s) \to P^X_{A[\gamma]}\,s.$$

$S_{A[\gamma]}$ is defined as the below pullback.

$$\begin{array}{ccc} S_{A[\gamma]} & \xrightarrow{s_A} & S_A \\ {\scriptstyle s_\Delta}\downarrow & \lrcorner & \downarrow{\scriptstyle P^s_A} \\ S_\Delta & \xrightarrow[\gamma_s]{} & S_\Gamma \end{array}$$

$P^X_{A[\gamma]}$ is defined roughly[3] as the below pushout in h-sets (or set-pushout), for a fixed $s : S_{A[\gamma]}$ with projections s_Δ and s_A.

$$\begin{array}{ccc} P_\Gamma\,(\gamma_s\,(s_\Delta\,s)) & \xrightarrow{P^f_A\,s_A} & P^X_A\,(s_A\,s) \\ {\scriptstyle \gamma_p\,(s_\Delta\,s)}\downarrow & \ulcorner & \downarrow{\scriptstyle \mathrm{inr}} \\ P_\Delta\,(s_\Delta\,s) & \xrightarrow[\mathrm{inl}]{} & P^X_{A[\gamma]}\,s \end{array}$$

The other components can then be read off from these squares: $P^s_{A[\gamma]}$ is defined as s_Δ, and $P^f_{A[\gamma]}$ is defined as $\lambda\_ \to \mathrm{inl}$.
In our formalisation, we adapt the above (written in the fibred setting) to the indexed setting. By going through the equivalence between the two, we obtain the below components for $A[\gamma]$.

$$\begin{aligned} \mathsf{S} &: \Delta\,.\mathsf{S} \to \mathsf{Type} \\ \mathsf{S}\,s\Delta &\coloneqq A\,.\mathsf{S}\,(\gamma\,.\mathsf{s}\,s\Delta) \end{aligned}$$

The family S completely encapsulates the pullback square of $S_{A[\gamma]}$. This is the main advantage of switching to the indexed setting: we are able to represent pullbacks as families instead of as a Σ-type, and by doing so, we get many more strict equalities which significantly simplify later proofs.

$$\begin{aligned} \mathsf{P} &: \{s\Delta : \Delta\,.\mathsf{S}\} \to \mathsf{S}\,s\Delta \to \mathsf{Type} \\ \mathsf{P}\,s &\coloneqq \mathsf{SetPushout}(\gamma\,.\mathsf{p})(A\,.\mathsf{Q}\,s) \end{aligned}$$

The P component is a pushout in h-sets. Pictorially, it is the below pushout square.

$$\begin{array}{ccc} \Gamma\,.\mathsf{P}\,(\gamma\,.\mathsf{s}\,s\Delta) & \xrightarrow{A\,.\mathsf{Q}\,s} & A\,.\mathsf{P}\,s \\ {\scriptstyle \gamma\,.\mathsf{p}\,\{s\Delta\}}\downarrow & & \downarrow \\ \Delta\,.\mathsf{P}\,s\Delta & \longrightarrow & * \end{array}$$

It is a well-known fact that pushouts of h-sets need not be h-sets. In this case, we want to take the pushout in **Set**, which is equivalent to taking the pushout and then set-truncating it. Indeed, SetPushout is defined as

[3] This square technically does not typecheck, since $P^f_A\,s_A$ does not have the correct type. It only typechecks up to the commutativity of the pullback square defining $S_{A[\gamma]}$.

```
SetPushout : {A : Type ℓ} {B : Type ℓ'} {C : Type ℓ''}
             (f : A → B) (g : A → C) → Type _
SetPushout f g = || Pushout f g ||₂
```

where Pushout is defined in the `Cubical Agda` standard library as the following HIT

```
data Pushout {A : Type ℓ} {B : Type ℓ'} {C : Type ℓ''}
             (f : A → B) (g : A → C) : Type (ℓ-max ℓ (ℓ-max ℓ' ℓ''))
             where
  inl : B → Pushout f g
  inr : C → Pushout f g
  push : (a : A) → inl (f a) ≡ inr (g a)
```

and set-truncation $||\_||_2$ is also defined in the standard library as a QIT.

```
data ||_||₂ {ℓ} (A : Type ℓ) : Type ℓ where
  |_|₂ : A → || A ||₂
  squash₂ : ∀ (x y : || A ||₂) (p q : x ≡ y) → p ≡ q
```

We also define an elimination principle for SetPushout into an h-set which will be useful later on.

```
elimSetPushout : {A : Type ℓ} {B : Type ℓ'} {C : Type ℓ''}
                 (f : A → B) (g : A → C)
                 (X : SetPushout f g → Type ℓ''') →
                 ((p : SetPushout f g) → isSet (X p)) →
                 (inl' : (b : B) → X (| inl b |₂)) →
                 (inr' : (c : C) → X (| inr c |₂)) →
                 (push' : (a : A) → PathP (λ i → X (| push a i |₂))
                                          (inl' (f a)) (inr' (g a))) →
                 (x : SetPushout f g) → X x
```

For the most part the eliminator is defined in the 'usual way', i.e. it calls the relevant function for each constructor of SetPushout, except for the case of squash_2, for which we note that elimSetPushout does not require an input. In this case we use the fact that we eliminate into an h-set to show that the predicate X holds on this constructor.

```
Q : {sΔ : Δ .S}(s : A .S sΔ) → Δ .P sΔ → P s
Q s pΔ ≔ | inl pΔ |₂
```

The Q component is an element of P, which by the indexed-fibred translation is defined as the $P^{f}_{A[\gamma]}$ component.

3.4.3 Ty preserves identity and composition

Prerequisites

In order to show that Ty preserves identity ($[\mathsf{id}]_{\mathrm{Ty}}$) and composition ($[\circ]_{\mathrm{Ty}}$), we need to establish some groundwork for GenCons first. We define a univalence principle specifically for the type GenCon, which we call uaGenCon.

```
uaGenCon : ∀ {Γ} {A B : GenCon Γ} → (A ≃GenCon B) → A ≡ B
```

uaGenCon takes an equivalence on GenCons and turns it into an equality. An equivalence between two generalised containers, that are both in some context Γ, consists of an equivalence s≃ between their S components, an equivalence p≃ of their P components up to s≃, and an equality q≡ between their Q components up to p≃. That the isSetS and isSetP components agree follows automatically from the fact that for any type X, isSet X is a proposition, so we don't require any proofs for these components.

```
record _≃GenCon_ {Γ} (A B : GenCon Γ) : Type where
  field
    s≃ : (sΓ : Γ .S) → A .S sΓ ≃ B .S sΓ
    p≃ : (s : Σ[ sΓ ∈ Γ .S ] (A .S sΓ)) →
         A .P {fst s} (snd s) ≃ B .P {fst s} (fst (s≃ (fst s)) (snd s))
    q≡ : (sΓ : Γ .S) (sA : A .S sΓ) (pΓ : Γ .P sΓ) →
         fst (p≃ _) (A .Q {sΓ} sA pΓ) ≡ B .Q {sΓ} _ pΓ
```

To prove uaGenCon, we new a few lemmas. First, we need reflexivity of _≃GenCon_.

Lemma 3.4.1 (id≃GenCon)**.** *For any* A : GenCon Γ, $A \simeq_{\mathsf{GenCon}} A$.

Proof. The s≃ and p≃ components are just the identity equivalence, and q≡ is refl. □

Next, we need to define an eliminator of equivalences on GenCons.

Lemma 3.4.2 (J≃GenCon)**.** *Fix an* A : GenCon Γ. *If a motive*

$$M : (B : \mathsf{GenCon}\ \Gamma) \to A \simeq_{\mathsf{GenCon}} B \to \mathsf{Type}\ \_$$

holds on A *and* id≃GenCon A, *then it holds for all* B : GenCon Γ *and* $eq : A \simeq_{\mathsf{GenCon}} B$.

Proof. We prove this using the usual path induction J, combined with an eliminator of families of equivalences:

```
famEquivJ : {C : Type ℓ''} {B : C → Type ℓ}
            (P : (A : C → Type ℓ) (e : (c : C) → A c ≃ B c) → Type ℓ') →
            P B (λ c → idEquiv (B c)) →
            {A : C → Type ℓ} (e : (c : C) → A c ≃ B c) → P A e
```

The proof of famEquivJ itself does not involve GenCons and is a general theorem on equivalences, so we omit the proof here and refer the interested reader to our formalisation. □

We can now prove uaGenCon.

Proof of uaGenCon. We make use of the fundamental theorem of identity types [Rij25], which states that if a relation R on a type A is

1. reflexive: $\forall x : A, R\,x\,x$, and

2. $\forall x : A, \sum_{y:A} R\, x\, y$ is contractible,

then $\forall x, y : A, (x \equiv y) \simeq (R\, x\, y)$.

If we set R to $\_\simeq\mathsf{GenCon}\_$, property (1) reduces to $\mathsf{id}\simeq\mathsf{GenCon}$. We can show property (2) for any $A : \mathsf{GenCon}\,\Gamma$ by setting the centre of contraction to $(A, \mathsf{id}\simeq\mathsf{GenCon}\,A)$, and then using $\mathsf{J}\simeq\mathsf{GenCon}$ with

$$M \coloneqq \lambda\, B\, e \to (A, \mathsf{id}\simeq\mathsf{GenCon}\,A) \equiv (B, e)$$

to show its uniqueness.

Now we have that $\forall A, B : \mathsf{GenCon}\,\Gamma, (A \equiv B) \simeq (A \simeq\mathsf{GenCon}\, B)$. Therefore to obtain an equality between A and B, we simply take the inverse direction of this equivalence. □

Proofs of $[\mathsf{id}]_{\mathsf{Ty}}$ and $[\circ]_{\mathsf{Ty}}$

We are now ready to prove $[\mathsf{id}]_{\mathsf{Ty}}$ and $[\circ]_{\mathsf{Ty}}$. We note that these proofs are not merely properties but data, since we will be proving coherence laws about them later on.

($[\mathsf{id}]_{\mathsf{Ty}}$) We show

$$[\mathsf{id}]_{\mathsf{Ty}} : \forall\{\Gamma\}\,(A : \mathsf{Ty}\ \Gamma) \to A[\mathrm{id}_\Gamma] \equiv A.$$

We prove $[\mathsf{id}]_{\mathsf{Ty}}$ as

$$[\mathsf{id}]_{\mathsf{Ty}}\, A \coloneqq \mathsf{uaGenCon}\,([\mathsf{id}]\mathsf{T\text{-}eq}\, A)$$

so that we can focus on constructing an equivalence

$$[\mathsf{id}]\mathsf{T\text{-}eq} : \forall\{\Gamma\}\, A \to A[\mathrm{id}_\Gamma] \simeq\mathsf{GenCon}\, A,$$

and $\mathsf{uaGenCon}$ will turn it into an equality.

We now construct the $\mathsf{s}\simeq$, $\mathsf{p}\simeq$, and $\mathsf{q}\equiv$ components of $[\mathsf{id}]\mathsf{T\text{-}eq}$. Firstly, for $s\Gamma : \Gamma\,.\mathsf{S}$, $A[\mathrm{id}_\Gamma]\,.\mathsf{S}\; s\Gamma = A\,.\mathsf{S}\,(\mathrm{id}_\Gamma\;.\mathsf{s}\; s\Gamma) = A\,.\mathsf{S}\; s\Gamma$ by definition of $A[-]$ and id_Γ, so

$$\mathsf{s}\simeq : A[\mathrm{id}_\Gamma]\,.\mathsf{S}\; s\Gamma \simeq A\,.\mathsf{S}\; s\Gamma$$

can be defined as the identity equivalence. Secondly, we show that for $s\Gamma : \Gamma\,.\mathsf{S}$ and $sA : A\,.\mathsf{S}\; s\Gamma$,

$$\mathsf{p}\simeq : \mathsf{SetPushout}\ \mathrm{id}\,(A\,.\mathsf{Q}\; sA) \simeq A\,.\mathsf{P}\; sA.$$

Pictorially, what we need to show is that the below pushout is equivalent to $A\,.\mathsf{P}\; sA$.

$$\begin{array}{ccc} \Gamma\,.\mathsf{P}\; s\Gamma & \xrightarrow{A\,.\mathsf{Q}\; sA} & A\,.\mathsf{P}\; sA \\ {\scriptstyle \mathrm{id}}\downarrow & \ulcorner & \downarrow \\ \Gamma\,.\mathsf{P}\; s\Gamma & \longrightarrow & * \end{array}$$

We make use of a lemma stating that in general, if the left morphism of a set-pushout square is id where the top-right corner is an h-set, then the square is equivalent to its top-right corner.

Lemma 3.4.3 (pushoutIdlEq). *For any type A and h-set B, and for any $f : A \to B$,* SetPushout id $f \simeq B$*:*

$$\begin{array}{ccc} A & \xrightarrow{f} & B \\ {\scriptstyle \mathrm{id}}\downarrow & & \downarrow{\scriptstyle |\,\mathsf{inr}\,-\,|_2} \\ A & \xrightarrow[|\,\mathsf{inl}\,-\,|_2]{} & \mathsf{SetPushout}\ \mathrm{id}\ f \end{array} \qquad \simeq \qquad B$$

Proof. We do this by constructing an isomorphism between the two, and then turning this into an equivalence using isoToEquiv. For this isomorphism, we define the below components.

- $\alpha :$ SetPushout id $f \to B$ sends

$$\begin{aligned} |\,\mathsf{inl}\ a\ |_2 &\mapsto f\ a \\ |\,\mathsf{inr}\ b\ |_2 &\mapsto b \\ |\,\mathsf{push}\ a\ |_2 &\mapsto \mathsf{refl} \end{aligned}$$

  via elimSetPushout.
- $\beta : B \to$ SetPushout id f sends $b \mapsto |\,\mathsf{inr}\ b\ |_2$.
- That $\alpha \circ \beta \equiv \mathrm{id}$ holds by refl.
- To prove that $\forall (x :$ SetPushout id $f), \beta\,(\alpha\,x) \equiv x$, we use elimSetPushout. We are eliminating into an h-set since an equality on SetPushouts is a proposition. For the inl and push cases, we call push on different path variables. □

Going back to our p$\simeq$ component, this can now be defined using pushoutIdlEq for $f = A.\mathsf{Q}\,sA$, where B is the h-set $A.\mathsf{P}\,sA$. Lastly we show that for $s\Gamma : \Gamma.\mathsf{S}$, $sA : A.\mathsf{S}\,s\Gamma$, and $p\Gamma : \Gamma.\mathsf{P}\,s\Gamma$,

$$\mathsf{q}\equiv : \alpha\,(|\,\mathsf{inl}\ p\Gamma\ |_2) \equiv A.\mathsf{Q}\,sA\,p\Gamma$$

where α is the function in the proof of Lemma 3.4.3. Since in our proof of p$\simeq$ we set $f \coloneqq A.\mathsf{Q}\,sA$, then $\alpha\,(|\,\mathsf{inl}\ p\Gamma\ |_2) = A.\mathsf{Q}\,sA\,p\Gamma$, and q$\equiv$ holds by refl. This concludes our $[\mathrm{id}]_{\mathrm{Ty}}$ proof.

($[\circ]_{\mathrm{Ty}}$) We show

$$[\circ]_{\mathrm{Ty}} : \forall \{\Gamma\ \Delta\ \Theta\}\ (A : \mathrm{Ty}\ \Theta)\ (\theta : \mathsf{Sub}\ \Delta\ \Theta)\ (\delta : \mathsf{Sub}\ \Gamma\ \Delta) \to A[\theta][\delta] \equiv A[\theta \circ \delta].$$

We prove $[\circ]_{\mathrm{Ty}}$ as

$$[\circ]_{\mathrm{Ty}}\ A\ \theta\ \delta \coloneqq \mathsf{uaGenCon}\ ([\circ]\text{T-eq}\ A\ \theta\ \delta)$$

so that we can focus on constructing an equivalence

$$[\circ]\text{T-eq} : \forall \{\Gamma\ \Delta\ \Theta\}\ A\ \theta\ \delta \to A[\theta][\delta] \simeq_{\mathsf{GenCon}} A[\theta \circ \delta],$$

and uaGenCon will turn it into an equality.

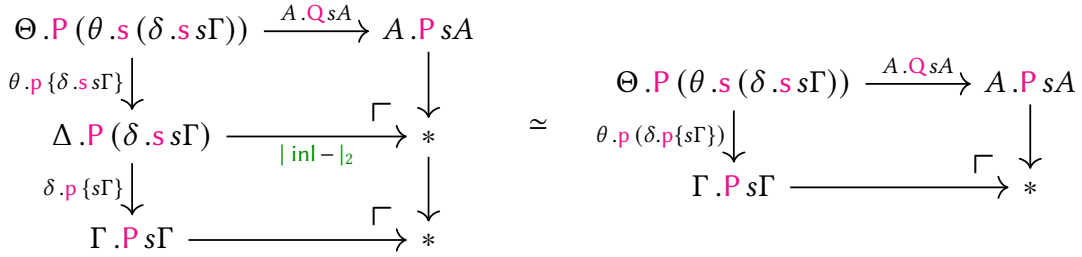
We construct the s$\simeq$, p$\simeq$, and q$\equiv$ components of $[\circ]$T-eq. Firstly, for $s\Gamma : \Gamma.\mathsf{S}$,

$$\mathsf{s}\simeq : (A[\theta])[\delta].\mathsf{S}\,s\Gamma \simeq A[\theta \circ \delta].\mathsf{S}\,s\Gamma,$$

By definition of $-[-]$ and composition of container morphisms, both sides are equal to $A.\mathsf{S}\,(\theta.\mathsf{s}\,(\delta.\mathsf{s}\,s\Gamma))$, and therefore $\mathsf{s}{\simeq}$ can be defined as the identity equivalence. Secondly, for $s\Gamma : \Gamma.\mathsf{S}$ and $sA : A.\mathsf{S}\,(\theta.\mathsf{s}\,(\delta.\mathsf{s}\,s\Gamma))$, we show

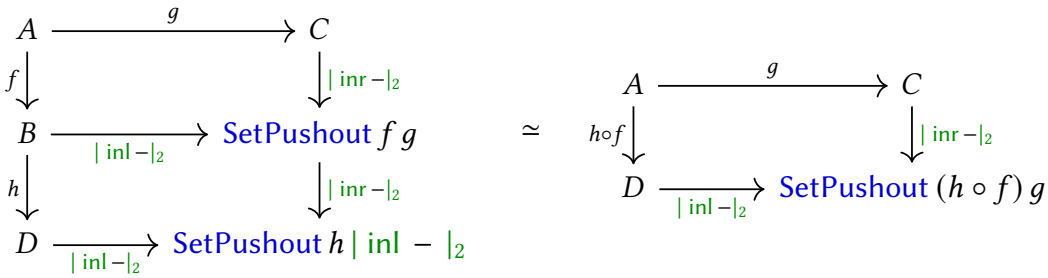
$$\mathsf{p}{\simeq} : \mathsf{SetPushout}\,(\delta.\mathsf{p})\,(|\,\mathsf{inl}\,-\,|_2) \simeq \mathsf{SetPushout}\,((\theta.\mathsf{p}) \circ (\delta.\mathsf{p}))\,(A.\mathsf{Q}\,sA).$$

Pictorially, what we need to show is the below equivalence.



We use a lemma that essentially proves pushout pasting: in general, taking the SetPushout twice is equivalent to taking it once on the composite vertical morphisms.

Lemma 3.4.4 (pushoutCompEq). *For any types A, B, C, D, and any $f : A \to B$, $g : A \to C$, and $h : B \to D$,* $\mathsf{SetPushout}\,\{B\}\,\{D\}\,\{\mathsf{SetPushout}\,f\,g\}\,h\,(|\,\mathsf{inl}\,-\,|_2) \simeq \mathsf{SetPushout}\,(h \circ f)\,g$*:*



Proof. We construct an isomorphism between the two SetPushouts, and then turn this into an equivalence via isoToEquiv. For this isomorphism, we define the below components.

- $\alpha : \mathsf{SetPushout}\,\{B\}\,\{D\}\,\{\mathsf{SetPushout}\,f\,g\}\,h\,(|\,\mathsf{inl}\,-\,|_2) \to \mathsf{SetPushout}\,(h \circ f)\,g$ sends

$$\begin{aligned}
|\,\mathsf{inl}\,d\,|_2 &\mapsto |\,\mathsf{inl}\,d\,|_2 \\
|\,\mathsf{inr}\,(|\,\mathsf{inl}\,b\,|_2)\,|_2 &\mapsto |\,\mathsf{inl}\,(h\,b)\,|_2 \\
|\,\mathsf{inr}\,(|\,\mathsf{inr}\,c\,|_2)\,|_2 &\mapsto |\,\mathsf{inr}\,c\,|_2 \\
|\,\mathsf{inr}\,(|\,\mathsf{push}\,a\,|_2)\,|_2 &\mapsto \lambda\,i \to |\,\mathsf{push}\,(a\,i)\,|_2 \\
|\,\mathsf{push}\,b\,|_2 &\mapsto \mathsf{refl}
\end{aligned}$$

  via elimSetPushout, used once to split on the outer cases, and once to split on the inr cases.

- β : SetPushout $(h \circ f)\, g \to$ SetPushout $\{B\}\, \{D\}\, \{\text{SetPushout}\, f\, g\}\, h\, (|\ \text{inl}\ -\ |_2)$ sends

$$\begin{aligned} |\ \text{inl}\, d\ |_2 &\mapsto |\ \text{inl}\, d\ |_2 \\ |\ \text{inr}\, c\ |_2 &\mapsto |\ \text{inr}\, (|\ \text{inr}\, c\ |_2)\ |_2 \\ |\ \text{push}\, a\ |_2 &\mapsto \lambda\, i \to \text{cmp} \end{aligned}$$

  where cmp is the set-truncation $(|\ \_\ |_2)$ of an hcomp of a partial square whose left side is inl $(h\, (f\, a))$, whose right side is inr $(|\ \text{push}\, a\, j\ |_2)$ (for a path variable j), and whose base is push $(f\, a)\, i$.

To show that $\alpha \circ \beta \equiv \text{id}$ and $\beta \circ \alpha \equiv \text{id}$, we use elimSetPushout in both cases. We omit the proofs here but the full proof of this lemma can be found in our formalisation. □

Thanks to the above lemma, the p≃ component can be defined using pushoutCompEq for $f := \theta.\text{p}$, $g := A.\text{Q}\, sA$, and $h := \delta.\text{p}$. Lastly, we show that for $s\Gamma : \Gamma.\text{S}$, $sA : A.\text{S}\, (\theta.\text{s}\, (\delta.\text{s}\, s\Gamma))$, and $p\Gamma : \Gamma.\text{P}\, s\Gamma$,

$$\text{q}{\equiv} : \alpha\, (|\ \text{inl}\, p\Gamma\ |_2) \equiv |\ \text{inl}\, p\Gamma\ |_2,$$

where α is as defined in Lemma 3.4.4. By definition of α, this is provable by refl. This concludes our $[\circ]_{\text{Ty}}$ proof.

3.4.4 Ty preserves identity and composition coherently

Prerequisites

Before we prove the coherence laws $[\text{id}]_{\text{Ty}}$-cohL, $[\text{id}]_{\text{Ty}}$-cohR, and $[\circ]_{\text{Ty}}$-coh, it is once again helpful to do some preliminary work first. Similarly to how we previously defined uaGenCon to focus on constructing equivalences of GenCons, we now define uaGenConSquare to be able to shift our proof goal from a square of equalities on GenCons to a square of equivalences of GenCons.

```
uaGenConSquare : ∀ {Γ} {A B : GenCon Γ} (b : A ≃GenCon B)
                   {D : GenCon Γ} (r : B ≃GenCon D)
                   {C : GenCon Γ} (l : A ≃GenCon C) (t : C ≃GenCon D) →
                   ≃GenConSquare b t l r →
                   Square (uaGenCon b)
                          (uaGenCon t)
                          (uaGenCon l)
                          (uaGenCon r)
```

uaGenConSquare takes a square of equivalences of GenCons and turns it into a Square of paths via uaGenCon. Therefore, we need to establish what a square of equivalences of GenCons is. We define the record type ≃GenConSquare which takes as input equivalences b, t, l, and r for the bottom, top, left, and right edges of a square respectively. Its definition is broken down below.

```
record ≃GenConSquare {Γ} {A B C D : GenCon Γ}
                     (b : A ≃GenCon B) (t : C ≃GenCon D)
                     (l : A ≃GenCon C) (r : B ≃GenCon D) : Type where
  field
```

For the s-eq component, we look at the first component (or the left-to-right map, called equivFun in `Cubical Agda`) of the .s≃ component of each input, and require that $t \circ l$ is equal to $r \circ b$. This equality is on the type D .S $s\Gamma$.

```
    s-eq : (sΓ : Γ .S) (sA : A .S sΓ) →
           equivFun (t .s≃ sΓ) (equivFun (l .s≃ sΓ) sA) ≡
           equivFun (r .s≃ sΓ) (equivFun (b .s≃ sΓ) sA)
```

For the p-eq component, we look at the first component of the .p≃ component of each input. We once again require that $t \circ l$ is equal to $r \circ b$, but now these two live in different types, so we will need a heterogenous equality between them, i.e. a PathP. This equality is on the type D .P (s-eq $s\Gamma$ sA i) for some path variable i.

```
    p-eq : (sΓ : Γ .S) (sA : A .S sΓ) (pA : A .P sA) →
           PathP (λ i → D .P (s-eq sΓ sA i))
                 (equivFun (t .p≃ (sΓ , equivFun (l .s≃ sΓ) sA))
                 (equivFun (l .p≃ (sΓ , sA)) pA))
                 (equivFun (r .p≃ (sΓ , equivFun (b .s≃ sΓ) sA))
                 (equivFun (b .p≃ (sΓ , sA)) pA))
```

For the q-eq component, we look at the .q≡ component of each input, which in each case is an equality on the .P component of either B, C, or D. We still require that $t \circ l$ is equal to $r \circ b$, but in order to get an equality that typechecks we need a SquareP, with the equality being on the type D .P (s-eq $s\Gamma$ sA i) for some path variables i and j (where j is not used in the type).

```
    q-eq : (sΓ : Γ .S) (sA : A .S sΓ) (pΓ : Γ .P sΓ) →
           SquareP
           (λ i j → D .P (s-eq sΓ sA i))
           (λ k → equivFun (t .p≃ (sΓ , _)) (l .q≡ sΓ sA pΓ k))
           (r .q≡ sΓ (equivFun (b .s≃ sΓ) sA ) pΓ)
           (p-eq sΓ sA (A .Q sA pΓ) ▷
             λ k → equivFun (r .p≃ (sΓ , _)) (b .q≡ sΓ sA pΓ k))
           (t .q≡ sΓ (equivFun (l .s≃ sΓ) sA) pΓ ◁
             (λ i → D .Q (s-eq sΓ sA i) pΓ))
```

Before we prove uaGenConSquare, we prove an important lemma: that uaGenCon on the identity equivalence computes to refl.

Lemma 3.4.5 (uaGenConId). *For any* A : GenCon Γ,

$$\text{uaGenCon}\ (\text{id≃GenCon}\ A) \equiv \text{refl}.$$

Proof. Assume we have $A : \mathsf{GenCon}\,\Gamma$. We once again use the fundamental theorem of identity types we used in the proof of uaGenCon in Section 3.4.3. In that proof, after applying the aforementioned theorem, we obtained

$$e : \forall(A, B : \mathsf{GenCon}\,\Gamma), (A \equiv B) \simeq (A \simeq_{\mathsf{GenCon}} B),$$

where recall that we defined the inverse map of the first component of $e\,A\,B$ to be $\mathsf{uaGenCon}\,A\,B$. Now if we write $e\,A\,A$, we get an equivalence

$$\hat{e} : (A \equiv A) \simeq (A \simeq_{\mathsf{GenCon}} A).$$

Call its first component $\hat{f} : (A \equiv A) \to (A \simeq_{\mathsf{GenCon}} A)$, and call its second component p. p states that $\forall(\epsilon : A \simeq_{\mathsf{GenCon}} A), \sum_{q:A\equiv A}((\hat{f}\,q) \equiv \epsilon)$ is contractible (since in the `Cubical Agda` library, equivalence is defined in terms of uniqueness of fibres). Explicitly,

$$p : \prod_{\epsilon : A \simeq_{\mathsf{GenCon}} A} \sum\Big((q, h) : \sum_{q:A\equiv A} (\hat{f}\,q \equiv \epsilon)\Big) \Big(\prod\Big((q', h') : \sum_{q':A\equiv A} (\hat{f}\,q' \equiv \epsilon)\Big)((q, h) \equiv (q', h'))\Big).$$

Then

$$p\ (\mathsf{id}\simeq_{\mathsf{GenCon}} A)\ .\mathsf{snd}\ (\mathsf{refl}, \mathsf{transportRefl}\ \_) : (q, h) \equiv (\mathsf{refl}, \mathsf{transportRefl}\ \_)$$

and by uaGenCon's definition, $q = \mathsf{uaGenCon}\ (\mathsf{id}\simeq_{\mathsf{GenCon}} A)$. So the first projection of the above gives us that $\mathsf{uaGenCon}\ (\mathsf{id}\simeq_{\mathsf{GenCon}} A) \equiv \mathsf{refl}$. □

We are now ready to prove uaGenConSquare.

Proof of uaGenConSquare. The order in which we stated the arguments for uaGenConSquare should make it clear that we can use J≃GenCon three times, to turn each one of b, r, and l into the identity equivalence in our proof goal. Eventually, we end up having to prove the below.

$$\begin{aligned} aux : {} & (t : A \simeq_{\mathsf{GenCon}} A) \to \\ & \simeq_{\mathsf{GenConSquare}}\ (\mathsf{id}\simeq_{\mathsf{GenCon}} A)\ t\ (\mathsf{id}\simeq_{\mathsf{GenCon}} A)\ (\mathsf{id}\simeq_{\mathsf{GenCon}} A) \to \\ & \mathsf{Square} \\ & \quad (\mathsf{uaGenCon}\ (\mathsf{id}\simeq_{\mathsf{GenCon}} A)) \\ & \quad (\mathsf{uaGenCon}\ t) \\ & \quad (\mathsf{uaGenCon}\ (\mathsf{id}\simeq_{\mathsf{GenCon}} A)) \\ & \quad (\mathsf{uaGenCon}\ (\mathsf{id}\simeq_{\mathsf{GenCon}} A)) \end{aligned}$$

To prove this, first we show $e : (\mathsf{id}\simeq_{\mathsf{GenCon}} A) \equiv t$, by using the components of the ≃GenConSquare we get as input. Then, by definition of Square we have that

$$\mathsf{cong}\ \mathsf{uaGenCon}\ e : \mathsf{Square}$$

(uaGenCon (id≃GenCon A))
(uaGenCon t)
refl
refl

and we also know that

$$\mathsf{sym}\ (\mathsf{uaGenConId}\ A) : \mathsf{refl} \equiv \mathsf{uaGenCon}\ (\mathsf{id}{\simeq}\mathsf{GenCon}\ A)$$

so by subst the theorem holds. □

We will need another lemma stating that if two GenCons are equivalent, they remain equivalent after the same type substitution is applied to both.

Lemma 3.4.6 (cong[_]). *Given a substitution* $\gamma : \Delta \to \Gamma$,

$$\forall\{A, B : \mathsf{GenCon}\ \Gamma\} \to A \simeq\mathsf{GenCon}\ B \to A[\gamma] \simeq\mathsf{GenCon}\ B[\gamma].$$

Proof. Assume A, B, and $e : A \simeq\mathsf{GenCon}\ B$. We construct the required ≃GenCon as follows.

- For the s≃ component, given $s\Delta : \Delta\ .\mathsf{S}$, we need an equivalence $A\ .\mathsf{S}\ (\gamma\ .\mathsf{s}\ s\Delta) \simeq B\ .\mathsf{S}\ (\gamma\ .\mathsf{s}\ s\Delta)$. We have such an equivalence: $e\ .\mathsf{s}{\simeq}\ (\gamma\ .\mathsf{s}\ s\Delta)$.
- For the p≃ component, given $s\Delta : \Delta\ .\mathsf{S}$ and $sA : A\ .\mathsf{S}\ (\gamma\ .\mathsf{s}\ s\Delta)$, we need an equivalence

  $$\mathsf{setPushout}\ (\gamma\ .\mathsf{p})\ (A\ .\mathsf{Q}\ sA) \simeq \mathsf{setPushout}\ (\gamma\ .\mathsf{p})\ (B\ .\mathsf{Q}\ (e\ .\mathsf{s}{\simeq}\ (\gamma\ .\mathsf{s}\ s\Delta)\ .\mathsf{fst}\ sA)).$$

  We know that $A\ .\mathsf{Q}$ and $B\ .\mathsf{Q}$ are equal up to $(e\ .\mathsf{p}{\simeq})$ by $(e\ .\mathsf{q}{\equiv})$, so this equivalence can be obtained by constructing an isomorphism between the setPushouts above using $(e\ .\mathsf{p}{\simeq})$ and $(e\ .\mathsf{q}{\equiv})$, and then applying isoToEquiv at the end.
- The q≡ component involves proving $|\ \mathsf{inl}\ p\Delta\ |_2 \equiv |\ \mathsf{inl}\ p\Delta\ |_2$ for some $p\Delta$, which is provable by refl. □

We also prove that cong[_] acts in the expected ways: cong[γ] on the identity equivalence is the identity equivalence (Lemma 3.4.7), and applying uaGenCon to an equivalence produced by cong[_] is the same as applying cong to an equivalence produced by uaGenCon (Lemma 3.4.8).

Lemma 3.4.7 (cong[_]Id).

$$\mathsf{cong}[\gamma]\ (\mathsf{id}{\simeq}\mathsf{GenCon}\ A) \equiv \mathsf{id}{\simeq}\mathsf{GenCon}\ (A[\gamma]).$$

Proof. We introduce a path variable i and construct a GenCon equivalence of type $A[\gamma] \simeq\mathsf{GenCon}\ A[\gamma]$ as follows.

- For the s≃ component, given $s\Delta : \Delta\ .\mathsf{S}$, we need an equivalence $A\ .\mathsf{S}\ (\gamma\ .\mathsf{s}\ s\Delta) \simeq\mathsf{GenCon}\ A\ .\mathsf{S}\ (\gamma\ .\mathsf{s}\ s\Delta)$ whose boundary is idEquiv _ on both sides, so we construct it using idEquiv.

- For the p≃ component, given $s\Delta : \Delta.S$ and $sA : A.S\,(\gamma.s\,s\Delta)$, we need an equivalence $(\mathsf{SetPushout}\,(\gamma.p)\,(A.Q\,sA)) \simeq_{\mathsf{GenCon}} (\mathsf{SetPushout}\,(\gamma.p)\,(A.Q\,sA))$. Its boundary on the left hand side is $\mathsf{isoToEquiv}\,x$, where x is an isomorphism of SetPushouts constructed as described in the p≃ component of Lemma 3.4.6, and on the right hand side is $\mathsf{idEquiv}\,(\mathsf{SetPushout}\,(\gamma.p)\,(A.Q\,sA))$. To show two equivalences are equal, we only need to show that their first components (or equivFuns) are equal (since being an equivalence is a proposition), and we do so by using elimSetPushout.
- For the q≡ component, we need to show $|\,\mathsf{inl}\,p\Delta\,|_2 \equiv |\,\mathsf{inl}\,p\Delta\,|_2$ for some $p\Delta$, with both boundaries being $\lambda\,i \to |\,\mathsf{inl}\,p\Delta\,|_2$. This is provable by refl. □

Lemma 3.4.8. *For any* $A, B : \mathsf{GenCon}\,\Gamma$, $\gamma : \Delta \to \Gamma$, *and* $e : A \simeq_{\mathsf{GenCon}} B$,

$$\mathsf{cong}\,(\lambda\,A \to A[\gamma])\,(\mathsf{uaGenCon}\,e) \equiv \mathsf{uaGenCon}\,(\mathsf{cong}[\gamma]\,e).$$

Proof. We use $\mathsf{J}{\simeq}\mathsf{GenCon}\,A\,M$, with

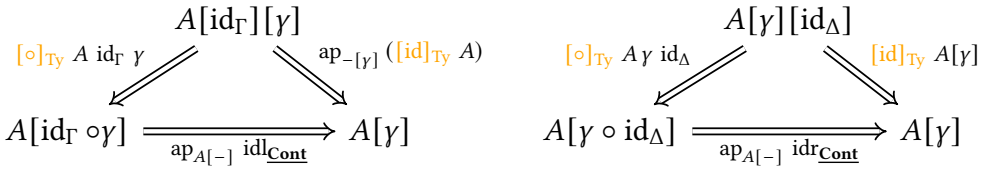
$$M \coloneqq (\lambda\,B\,e \to \mathsf{cong}\,(\lambda\,A \to A[\gamma])\,(\mathsf{uaGenCon}\,e) \equiv \mathsf{uaGenCon}\,(\mathsf{cong}[\gamma]\,e).$$

That M holds on A and $(\mathsf{id}{\simeq}\mathsf{GenCon}\,A)$ follows from Lemmas 3.4.5 and 3.4.7. □

Proofs of $[\mathsf{id}]_{\mathsf{Ty}}$-cohL, $[\mathsf{id}]_{\mathsf{Ty}}$-cohR, and $[\circ]_{\mathsf{Ty}}$-coh

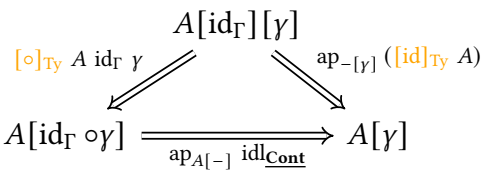
We start off by proving the triangulators.

(Triangulators) We check left and right higher coherence laws for $[\mathsf{id}]_{\mathsf{Ty}}$. Namely, if we assume $\gamma : \Delta \to \Gamma$ and $A : \mathsf{Ty}\,\Gamma$, we need to show the below triangles commute. The left diagram is called $[\mathsf{id}]_{\mathsf{Ty}}$-cohL and the right diagram is called $[\mathsf{id}]_{\mathsf{Ty}}$-cohR.



We give here the proof of $[\mathsf{id}]_{\mathsf{Ty}}$-cohL, the proof of $[\mathsf{id}]_{\mathsf{Ty}}$-cohR follows the same general blueprint. Both proofs are fully written out in our formalisation.

($[\mathsf{id}]_{\mathsf{Ty}}$-cohL) Given $A : \mathsf{Ty}\,\Gamma$ and $\gamma : \Delta \to \Gamma$, we show that the below diagram of equalities between GenCons commutes.



Since we work in `Cubical Agda`, we start off by stating the above triangle as a Square with one side being refl.

$$[\mathsf{id}]_{\mathsf{Ty}}\text{-cohL} : \forall\{\Gamma, \Delta\}\,(A : \mathsf{Ty}\,\Gamma)\,(\gamma : \Delta \to \Gamma) \to$$

```
Square
  (λ i → ([id]Ty A i) [γ]) -- bottom
  (λ i → A[idl_Cont i]) -- top
  ([∘]Ty A idΓ γ) -- left
  refl -- right
```

This can be drawn pictorially as the below square.

$$\begin{array}{ccc} A[\mathrm{id}_\Gamma \circ \gamma] & \xrightarrow{\lambda\, i \to A[\mathrm{idl}_{\underline{\mathbf{Cont}}}\, i]} & A[\gamma] \\ {\scriptstyle [\circ]_{\mathrm{Ty}}\ A\ \mathrm{id}_\Gamma\ \gamma}\uparrow & & \uparrow{\scriptstyle \mathrm{refl}} \\ A[\mathrm{id}_\Gamma][\gamma] & \xrightarrow[\lambda\, i \to ([\mathrm{id}]_{\mathrm{Ty}}\, A\, i)[\gamma]]{} & A[\gamma] \end{array} \tag{3.4.9}$$

In our case, we have that definitionally, the top-left corner is $A[\gamma]$ and the top arrow is refl. Moreover, we can take advantage of the groundwork we constructed earlier for GenCons to express each arrow in the above square to be of the form uaGenCon ..., as below.

$$\begin{array}{ccc} A[\gamma] & \xrightarrow{\mathrm{uaGenCon}\ (\mathrm{id}{\simeq}\mathrm{GenCon}\ A[\gamma])} & A[\gamma] \\ {\scriptstyle \mathrm{uaGenCon}\ ([\circ]\text{T-eq}\ A\ \mathrm{id}_\Gamma\ \gamma)}\uparrow & & \uparrow{\scriptstyle \mathrm{uaGenCon}\ (\mathrm{id}{\simeq}\mathrm{GenCon}\ A[\gamma])} \\ A[\mathrm{id}_\Gamma][\gamma] & \xrightarrow[\mathrm{uaGenCon}\ (\mathrm{cong}[\gamma]\ ([\mathrm{id}]\text{T-eq}\ A))]{} & A[\gamma] \end{array} \tag{3.4.10}$$

We will be precise about how squares 3.4.9 and 3.4.10 are equal later. At this point, we use uaGenConSquare to shift our goal to constructing the ≃GenConSquare below.

```
sqGenCon : ≃GenConSquare
             (cong[γ] ([id]T-eq A)) -- bottom
             (id≃GenCon A[γ]) -- top
             ([∘]T-eq A idΓ γ) -- left
             (id≃GenCon A[γ]) -- right
```

We construct sqGenCon's components as follows.

- For s-eq, since (id≃GenCon $A[\gamma]$) .s≃ is the identity equivalence, we have to prove that given $s\Delta : \Delta$ .S and $sA : A$ .S (γ .s $s\Delta$),

  $$((([\circ]\text{T-eq}\ A\ \mathrm{id}_\Gamma\ \gamma)\ .s{\simeq}\ s\Delta)\ .\mathrm{fst}\ sA \equiv ((\mathrm{cong}[\gamma]\ ([\mathrm{id}]\text{T-eq}\ A))\ .s{\simeq}\ s\Delta)\ .\mathrm{fst}\ sA.$$

  By definition these are both the identity equivalence on sA, so this is provable by refl.
- For p-eq, we assume $s\Delta : \Delta$ .S, $sA : A$ .S (γ .s $s\Delta$), and p : SetPushout (γ .p) (| inl – $|_2$). Recall that p-eq is a PathP over s-eq, which in this case is refl, so p-eq now becomes a homogenous equation. We need to show that

  $$\mathrm{pushoutCompEq}\ \mathrm{id}\ (A\ .\mathrm{Q}\ sA)\ (\gamma\ .\mathrm{p})\ .\mathrm{fst}\ p \equiv$$

$$\mathsf{cong}[\gamma]\ (\mathsf{[id]T\text{-}eq}\ A)\ .\mathsf{p}\simeq\ (s\Delta, sA))\ .\mathsf{fst}\ p.$$

We do this by applying elimSetPushout to eliminate out of p. In the *inr'* case of elimSetPushout, we once again apply elimSetPushout .

- For q-eq, our goal boils down to constructing a Square where every edge is refl, which after introducing a path variable i can be proved by refl.

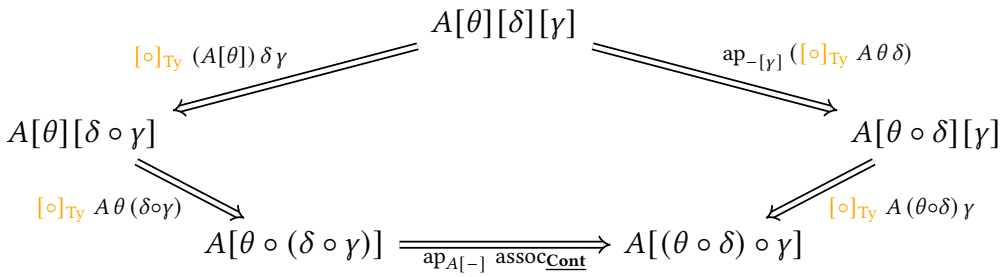
So we now have square 3.4.10. To get square 3.4.9, we note that

- (bottom) uaGenCon (cong[γ] ([id]T-eq A)) $\equiv$ cong ($\lambda A \to A[\gamma]$) ($[\mathsf{id}]_{\mathsf{Ty}}\ A$) holds by Lemma 3.4.8.
- (top) uaGenCon (id≃GenCon $A[\gamma]$) $\equiv$ refl by uaGenConId
- (left) uaGenCon ([∘]T-eq A id_Γ γ) = $[\circ]_{\mathsf{Ty}}\ A\ \mathrm{id}_\Gamma\ \gamma$ by definition
- (right) same as (top).

Therefore we can get 3.4.9 by transporting over a Square with the above mentioned edges. This concludes our proof of $[\mathsf{id}]_{\mathsf{Ty}}$-cohL.

We now move on to proving the pentagonator law.

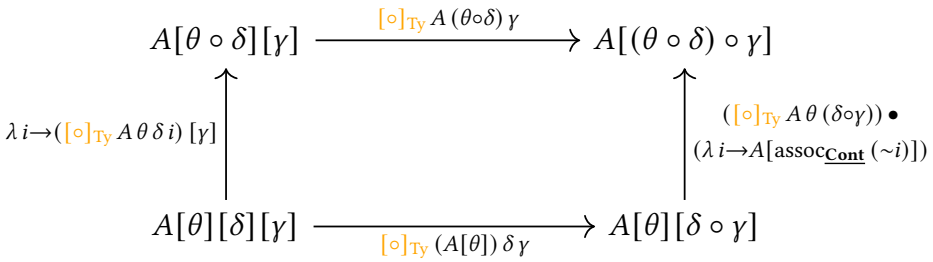
(Pentagonator, $[\circ]_{\mathsf{Ty}}$-coh) We check a higher coherence law for $[\circ]_{\mathsf{Ty}}$, which we call $[\circ]_{\mathsf{Ty}}$-coh. Namely, if we assume $\Xi \xrightarrow{\gamma} \Gamma \xrightarrow{\delta} \Delta \xrightarrow{\theta} \Theta$ and $A : \mathsf{Ty}\ \Theta$, then we need to show that the below pentagon commutes.



We start off by stating the above pentagon as a Square. We will have to adjust for having less edges by composing two arrows, as shown below.

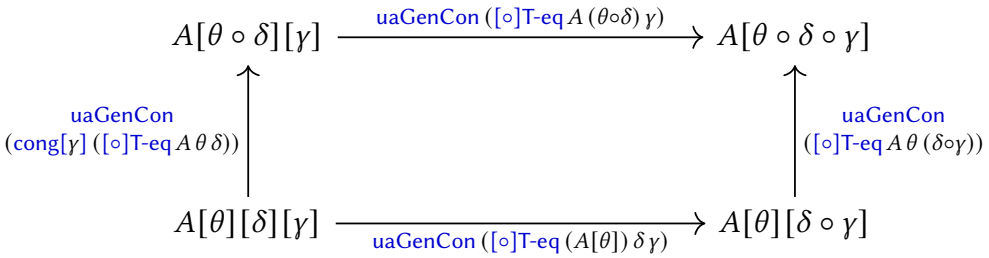
$$\begin{aligned}
&[\circ]_{\mathsf{Ty}}\text{-coh} : \forall\{\Gamma\ \Delta\ \Theta, \Xi\}\ (A : \mathsf{Ty}\ \Theta)\ (\theta : \Delta \to \Theta)\ (\delta : \Gamma \to \Delta)\ (\gamma : \Xi \to \Gamma) \\
&\qquad \mathsf{Square} \\
&\qquad\quad ([\circ]_{\mathsf{Ty}}\ (A[\theta])\ \delta\ \gamma)\ \text{-- bottom} \\
&\qquad\quad ([\circ]_{\mathsf{Ty}}\ A\ (\theta \circ \delta)\ \gamma)\ \text{-- top} \\
&\qquad\quad (\lambda\, i \to ([\circ]_{\mathsf{Ty}}\ A\ \theta\ \delta\ i)\ [\gamma])\ \text{-- left} \\
&\qquad\quad ((\,[\circ]_{\mathsf{Ty}}\ A\ \theta\ (\delta \circ \gamma)) \bullet (\lambda\, i \to A[\mathsf{assoc}_{\underline{\mathbf{Cont}}}\ (\sim i)]))\ \text{-- right}
\end{aligned}$$

The reason we set up the Square this way is that in our case, $\mathsf{assoc}_{\underline{\mathbf{Cont}}}$ is refl, so the composition in the right edge will disappear. We can draw the Square as shown below.



(3.4.11)

Since $\mathrm{assoc}_{\underline{\mathbf{Cont}}}$ is refl, and using our previously defined helper functions, we can redraw this square so that every arrow is in terms of uaGenCon as follows.



(3.4.12)

We now use uaGenConSquare to shift our goal to constructing the ≃GenConSquare below.

```
sqGenCon :≃GenConSquare
            ([∘]T-eq (A[θ]) δ γ) -- bottom
            ([∘]T-eq A (θ ∘ δ) γ) -- top
            (cong[γ] ([∘]T-eq A θ δ)) -- left
            ([∘]T-eq A θ (δ ∘ γ)) -- right
```

We construct sqGenCon as follows.

- For s-eq, given $s\Xi : \Xi.\mathsf{S}$ and $sA : A.\mathsf{S}\ (\theta.\mathsf{s}\ (\delta.\mathsf{s}\ (\gamma.\mathsf{s}\ s\Xi)))$, since the definition of the s≃ component of [∘]T-eq is idEquiv, we just need to show that $sA \equiv sA$ which we prove by refl.
- For p-eq, we assume $s\Xi : \Xi.\mathsf{S}$, $sA : A.\mathsf{S}\ (\theta.\mathsf{s}\ (\delta.\mathsf{s}\ (\gamma.\mathsf{s}\ s\Xi)))$, and

```
p : SetPushout {Γ.P (γ.s sΞ)} {Ξ.P sΞ}
                {SetPushout {Δ.P (δ.s (γ.s sΞ))} {Γ.P (γ.s sΞ)}
                            {SetPushout {Θ.P (θ.s (δ.s (γ.s sΞ)))}
                                        {Δ.P (δ.s (γ.s sΞ))}
                                        {A.P sA}
                                        (θ.p) (A.Q sA)}
                            (δ.p) (| inl − |₂)}
                (γ.p) (| inl − |₂).
```

Note that p is a pasting of 3 SetPushouts. Since p-eq is a PathP over s-eq and we just defined this to be refl, p-eq becomes the below homogenous equation.

$$\begin{aligned}&\mathsf{pushoutCompEq}\ ((\delta\ .\mathsf{p}) \circ (\theta\ .\mathsf{p}))\ (A\ .\mathsf{Q}\,sA)\ (\gamma\ .\mathsf{p})\ .\mathsf{fst}\ \hat{p} \equiv\\ &\mathsf{pushoutCompEq}\ (\theta\ .\mathsf{p})\ (A\ .\mathsf{Q}\,sA)\ ((\gamma\ .\mathsf{p}) \circ (\delta\ .\mathsf{p}))\ .\mathsf{fst}\\ &\quad(\mathsf{pushoutCompEq}\ (\delta\ .\mathsf{p})\ (|\ \mathsf{inl}\ -\ |_2)\ (\gamma\ .\mathsf{p})\ .\mathsf{fst}\ p)\end{aligned}$$

where $\hat{p}$ is p coerced into a pasting of 2 SetPushout squares, with the top 2 squares composed together:

$$\begin{aligned}\hat{p} : \mathsf{SetPushout}\ &\{\Gamma\ .\mathsf{P}\ (\gamma\ .\mathsf{s}\ s\Xi)\}\ \{\Xi\ .\mathsf{P}\ s\Xi\}\\ &\{\mathsf{SetPushout}\ \{\Theta\ .\mathsf{P}\ (\theta\ .\mathsf{s}\ (\delta\ .\mathsf{s}\ (\gamma\ .\mathsf{s}\ s\Xi)))\}\{\Gamma\ .\mathsf{P}\ (\gamma\ .\mathsf{s}\ s\Xi)\}\\ &\qquad\{A\ .\mathsf{P}\ sA\}\\ &\qquad((\delta\ .\mathsf{p}) \circ (\theta\ .\mathsf{p}))\ (A\ .\mathsf{Q}\,sA)\}\\ &(\gamma\ .\mathsf{p})\ (|\ \mathsf{inl}\ -\ |_2).\end{aligned}$$

We prove this by applying elimSetPushout 3 times, first applying it to p, and with subsequent applications nested inside each other for the *inr'* cases.

- For q-eq, our goal boils down to constructing a Square where every edge is refl, which after introducing a path variable i can be proved by refl.

We now have square 3.4.12. To get square 3.4.11, we note that

- (bottom) $\mathsf{uaGenCon}\ ([\circ]\mathsf{T\text{-}eq}\ (A[\theta])\ \delta\ \gamma) = [\circ]_{\mathsf{Ty}}\ (A[\theta])\ \delta\ \gamma$ by definition
- (top) $\mathsf{uaGenCon}\ ([\circ]\mathsf{T\text{-}eq}\ A\ (\theta \circ \delta)\ \gamma) = [\circ]_{\mathsf{Ty}}\ A\ (\theta \circ \delta)\ \gamma$ by definition
- (left) $\mathsf{uaGenCon}\ (\mathsf{cong}[\gamma]\ ([\circ]\mathsf{T\text{-}eq}\ A\ \theta\ \delta)) \equiv \mathsf{cong}\ (\lambda A \to A[\gamma])\ ([\circ]_{\mathsf{Ty}}\ A\ \theta\ \delta)$ holds by Lemma 3.4.8
- (right) $\mathsf{uaGenCon}([\circ]\mathsf{T\text{-}eq}\ A\ \theta\ (\delta \circ \gamma)) \equiv ([\circ]_{\mathsf{Ty}}\ A\ \theta\ (\delta \circ \gamma)) \bullet \mathsf{cong}\ (\lambda X \to A[X])\ (\mathsf{sym}\ \mathsf{assoc}_{\underline{\mathbf{Cont}}})$ by $\mathsf{rUnit} : \forall p, p \equiv p \bullet \mathsf{refl}$.

Therefore we get 3.4.11 by transporting over a Square with the above mentioned edges. This concludes our proof of $[\circ]_{\mathsf{Ty}}$-coh.

3.4.5 Terms and term substitution

Before we define terms and term substitution, we recall that every $A : \mathsf{Ty}\ \Gamma$ gives rise to a functor from the category of elements of $[\![\Gamma]\!]$ to the category of h-sets, whose definition on objects we explicitly write down below.

$$\begin{aligned}[\![A]\!] &: \textstyle\int [\![\Gamma]\!] \to \underline{\mathbf{Set}}\\ [\![A]\!]\ (X, (s\Gamma, p\Gamma)) &\coloneqq \sum_{sA : S_A} (\textstyle\int [\![\Gamma]\!](P_A\ sA, (X, (s\Gamma, p\Gamma))))\\ &= \sum_{sA : S_A} (f : P_A^X\ s \to X) \to (P_A^s\ sA \equiv s\Gamma) \times (f \circ P_A^f\ sA \equiv p\Gamma)\\ &\quad \text{for } s\Gamma : S_\Gamma, p\Gamma : P_\Gamma\ s\Gamma \to X.\end{aligned}$$

We also state a covariant and dependent version of the Yoneda lemma.

Theorem 3.4.13 (Covariant dependent Yoneda lemma)**.** *Given a category* $\underline{\boldsymbol{C}}$*, a functor* $F : \underline{\boldsymbol{C}} \to \underline{\boldsymbol{Set}}$*, a functor* $A : (\int F) \to \underline{\boldsymbol{Set}}$*, an object* $I : |\underline{\boldsymbol{C}}|$*, and* $\phi : F\,I$*, we have a natural isomorphism*

$$\int_{X:|\underline{\boldsymbol{C}}|} (i : \underline{\boldsymbol{C}}(I, X)) \to A(X, (F\,i)\,\phi) \cong A(I, \phi).$$

The term structure $\mathrm{Tm} : (\int \mathrm{Ty})^{\mathrm{op}} \to \underline{\mathbf{Set}}$ is defined as follows.

- (Objects) For a set-container Γ and a generalised container $A : \mathrm{Ty}\ \Gamma$, $\mathrm{Tm}\ (\Gamma, A)$ is the set of dependent natural transformations $[\![\Gamma]\!] \to [\![A]\!]$. If $a : \mathrm{Tm}\ (\Gamma, A)$, then

$$a : \int_{X:\mathrm{Set}} (\gamma : [\![\Gamma]\!]\ X) \to [\![A]\!]\ (X, \gamma)$$

  such that for $f : X \to Y$ in $\underline{\mathbf{Set}}$, the below dependent square commutes.

$$\begin{array}{ccc} ((s\Gamma, p\Gamma) : [\![\Gamma]\!]\ X) & \xrightarrow{a_X} & [\![A]\!]\ (X, (s\Gamma, p\Gamma)) \\ \downarrow{\scriptstyle [\![\Gamma]\!]\ f} & & \downarrow{\scriptstyle [\![A]\!]\ f} \\ ((s\Gamma, f \circ p\Gamma) : [\![\Gamma]\!]\ Y) & \xrightarrow[a_Y]{} & [\![A]\!](Y, (s\Gamma, f \circ p\Gamma)) \end{array}$$

  We can rewrite the type of a by using the covariant dependent Yoneda lemma (Theorem 3.4.13) as follows.

$$\begin{aligned} a &: \int_{X:\mathrm{Set}} \left((s\Gamma, p\Gamma) : \sum_{s\Gamma:S_\Gamma} (P_\Gamma\ \gamma_s \to X)\right) \to [\![A]\!]\ (X, (s\Gamma, p\Gamma)) \\ &\cong (s\Gamma : S_\Gamma) \to \int_{X:\mathrm{Set}} (p\Gamma : P_\Gamma\ s\Gamma \to X) \to ([\![A]\!]\ (X, (s\Gamma, p\Gamma))) \\ &\cong (s\Gamma : S_\Gamma) \to [\![A]\!]\ (P_\Gamma\ s\Gamma, (s\Gamma, \mathrm{id})) \\ &= (s\Gamma : S_\Gamma) \to \sum_{sA:S_A} (pA : P_A^X\ sA \to P_\Gamma\ s\Gamma) \to (P_A^s\ s_a \equiv s\Gamma) \times (pA \circ P_A^f\ sA \equiv \mathrm{id}) \\ &\cong \sum_{f_s :\ S_\Gamma \to S_A} \left(f_p : (s\Gamma : S_\Gamma) \to P_A^X\ (f_s\ s\Gamma) \to P_\Gamma\ s\Gamma\right) \to \\ &\quad \left((s\Gamma : S_\Gamma) \to P_A^s\ (f_s\ s\Gamma) \equiv s\Gamma\right) \times \left((s\Gamma : S_\Gamma) \to (f_p\ s\Gamma) \circ (P_A^f\ (f_s\ s\Gamma)) \equiv \mathrm{id}\right). \end{aligned} \tag{3.4.14}$$

  We can simplify the above type further by going through the indexed-fibred translation detailed in Section 3.4.2. After doing this, we end up with $\mathrm{Tm}\ (\Gamma, A)$ having the below components.

```
record Tm (Γ : SetCon) (A : GenCon Γ) : Type where
  field
    S : (sΓ : Γ .S) → A .S sΓ
    P : {sΓ : Γ .S} → A .P (S sΓ) → Γ .P sΓ
    Q : {sΓ : Γ .S} {pΓ : Γ .P sΓ} → pΓ ≡ P (A .Q (S sΓ) pΓ)
```

- (Morphisms) Given a substitution $\gamma : \Delta \to \Gamma$, a type $A : \mathrm{Ty}\ \Gamma$, and a term $a : \mathrm{Tm}\ (\Gamma, A)$, we define $a[\gamma] : \mathrm{Tm}\ (\Delta, A[\gamma])$ as having the below components.

$$\begin{aligned} \mathsf{S} &: (s\Delta : \Delta.\mathsf{S}) \to A[\gamma].\mathsf{S}\ s\Delta \\ \mathsf{S}\ s\Delta &\coloneqq a.\mathsf{S}\ (\gamma.\mathsf{s}\ s\Delta) \end{aligned}$$

  The S component of $a[\gamma]$ is defined as a's S component composed with γ's s component.

$$\mathsf{P} : \{s\Delta : \Delta.\mathsf{S}\} \to A[\gamma].\mathsf{P}\ (\mathsf{S}\ s\Delta) \to \Delta.\mathsf{P}\ s\Delta$$

  The P component of $a[\gamma]$ is a map out of a SetPushout into an h-set, so we define it using elimSetPushout with the maps

$$\begin{aligned} &|\ \mathsf{inl}\ p\Delta\ |_2 \mapsto p\Delta \\ &|\ \mathsf{inr}\ pA\ |_2 \mapsto \gamma.\mathsf{p}\ (a.\mathsf{P}\ pA) \\ &|\ \mathsf{push}\ \_\ |_2 \mapsto \mathsf{cong}\ (\gamma.\mathsf{p})\ (a.\mathsf{Q}). \end{aligned}$$

  Since $A.\mathsf{Q}\ \_\ p = |\ \mathsf{inl}\ p\ |_2$, the Q component of $a[\gamma]$ is just refl.

$$\begin{aligned} \mathsf{Q} &: \{s\Delta : \Delta.\mathsf{S}\}\ \{p\Delta : \Delta.\mathsf{P}\ s\Delta\} \to p\Delta \equiv \mathsf{P}\ (A.\mathsf{Q}\ (\mathsf{S}\ s\Delta)\ p\Delta) \\ \mathsf{Q} &\coloneqq \mathsf{refl} \end{aligned}$$

We omit the proofs of Tm preserving identity ($[\mathrm{id}]_{\mathrm{Tm}}$) and composition ($[\circ]_{\mathrm{Tm}}$) here, but the interested reader can refer to our formalisation [Dam25].

3.4.6 Context comprehension

For a context Γ and a type $A : \mathrm{Ty}\ \Gamma$, we define a context $\Gamma.A : \mathsf{SetCon}$ of Γ extended by A, having the below components.

$$\begin{aligned} \mathsf{S} &: \mathsf{Type} \\ \mathsf{S} &\coloneqq \sum_{\Gamma.\mathsf{S}} A.\mathsf{S} \end{aligned}$$

$$\begin{aligned} \mathsf{P} &: \mathsf{S} \to \mathsf{Type} \\ \mathsf{P}\ (s\Gamma, sA) &\coloneqq A.\mathsf{P}\ sA \end{aligned}$$

Both components are clearly h-sets.

We define a substitution $\mathbf{p} : \mathsf{Sub}\ (\Gamma.A)\ \Gamma$ made up of the following shape and position maps.

$$\begin{aligned} \mathsf{s} &: (\Gamma.A).\mathsf{S} \to \Gamma.\mathsf{S} \\ \mathsf{s}\ (s\Gamma, sA) &\coloneqq s\Gamma \end{aligned}$$

$$\begin{aligned} \mathsf{p} &: \{\hat{s} : (\Gamma.A).\mathsf{S}\} \to \Gamma.\mathsf{P}\ (\mathsf{s}\ \hat{s}) \to (\Gamma.A).\mathsf{P}\ \hat{s} \\ \mathsf{p}\ \{s\Gamma, sA\}\ p\Gamma &\coloneqq A.\mathsf{Q}\ sA\ p\Gamma \end{aligned}$$

We define a term $\mathbf{q} : \mathsf{Tm}\ (\Gamma.A)\ (A[\mathbf{p}])$ having the following components.

$$\begin{aligned} \mathsf{S} &: (s : (\Gamma.A).\mathsf{S}) \to A.\mathsf{S}\ (\mathsf{fst}\ s) \\ \mathsf{S}\ (s\Gamma, sA) &\coloneqq sA \end{aligned}$$

$$\mathsf{P} : \{s : (\Gamma.A).\mathsf{S}\} \to \mathsf{SetPushout}\ (A.\mathsf{Q}\ (\mathsf{fst}\ s))\ (A.\mathsf{Q}\ (\mathsf{fst}\ s)) \to A.\mathsf{P}\ sA$$

The P component maps a SetPushout to the h-set $A.\mathsf{P}\ sA$, so we define it using elimSetPushout, with the maps

$$\begin{aligned} |\ \mathsf{inl}\ pA\ |_2 &\mapsto pA \\ |\ \mathsf{inr}\ pA\ |_2 &\mapsto pA \\ |\ \mathsf{push}\ \_\ |_2 &\mapsto \mathsf{refl}. \end{aligned}$$

$$\begin{aligned} \mathsf{Q} &: \{s : (\Gamma.A).\mathsf{S}\}\ \{pA : A.\mathsf{P}\ (\mathsf{snd}\ s)\} \to pA \equiv pA \\ \mathsf{Q} &\coloneqq \mathsf{refl} \end{aligned}$$

Lastly, for a substitution $\gamma : \Delta \to \Gamma$, a type $A : \mathsf{Ty}\ \Gamma$, and a term $a : \mathsf{Tm}\ (\Delta, A[\gamma])$, we define a context substitution

$$\langle \gamma, a \rangle : \mathsf{Sub}\ \Delta\ (\Gamma.A)$$

with components

$$\begin{aligned} \mathsf{s} &: \Delta.\mathsf{S} \to (\Gamma.A).\mathsf{S} \\ \mathsf{s}\ s\Delta &\coloneqq (\gamma.\mathsf{s}\ s\Delta, a.\mathsf{S}\ s\Delta) \\ \mathsf{p} &: \{s\Delta : \Delta.\mathsf{S}\} \to (\Gamma.A).\mathsf{P}\ (\mathsf{s}\ s\Delta) \to \Delta.\mathsf{P}\ s\Delta \\ \mathsf{p}\ q &\coloneqq a.\mathsf{P}\ |\ \mathsf{inr}\ q\ |_2. \end{aligned}$$

We omit the proofs of the equalities $\mathbf{p}\beta$, $\mathbf{q}\beta$, $\langle\eta\rangle$, and $\langle\circ\rangle$ here, but the interested reader can find them in our formalisation [Dam25].

3.4.7 Type formers

Now that we established that containers form a GCwF, we can ask whether this model of type theory supports type formers we might be interested, such as dependent sums, dependent products, the identity type, and universes. We leave most of this for future work, but we discuss our brief investigation into dependent sums and products.

Definition 3.4.15. A (G)CwF has *dependent sums*, or Σ*-types*, if for every context Γ in $\underline{\mathbf{C}}$ and types $A : \mathsf{Ty}\ \Gamma$ and $B : \mathsf{Ty}\ (\Gamma.A)$, there is a context $\Sigma\ A\ B$ in $\underline{\mathbf{C}}$ such that the following isomorphism holds and is natural in Γ

$$\mathsf{Tm}\ (\Gamma, \Sigma\ A\ B) \cong (a : \mathsf{Tm}\ (\Gamma, A)) \times \mathsf{Tm}\ (\Gamma, B[\langle \mathsf{id}, a \rangle])$$

and for $\gamma : \Delta \to \Gamma$, the below substitution law holds.

$$\Sigma\ A\ B[\gamma] \equiv \Sigma\ (A[\gamma])\ (B[\langle \gamma \circ \mathbf{p}, \mathbf{q} \rangle]).$$ ◊

We conjecture that the container GCwF has dependent sums. Given a set-container Γ and generalised containers $A : \mathrm{Ty}\ \Gamma$ and $B : \mathrm{Ty}\ (\Gamma.A)$, we define the generalised container $\Sigma\, A\, B : \mathrm{Ty}\ \Gamma$ as follows.

$$\begin{aligned} S_{\Sigma\, A\, B} &\coloneqq S_B \\ P^X_{\Sigma\, A\, B} &\coloneqq P^X_B \\ P^s_{\Sigma\, A\, B} &\coloneqq P^s_A \circ P^s_B \\ P^f_{\Sigma\, A\, B}\, s &\coloneqq (P^f_B\, s) \circ P^f_A\, (P^s_B\, s) \end{aligned}$$

Definition 3.4.16. A (G)CwF has *dependent products*, or Π*-types*, if for every context Γ in $\underline{\mathbf{C}}$ and types $A : \mathrm{Ty}\ \Gamma$ and $B : \mathrm{Ty}\ (\Gamma.A)$, there is a context $\Pi\, A\, B$ in $\underline{\mathbf{C}}$ such that the following isomorphism holds and is natural in Γ

$$\mathrm{Tm}\ (\Gamma, \Pi\, A\, B) \cong \mathrm{Tm}\ (\Gamma.A, B)$$

and for $\gamma : \Delta \to \Gamma$, the below substitution law holds.

$$\Pi\, A\, B[\gamma] \equiv \Pi\, (A[\gamma])\ (B[\langle \gamma \circ \mathbf{p}, \mathbf{q} \rangle]). \qquad \Diamond$$

It is still unclear to us whether it is the case that the container GCwF has dependent products, and our investigation so far seems to suggest that we might not have dependent products in general. Conditions under which we could have dependent products between given types A and B are still being looked into.

3.5 Specifying QIITs using Containers

Having constructed a container model of type theory, we describe our brief exploration of giving semantics for QIITs via containers. This section is meant to act as a starting point for future research in this direction.

As we already started discussing in Section 3.1, we do not have a way of expressing a (Q)IIT's signature as an endofunctor, due to the high level of dependency allowed between sorts. Altenkirch et al. [ACD+18] propose a new approach for expressing a QIIT's signature. We illustrate this approach via an example: the Con-Ty QIIT we saw earlier in Section 3.1.

```
data Con : Set
data Ty : Con → Set

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

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

Their approach goes as follows, with the idea being that we start by constructing a category encoding the sorts, and then add an encoding of each constructor at a time.

We start by constructing a base category of sorts $\underline{\mathbf{A}}_0$ according to the types of the sorts, and in this case $\underline{\mathbf{A}}_0 \coloneqq \underline{\mathbf{Fam}}_{\underline{\mathbf{Set}}}$, having

- objects of type
$$\sum_{C:\mathrm{Set}} (T : C \to \mathrm{Set}),$$
- morphisms $(C, T) \to (C', T')$ are functions $f : C \to C'$ and $g : (c : C) \to T\,c \to T'\,(f\,c)$.

Next, we add the specification of the $\diamond : \mathrm{Con}$ constructor. We do so by defining two functors L_0 and R_0, where L_0 specifies the left-hand side of the constructor, or its arguments, and R_0 specifies the right-hand side, or its target type.

$$L_0 : \mathbf{A_0} \to \mathbf{Set} \qquad R_0 : \textstyle\int L_0 \to \mathbf{Set}$$
$$L_0\,(C, T) \coloneqq \top \qquad = \sum \Big(\sum_{C:\mathrm{Set}} (T : C \to \mathrm{Set}) \Big) (L_0\,(C, T)) \to \mathbf{Set}$$
$$R_0\,(C, T, \mathrm{tt}) \coloneqq C$$

In this case, $\diamond$ takes no arguments, hence $L_0\,(C, T)$ is defined as $\top$, and its target type is Con, which we've encoded as the C component of objects in $\underline{\mathbf{A}}_0$. Based on these two functors, we construct a new category $\underline{\mathbf{A}}_1$, having

- objects of type
$$\sum \Big(\sum_{C:\mathrm{Set}} (T : C \to \mathrm{Set}) \Big) (e : (x : L_0\,(C, T)) \to R_0\,(C, T, x))$$
$$\cong \sum \Big(\sum_{C:\mathrm{Set}} (T : C \to \mathrm{Set}) \Big) (e : C)$$
- morphisms $(C, T, e) \to (C', T', e')$ are functions $f : C \to C'$ and $g : (c : C) \to T\,c \to T'\,(f\,c)$ such that $e' \equiv f(e)$.

Now $\underline{\mathbf{A}}_1$ encodes Con, Ty, and $\diamond$. The next step is to add the constructor

$$\_,\_ : (\Gamma : \mathrm{Con}) \to \mathrm{Ty}\,\Gamma \to \mathrm{Con}.$$

We define two functors L_1 and R_1 encoding the left- and right-hand sides of this constructor. It takes two arguments $\Gamma : \mathrm{Con}$ and $\mathrm{Ty}\,\Gamma$, which we encode using L_1, and outputs a Con, which we encode using R_1.

$$L_1 : \mathbf{A_1} \to \mathbf{Set} \qquad R_1 : \textstyle\int L_1 \to \mathbf{Set}$$
$$L_1\,(C,T,e) \coloneqq \sum_{\Gamma:C}(T\,\Gamma) \qquad = \sum\Big(\sum\Big(\sum_{C:\mathrm{Set}}(T : C \to \mathrm{Set})\Big)(e : C)\Big)$$
$$(L_1\,(C,T,e)) \to \mathbf{Set}$$
$$R_1\,(C,T,e,(\Gamma,A)) \coloneqq C$$

We now construct a new category $\underline{\mathbf{A}}_2$ that encodes Con, Ty, $\diamond$, and $\_,\_$ having

- objects of type

$$\sum\Big(\sum\Big(\sum_{C:\mathrm{Set}}(T : C \to \mathrm{Set})\Big)(e : C)\Big)(ex : (x : L_1\,(C,T,e)) \to R_1\,(C,T,e,x))$$
$$\cong \sum\Big(\sum\Big(\sum_{C:\mathrm{Set}}(T : C \to \mathrm{Set})\Big)(e : C)\Big)(ex : \textstyle\sum(\Gamma : C)(T\,\Gamma) \to C)$$

- morphisms $(C,T,e,ex) \to (C',T',e',ex')$ are functions $f : C \to C'$ and $g : (c : C) \to T\,c \to T'\,(f\,c)$ such that $e' \equiv f(e)$ and $ex' \circ (f\,\Gamma, g\,\Gamma\,t) \equiv f \circ ex\,(\Gamma,t)$.

To encode the entire Con-Ty example, we keep going as above, where for every constructor we introduce two functors L_n and R_n and then construct a new category $\underline{\mathbf{A}}_{n+1}$. If the target of a constructor is an equality instead of a sort (like in the case of eq), we simply define R_n to be an equality. At the end of this process, we end up with a category of 'dependent dialgebras', whose initial object corresponds to the QIIT.

This work addresses the problem of how to represent a QIIT's signature, but it does not tell us what kinds of signatures correspond to strictly positive QIITs, i.e. the ones which are guaranteed to have initial (di)algebras. This is precisely the problem we want to tackle, namely, we want to provide a canonical way to represent QIIT specifications that admit an initial algebra. We do so by 'containerifying' the existing semantics illustrated above.

3.5.1 Containerification

Our approach involves applying restrictions to the pair of functors L_n and R_n in order to only allow QIIT specifications that are guaranteed to have an initial algebra. One such restriction is ensuring L_n and R_n are container functors. Since these functors will have types $L_n : \underline{\mathbf{A}}_n \to \mathbf{Set}$ and $R_n : \int L_n \to \mathbf{Set}$, the first restriction we identified is for L_n and R_n to be generalised container functors. Recall from Section 1.5 that a generalised container over an arbitrary category $\underline{\mathbf{C}}$ is given by a set of shapes $S : \mathrm{Set}$ and a family of positions over the shapes $P : S \to |\underline{\mathbf{C}}|$, written $S \triangleleft_{\underline{\mathbf{C}}} P$. This gives rise to a functor $[\![S \triangleleft P]\!] : \underline{\mathbf{C}} \to \mathbf{Set}$, which on objects $X : |\underline{\mathbf{C}}|$ is defined as $[\![S \triangleleft P]\!]\,X \coloneqq \sum(s : S)(\underline{\mathbf{C}}(P\,s, X))$.

In the case that L_n and R_n are generalised container functors, we have $S_{L,n} : \mathrm{Set}$ and $P_{L,n} : S_{L,n} \to |\underline{\mathbf{A}}_n|$ and be able to write L_n as

$$L_n : \underline{\mathbf{A}}_n \to \mathbf{Set}$$

$$\begin{aligned} L_n\, X &\cong [\![S_{L,n} \triangleleft P_{L,n}]\!]\, X \\ &= \textstyle\sum_{s : S_{L,n}} \underline{\mathbf{A}}_n (P_{L,n}\, s, X). \end{aligned}$$

We would also have $S_{R,n} : \mathrm{Set}$ and $P_{R,n} : S_{R,n} \to |\int L_n|$ with components $P^X_{R,n}, P^s_{R,n}$, and $P^f_{R,n}$. Further restrictions we identified for strict positivity are (i) $S_{R,n} = S_{L,n}$ and (ii) $P^s_{R,n}\, t = t$, which allow us to write $R_n : \int L_n \to \underline{\mathbf{Set}}$ just using the two parameters $P^X_{R,n}$ and $P^f_{R,n}$:

$$\begin{aligned} R_n\, (X, (s, f)) &\cong [\![(t : S_{R,n}) \triangleleft (P^X_{R,n}\, t, (P^s_{R,n}\, t, P^f_{R,n}\, t))]\!]\, (X, (s, f)) \\ &\cong \textstyle\sum (h : \underline{\mathbf{A}}_n (P^X_{R,n}\, s, X)) (h \circ P^f_{R,n}\, s = f). \end{aligned}$$

Assuming the existence of QIITs in the metatheory, this scheme is general enough to encode both point and path constructors.

The 'containerified' semantics described above also gives rise to a syntax for QIITs. A specification of a QIIT consists of a list of constructors, each of which is specified by the parameters $S_{L,n}, P_{L,n}, P^X_{R,n}$ and $P^f_{R,n}$. We expect this syntax to be a refinement of the theory of signatures presented in [KKA19]. We hope this alternative syntax facilitates a formal reduction from inductive-inductive types (IITs) to inductive families, and helps us identify a so-called QW-type, which would be a universal type to which all strictly positive QIITs could be reduced.

Chapter 4

Distributive Laws of Monadic Containers

We have seen in Chapter 1 that containers give an algebraic presentation of a wide class of strictly positive data types in terms of shapes and positions and they can be interpreted via a fully-faithful functor into endofunctors on **Set**. Often, reasoning about strictly positive data types in terms of their container representation is simpler than their functorial representation — for example, transformations between container functors constructed from container morphisms are automatically natural.

Monadic containers are those containers whose interpretation as a **Set** functor carries a monad structure. Monads have received a lot of attention in functional programming [HPW+92] and denotational semantics for their ability to model a wide range of programmatic side-effects [Mog91]. In practice, it is rare for side-effects to appear individually — developing a way of composing monads is useful for situations where multiple effects are interleaved. In general, however, the composition of two monads need not result in another monad. While the category of containers is closed under container composition and is a monoidal category (see Section 1.3.3), monadic containers do not in general compose. Distributive laws [Bec69] were developed as a sufficient condition for such a composition to form a monad, thereby ensuring that the corresponding side-effects are interleaved in a coherent way. Constructing distributive laws is known to be quite difficult due to the complexities involved in checking their axioms, besides the fact that some monads, even if they are composable, do not admit a distributive law in the first place [ZM22, Remark 4.19]. As a result, various work has been done on different approaches for constructing distributive laws [MM07; MM08], and for identifying cases where there are none [ZM22; KS24].

In this chapter, we develop a characterisation of distributive laws of monadic containers [Uus17], with the goal of providing an algebraic way of reasoning about distributive

laws between strictly positive data types. We build on similar characterisations in the literature: Ahman, Chapman, and Uustalu develop directed containers [ACU12], i.e. containers whose functorial interpretation carries a *comonad* structure, and the first and last authors provide a characterisation of their distributive laws [AU13]. Uustalu also develops monadic containers [Uus17], but to our knowledge, no work has been done on characterising their distributive laws. Our work parallels the development in [AU13] for monadic containers and therefore closes this gap. Furthermore, by combining our work with [AU13], we construct characterisations of mixed distributive laws, i.e. of directed containers over monadic containers and vice versa, thereby completing the 'zoo' of container characterisations of (co)monads and their distributive laws.

Formalisation

Our representation of distributive laws involves a list of highly dependent equalities, and therefore lends itself well to formalisation in a proof assistant such as `Cubical Agda` [VMA21], in which we have formalised our characterisations as well as those in [ACU12; AU13]. An HTML rendered version of the formalised statements in this chapter can be found at `stefaniatadama.com/distr-law-thesis-html`, and the source code can be found at [PD25a]. The aforementioned code implicitly assumes that we are dealing with h-sets throughout. A version of this code that makes this explicit and uses definitions from the `Cubical` library [Agd25a] is a work in progress. This version can be found at `github.com/stefaniatadama/cubical/tree/distr-laws`.

While our formalisation is written in a cubical type theory [CCHM18; VMA21], our work holds in any intensional Martin-Löf type theory by assuming extensionality principles that are provable in cubical type theory, namely function extensionality (in particular, we do *not* use univalence).

4.1 Related Work

Polynomial monads (polynomial functors equipped with *cartesian* monad maps) have been studied by Gambino and Kock [GK13], but to our knowledge there is no characterisation of when polynomial monads can be composed via a distributive law or otherwise. Monadic containers fail to be a special case of polynomial monads, as monadic containers are not required to be cartesian. However, we do have that cartesian monadic containers, referred to as Σ-universes by Altenkirch and Pinyo [AP17], are special cases of polynomial monads. Awodey first mentions the connection between lax Σ-universes and a monad structure on polynomial functors in [Awo18, Remark 13].

Manes and Mulry [MM07; MM08] have developed general theorems concerning the existence of distributive laws. Zwart and Marsden [ZM22; Zwa20] have developed 'no-go theorems' for monads that can be represented as algebraic theories, filling many holes in an extended version of the Boom hierarchy [Bun94]. Algebraic theories are disjoint from monadic containers in the sense that all monads representable by algebraic theories are finitary. On the other hand, the container $\top \triangleleft (\lambda \_ \to A)$ can be uniquely equipped with monadic container data, and the extension of this container is not finitary

if A is non-finite [Koc09]. Ahman, Chapman, and Uustalu's characterisation of directed containers [ACU12] has been used by Karamlou and Shah to develop no-go theorems relevant to finite model theory [KS24].

Directed containers and certain theorems in [ACU12] have already been formalised in vanilla `Agda` [Ahm11], but to our knowledge their distributive laws have not. We formalise directed containers and their distributive laws alongside our own developments in `Cubical Agda`, but do not recreate all proofs included in [Ahm11].

4.2 Monadic Containers

Recall that we have previously defined (unary) containers $S \triangleleft P$ (Definition 1.3.1) and their functor representation $[\![S \triangleleft P]\!]$, which we previously referred to as their container functor (Definition 1.3.5). Also, recall that containers form a category $\underline{\mathbf{Cont}}$ (Definition 1.3.9), and that there is a fully-faithful functor $[\![\_]\!] : \underline{\mathbf{Cont}} \to [\underline{\mathbf{Set}}, \underline{\mathbf{Set}}]$ mapping containers to their functorial representation as endofunctors on $\underline{\mathbf{Set}}$ (Definition 1.3.11 and Theorem 1.3.12).

Crucially for us, we can equip $\underline{\mathbf{Cont}}$ with a monoidal category structure, by viewing composition of containers as the tensor product and the identity container $\mathrm{id}^C = \top \triangleleft (\lambda \_ \to \top)$ as the unit object [Uus17]. Recall that given containers $S \triangleleft P$ and $T \triangleleft Q$, their composite, denoted $(S \triangleleft P) \circ (T \triangleleft Q)$, is defined as the container

$$\begin{aligned}(S \triangleleft P) \circ (T \triangleleft Q) &\coloneqq \left((s, f) : \sum_{s:S} (P\,s \to T)\right) \triangleleft \left(\sum_{p:P\,s} Q\,(f\,p)\right) \\ &= ((s, f) : [\![S \triangleleft P]\!]\,T) \triangleleft \left(\sum_{p:P\,s} Q\,(f\,p)\right)\end{aligned}$$

Moreover, the functor $[\![\_]\!] : (\underline{\mathbf{Cont}}, \mathrm{id}^C, \circ) \to ([\underline{\mathbf{Set}}, \underline{\mathbf{Set}}], \mathrm{Id}, \cdot)$ from $\underline{\mathbf{Cont}}$ with the aforementioned monoidal structure to the strict monoidal category of endofunctors on $\underline{\mathbf{Set}}$, is (lax) monoidal, i.e. it preserves monoidal multiplication and unit:

$$\begin{aligned}[\![\mathrm{id}^C]\!] &\cong \mathrm{Id} \\ [\![(S \triangleleft P) \circ (T \triangleleft Q)]\!] &\cong [\![S \triangleleft P]\!] \cdot [\![T \triangleleft Q]\!].\end{aligned}$$

For the rest of this section, we state the definitions of monadic containers and their monad interpretations, and provide some examples.

Monadic containers were developed by Uustalu [Uus17] as a characterisation of monads whose underlying functors are container functors. We adopt conventions used by Altenkirch and Pinyo in [AP17], referring to them as monadic containers (rather than mnd-containers) and using their presentation as lax Σ-universes.

Definition 4.2.1. A *monadic container* is a tuple $(S \triangleleft P, \iota, \sigma, \mathrm{pr})$ where

$$\begin{aligned}&S \triangleleft P : |\underline{\mathbf{Cont}}| \\ &\iota : S\end{aligned}$$

$$\sigma : \prod_{s:S} (P\,s \to S) \to S$$
$$\mathsf{pr} : \prod_{\{s:S\}} \prod_{\{f:P\,s\to S\}} P\,(\sigma\,s\,f) \to \sum_{p:P\,s} P\,(f\,p)$$

satisfying the following equations:

$$\sigma\,\iota\,(\lambda\_ \to s) \equiv s \qquad \sigma\,s\,(\lambda\_ \to \iota) \equiv s$$
$$\mathsf{pr}_2\,\{\iota\}\,\{\lambda\_ \to s\}\,p \equiv p \qquad \mathsf{pr}_1\,\{s\}\,\{\lambda\_ \to \iota\}\,p \equiv p$$

$$\sigma\,s\,(\lambda p \to \sigma\,(f\,p)\,(g \circ (p, -))) \equiv \sigma\,(\sigma\,s\,f)\,(g \circ \mathsf{pr})$$
$$\mathsf{pr}_1\,\{s\}\,\{\lambda p \to \sigma\,(f\,p)\,(g \circ (p,-))\}\,p \equiv \mathsf{pr}_1\,\{s\}\,\{f\}\,(\mathsf{pr}_1\,\{\sigma\,s\,f\}\,\{g \circ \mathsf{pr}\}\,p)$$
$$\mathsf{pr}_1\,\{f\,p\}\,\{\lambda p' \to g\,(p,p')\}\,(\mathsf{pr}_2\,\{s\}\,\{\lambda p \to \sigma\,(f\,p)\,(g \circ (p,-))\}\,p) \equiv$$
$$\quad \mathsf{pr}_2\,\{s\}\,\{f\}\,(\mathsf{pr}_1\,\{\sigma\,s\,f\}\,\{g \circ \mathsf{pr}\}\,p)$$
$$\mathsf{pr}_2\,\{f\,p\}\,\{\lambda p' \to g\,(p,p')\}\,(\mathsf{pr}_2\,\{s\}\,\{\lambda p \to \sigma\,(f\,p)\,(g \circ (p,-))\}\,p) \equiv \mathsf{pr}_2\,\{\sigma\,s\,f\}\,\{g \circ \mathsf{pr}\}\,p$$

where we use the shorthands $\mathsf{pr}_i := \pi_i \circ \mathsf{pr}$, and $(p, -) := \lambda p' \to (p, p')$. The equalities for pr hold up to the corresponding equalities for σ above them. ◊

For clarity we include all implicit arguments in grey in the equations above, but we will often omit them when they can be inferred from the context. For example, given $p : P\,(\sigma\,(\sigma\,s\,f)\,(g \circ \mathsf{pr}))$, we can write the last three equalities for pr as

$$\mathsf{pr}_1\,p = \mathsf{pr}_1\,(\mathsf{pr}_1\,p) \qquad \mathsf{pr}_1\,(\mathsf{pr}_2\,p) = \mathsf{pr}_2\,(\mathsf{pr}_1\,p) \qquad \mathsf{pr}_2\,(\mathsf{pr}_2\,p) = \mathsf{pr}_2\,p$$

We will sometimes abuse notation and refer to a monadic container $(S \triangleleft P, \iota, \sigma, \mathsf{pr})$ by its container $S \triangleleft P$, when ι, σ and pr are clear from the context.

Monadic containers can be thought of as containers whose set of shapes is pointed, and closed under taking 'container-fulls' of shapes, in the sense that any element $(s, f) : [\![S \triangleleft P]\!]\,S$ gives a new shape $\sigma\,s\,f : S$. The role of pr is to specify how positions $p : P\,(\sigma\,s\,f)$ map to positions $p_1 : P\,s$ and $p_2 : P\,(f\,p_1)$.

An alternative intuition is that monadic containers can be seen as lax versions of (Tarski-style) type universes closed under singleton and dependent sum types [AP17]. Shapes are interpreted as codes for types, and the position family is the map that interprets each code as a concrete type. ι is the code for the singleton type, and σ constructs codes for Σ-types.

Definition 4.2.2. A *Σ-universe* $(S \triangleleft P, \iota, \mathsf{un}, \sigma, \mathsf{pr})$ is given by a monadic container $(S \triangleleft P, \iota, \sigma, \mathsf{pr})$ with the isomorphisms

$$\mathsf{un} : P\,\iota \cong \top$$
$$\mathsf{pr} : \prod_{\{s:S\}} \prod_{\{f:P\,s\to S\}} P\,(\sigma\,s\,f) \cong \sum_{p:P\,s} P\,(f\,p)$$

i.e. where for all s and f, $\mathsf{pr}\,\{s\}\,\{f\}$ is an isomorphism, and $P\,\iota$ is isomorphic to $\top$. A Σ-universe is called *univalent* if the family P is injective. ◊

Every monadic container can be interpreted as a monad on **Set**, this interpretation being an extension of the usual interpretation of containers as functors.

Definition 4.2.3. The *monad interpretation* of a monadic container $(S \triangleleft P, \iota, \sigma, \mathrm{pr})$ is defined as the monad $([\![S \triangleleft P]\!], \eta, \mu)$ on **Set**, denoted by $[\![S \triangleleft P, \iota, \sigma, \mathrm{pr}]\!]^{\mathrm{mc}}$, or simply $[\![S \triangleleft P]\!]^{\mathrm{mc}}$ when the monadic container structure is clear from the context, where

$$\begin{aligned} &\eta : \mathrm{Id} \to [\![S \triangleleft P]\!] && \mu : [\![S \triangleleft P]\!] \circ [\![S \triangleleft P]\!] \to [\![S \triangleleft P]\!] \\ &\eta_A\, a := (\iota, \lambda\_ \to a) && \mu_A\, (s, f) := (\sigma\, s\, (\pi_1 \circ f), \pi_2 \circ f \circ \mathrm{pr}). \end{aligned}$$ ◊

Example 4.2.4. The list container $\mathbb{N} \triangleleft \mathsf{Fin}$ can be extended to a monadic container by taking

$$\begin{aligned} \iota &:= 1 \\ \sigma\, n\, f &:= f\, 0 + \cdots + f\, (n-1) \\ \mathrm{pr}_1\, \{n\}\, \{f\}\, p &:= \max \{i \in [0..n) \mid f\, 0 + \cdots + f\, i \leq p\} \\ \mathrm{pr}_2\, \{n\}\, \{f\}\, p &:= p - (f\, 0 + \cdots + f\, (\mathrm{pr}_1\, \{n\}\, \{f\}\, p)). \end{aligned}$$

This is also an example of a Σ-universe. The monad interpretation of this container is isomorphic to the list monad with concatenation. ◊

Example 4.2.5. Given some set S, the container $(S \to S) \triangleleft (\lambda\_ \to S)$ can be extended to a monadic container by taking

$$\begin{aligned} \iota &:= \lambda x \to x \\ \sigma\, f\, g &:= \lambda x \to g\, x\, (f\, x) \\ \mathrm{pr}\, \{f\}\, x &:= (x, f\, x). \end{aligned}$$

The monad interpretation of this is the well-known state monad from functional programming. ◊

Example 4.2.6. The container $(\top + E) \triangleleft \lambda \begin{cases} \mathsf{inl}\ \mathsf{tt} \to \top \\ \mathsf{inr}\ \_ \to \bot \end{cases}$ of coproducts with E can be extended to a monadic container by taking

$$\begin{aligned} \iota &:= \mathsf{inl}\ \mathsf{tt} \\ \sigma\, (\mathsf{inl}\ \mathsf{tt})\, f &:= f\, \mathsf{tt} \\ \sigma\, (\mathsf{inr}\ e)\, \_ &:= e \\ \mathrm{pr}\, \{\mathsf{inl}\ \mathsf{tt}\}\, \mathsf{tt} &:= (\mathsf{tt}, \mathsf{tt}). \end{aligned}$$

This is another example of a Σ-universe. The monad interpretation of this is known to the functional programming community as the exception monad. ◊

Example 4.2.7. Given a monoid $(A, \otimes, e)$, the container $A \triangleleft (\lambda\_ \to \top)$ can be extended to a monadic container by taking

$$\iota := e$$

$$\sigma\, a\, f := a \otimes f\, \mathsf{tt}$$
$$\mathsf{pr}\, \mathsf{tt} := (\mathsf{tt}, \mathsf{tt}).$$

The monadic container equalities hold as a consequence of the monoid equalities. We call this the *writer monadic container*. Monadic containers on $A \triangleleft (\lambda \_ \to \top)$ are in bijection with monoids on A. The monad interpretation of this is typically called the writer monad. ◊

We remark that monadic containers form a category **MCont**, whose identity and composition are inherited from **Cont**. The operation $[\![\_]\!]^{\mathrm{mc}}$ extends to a fully faithful functor of type **MCont** $\to$ **Monad**(**Set**). Similarly to how monads on **Set** are monoids in the monoidal category $([\mathbf{Set}, \mathbf{Set}], \mathrm{Id}, \cdot)$, monadic containers are monoids in the monoidal category $(\mathbf{Cont}, \mathrm{id}^C, \circ)$.

Lastly, we recall the definition of directed containers [ACU12], for use in later sections of this chapter.

Definition 4.2.8. Let $S \triangleleft P$ be a container. A *directed container* on $S \triangleleft P$ is a tuple $(S \triangleleft P, o, \oplus, \downarrow)$ where

$$o : \prod_{\{s:S\}} P\, s$$
$$\downarrow : \prod_{s:S} P\, s \to S$$
$$\oplus : \prod_{\{s:S\}} \prod_{p:P\, s} P\, (s \downarrow p) \to P\, s$$

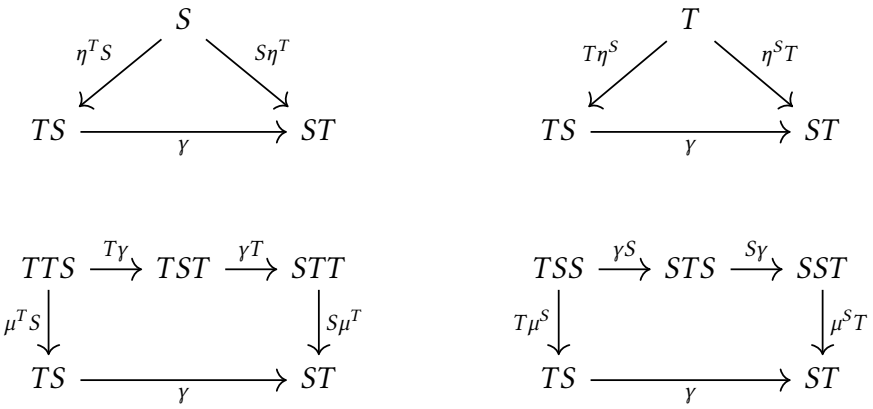
$$s \downarrow o = s$$
$$s \downarrow (p \oplus p') = (s \downarrow p) \downarrow p'$$
$$p \oplus o = p$$
$$o \oplus p = p$$
$$(p \oplus p') \oplus p'' = p \oplus (p' \oplus p'').$$ ◊

4.3 Distributive Laws

The question of whether two monads compose has applications in many areas. In functional programming and denotational semantics of effectful languages, for example, one might consider composition of two (or more) notions of side effect. As remarked in [ZM22, Remark 1.1], distributive laws provide a particularly nice way to compose monads, that imbue the resulting composite monad with a variety of desirable properties.

Distributive laws were first introduced by Beck [Bec69] as a sufficient condition for the composition of the underlying functors of two monads to carry a monad structure.

Definition 4.3.1. Let $\mathbf{S} = (S, \eta^S, \mu^S)$ and $\mathbf{T} = (T, \eta^T, \mu^T)$ be monads. A *distributive law* of $\mathbf{S}$ over $\mathbf{T}$ is a natural transformation $\gamma : TS \Rightarrow ST$ such that the following diagrams commute.



Our approach involves specialising Definition 4.3.1 to the case when S and T are container functors, making $\mathbf{S}$ and $\mathbf{T}$ monads on container functors. We rely on the fact that the container interpretation functor $[\![\_]\!] : \underline{\mathbf{Cont}} \to [\underline{\mathbf{Set}}, \underline{\mathbf{Set}}]$ is fully faithful and monoidal. This gives us that each distributive law of $[\![S \triangleleft P]\!]^{\mathrm{mc}}$ over $[\![T \triangleleft Q]\!]^{\mathrm{mc}}$ corresponds to a unique monadic container distributive law of $S \triangleleft P$ over $T \triangleleft Q$, and lets us directly interpret the diagrams in Definition 4.3.1 as diagrams in $\underline{\mathbf{Cont}}$.

Definition 4.3.2. Let $(S \triangleleft P, \iota^S, \sigma^S, \mathrm{pr}^S)$ and $(T \triangleleft Q, \iota^T, \sigma^T, \mathrm{pr}^T)$ be monadic containers. A *monadic container distributive law* of $T \triangleleft Q$ over[1] $S \triangleleft P$ is given by the data

$$u_1 : \prod_{s:S} (P\,s \to T) \to T$$
$$u_2 : \prod_{s:S} \prod_{f:P\,s \to T} Q\,(u_1\,s\,f) \to S$$
$$v_1 : \prod_{\{s:S\}} \prod_{\{f:P\,s \to T\}} \prod_{q:Q\,(u_1\,s\,f)} P\,(u_2\,s\,f\,q) \to P\,s$$
$$v_2 : \prod_{\{s:S\}} \prod_{\{f:P\,s \to T\}} \prod_{q:Q\,(u_1\,s\,f)} \prod_{p:P\,(u_2\,s\,f\,q)} Q\,(f\,(v_1\,q\,p))$$

which satisfy the following equalities.

$$u_1\,\iota^S\,(\lambda\_ \to t) = t \qquad \text{(unit-}\iota S\text{-}s_1\text{)}$$
$$u_2\,\iota^S\,(\lambda\_ \to t) = \lambda\_ \to \iota^S \qquad \text{(unit-}\iota S\text{-}s_2\text{)}$$
$$v_1\,\{\iota^S\}\,\{\lambda\_ \to t\}\,q\,p = p \qquad \text{(unit-}\iota S\text{-}p_1\text{)}$$
$$v_2\,\{\iota^S\}\,\{\lambda\_ \to t\}\,q\,p = q \qquad \text{(unit-}\iota S\text{-}p_2\text{)}$$

$$u_1\,s\,(\lambda\_ \to \iota^T) = \iota^T \qquad \text{(unit-}\iota T\text{-}s_1\text{)}$$
$$u_2\,s\,(\lambda\_ \to \iota^T) = \lambda\_ \to s \qquad \text{(unit-}\iota T\text{-}s_2\text{)}$$
$$v_1\,\{s\}\,\{\lambda\_ \to \iota^T\}\,q\,p = p \qquad \text{(unit-}\iota T\text{-}p_1\text{)}$$
$$v_2\,\{s\}\,\{\lambda\_ \to \iota^T\}\,q\,p = q \qquad \text{(unit-}\iota T\text{-}p_2\text{)}$$

[1] The ordering of the monadic containers in this terminology follows that used by Beck.

$$u_1\,(\sigma^S\,s\,f)\,(g \circ \mathrm{pr}^S) = u_1\,s\,(\lambda p \to u_1\,(f\,p)\,(g \circ (p, -))) \tag{mul-S-s_1}$$

$$u_2\,(\sigma^S\,s\,f)\,(g \circ \mathrm{pr}^S)\,q = \sigma^S\,(u_2\,s\,(\lambda p \to u_1\,(f\,p)\,(g \circ (p, -)))\,q)\;(\lambda p \to u_2\,(f\,(v_1\,q\,p))\,(g \circ (v_1\,q\,p, -))\,(v_2\,q\,p)) \tag{mul-S-s_2}$$

$$\mathrm{pr}_1^S\,(v_1\,q\,p) = v_1\,q\,(\mathrm{pr}_1^S\,p) \tag{mul-S-p_1}$$

$$\mathrm{pr}_2^S\,(v_1\,q\,p) = v_1\,(v_2\,q\,(\mathrm{pr}_1^S\,p))\,(\mathrm{pr}_2^S\,p) \tag{mul-S-p_{21}}$$

$$v_2\,q\,p = v_2\,(v_2\,q\,(\mathrm{pr}_1^S\,p))\,(\mathrm{pr}_2^S\,p) \tag{mul-S-p_{22}}$$

$$u_1\,s\,(\lambda p \to \sigma^T\,(f\,p)\,(g \circ (p, -))) = \sigma^T\,(u_1\,s\,f)\,(\lambda q \to u_1\,(u_2\,s\,f\,q)\,(g \circ v\,q)) \tag{mul-T-s_1}$$

$$u_2\,s\,(\lambda p \to \sigma^T\,(f\,p)\,(g \circ (p, -)))\,q = u_2\,(u_2\,s\,f\,(\mathrm{pr}_1^T\,q))\,(g \circ v\,q)\,(\mathrm{pr}_2^T\,q) \tag{mul-T-s_2}$$

$$v_1\,(\mathrm{pr}_1^T\,q)\,(v_1\,(\mathrm{pr}_2^T\,q)\,p) = v_1\,q\,p \tag{mul-T-p_1}$$

$$v_2\,(\mathrm{pr}_1^T\,q)\,(v_1\,(\mathrm{pr}_2^T\,q)\,p) = \mathrm{pr}_1^T\,(v_2\,q\,p) \tag{mul-T-p_{21}}$$

$$v_2\,(\mathrm{pr}_2^T\,q)\,p = \mathrm{pr}_2^T\,(v_2\,q\,p) \tag{mul-T-p_{22}}$$

$\Diamond$

We will use the shorthands

$$u\,s\,f \coloneqq (u_1\,s\,f, u_2\,s\,f)$$
$$v\,\{s\}\,\{f\}\,q\,p \coloneqq (v_1\,\{s\}\,\{f\}\,q\,p, v_2\,\{s\}\,\{f\}\,q\,p),$$

to simplify stating examples of monadic container distributive laws. We refer to equalities whose names end in s_1 or s_2 collectively as 'shape' equalities. The ones ending with names in p_1, p_2, p_{21}, or p_{22} are referred to collectively as 'position' equalities. Many of the equalities above are heterogenous/dependent, i.e. they hold up to previously stated equalities.

These equalities may seem rather unwieldy, but often in applications of this characterisation many of the equations simplify considerably. For example, when constructing a distributive law, if we define a u_1, u_2 such that we can take $v_1\,\{s\}\,\{f\}\,q\,p = p$ and $v_2\,\{s\}\,\{f\}\,q\,p = q$, then all the position equalities hold definitionally, and only the shape equalities must be proven (like in Example 4.3.3). The distributive law uniqueness lemmas (Lemmas 4.3.4, 4.3.6 and 4.3.16) and no-go theorem in Section 7 only require consideration of a few of these equalities.

While Definition 4.3.1 is a concise way of stating what a distributive law is, expanding this definition to check the naturality of γ and commutativity of the 4 diagrams turns out to be cumbersome in practice. Furthermore, our phrasing of distributive laws lends itself to formalisation in a proof assistant. In `Cubical Agda`, the dependencies between equations can be explicitly stated in terms of dependent paths, and we can prove properties of distributive laws without descending into "transport hell".

Example 4.3.3. For any E, S, and P, there is a monadic container distributive law of $S \triangleleft P$ over $(\top + E) \triangleleft \lambda \begin{cases} \mathsf{inl}\ \mathsf{tt} \to \top \\ \mathsf{inr}\ \_ \to \bot \end{cases}$ given by

$$u\,(\mathsf{inl}\ \mathsf{tt})\,f \coloneqq (f\,\mathsf{tt}, \lambda\ \_ \to \mathsf{inl}\ \mathsf{tt})$$

$$u\,(\mathsf{inr}\,e)\,\_ \coloneqq (\iota^S, \lambda\,\_ \to \mathsf{inr}\,e)$$
$$v\,\{\mathsf{inl}\,\mathsf{tt}\}\,\{f\}\,p\,\mathsf{tt} \coloneqq (\mathsf{tt}, p).$$ ◊

In fact, this is the only monadic container distributive law of this type, and the characterisation in Definition 4.3.2 offers a succinct proof of this:

Lemma 4.3.4. *The distributive law in Example 4.3.3 is the unique one of* $S \triangleleft P$ *over* $(\top + E) \triangleleft \lambda \begin{cases} \mathsf{inl}\,\mathsf{tt} \to \top \\ \mathsf{inr}\,\_ \to \bot \end{cases}$ *for any* E, S, *and* P.

Proof. We first note that $(\top \to S) \cong S$, and that $\bot \to S$ is contractible (and it has a point $\lambda\,\_ \to \iota^S$). Using these facts and the equalities unit-ιS-s_1, unit-ιS-s_2, unit-ιT-s_1, and unit-ιT-s_2, u is uniquely specified up to function extensionality. We can use the same reasoning and laws unit-ιS-p_1 and unit-ιS-p_2 to see that v is also uniquely specified up to function extensionality. □

Example 4.3.5. For any A, S, and P, there is a monadic container distributive law of $\top \triangleleft (\lambda\,\_ \to A)$ over $S \triangleleft P$ given by

$$u\,s\,f \coloneqq (\mathsf{tt}, \lambda\,\_ \to s)$$
$$v\,a\,p \coloneqq (p, a).$$ ◊

Lemma 4.3.6. *The distributive law in Example 4.3.5 is the unique one of* $\top \triangleleft (\lambda\,\_ \to A)$ *over* $S \triangleleft P$ *for any* A, S, *and* P.

Proof. Follows from the isomorphisms $\top \cong (P\,s \to \top)$ for any $s : S$, and the unit-ιT equalities for monadic container distributive laws. □

Ahman and Uustalu in [AU13] noticed that distributive laws of directed containers generalise *matching pairs of monoid actions*, and the composition of directed containers via a distributive law generalises the *Zappa-Szép product* of monoids [Bri05]. The following example illustrates how distributive laws and composition of monadic containers generalise the same constructions but in a slightly different way, which is unsurprising but worth noting.

Example 4.3.7. Given monoids $(A, \otimes^A, \iota^A)$ and $(B, \otimes^B, \iota^B)$ and a matching pair of monoid actions $(\alpha : A \times B \to A, \beta : A \times B \to B)$, there is a distributive law of $B \triangleleft (\lambda\,\_ \to \top)$ over $A \triangleleft (\lambda\,\_ \to \top)$ given by

$$u\,a\,b \coloneqq (\beta\,(a, b\,\mathsf{tt}), \lambda\,\_ \to \alpha\,(a, b\,\mathsf{tt}))$$
$$v\,\mathsf{tt}\,\mathsf{tt} \coloneqq (\mathsf{tt}, \mathsf{tt})$$

where the distributive law equalities follow from the equalities for the matching pair of monoid actions (up to the isomorphism $(\top \to B) \cong B$). Conversely, any distributive law of this type specifies a matching pair of monoid actions for the relevant monoids. ◊

4.3.1 Composing with distributive laws

Along with distributive laws, Beck also introduced the equivalent notion of compatible composites [Bec69], as a way to specify when a given monad can be considered a composite of two others. In this section, we characterise these in terms of monadic containers. For containers $S \triangleleft P$ and $T \triangleleft Q$, we will often use the shorthand

$$\Diamond_Q^P := \lambda(s, f) \to \sum_{p:P\,s} Q\,(f\,p) : [\![S \triangleleft P]\!]T \to \mathsf{Set}$$

to simplify the presentation of nested Σ-types.

Definition 4.3.8. A *compatible composite monadic container* of $(S \triangleleft P, \iota^S, \sigma^S, \mathrm{pr}^S)$ over $(T \triangleleft Q, \iota^T, \sigma^T, \mathrm{pr}^T)$ is a pair of maps

$$\sigma : \prod_{x:[\![S\triangleleft P]\!]\,T} (\Diamond_Q^P\,x \to [\![S \triangleleft P]\!]\ T) \to [\![S \triangleleft P]\!]\ T$$

$$\mathrm{pr} : \prod_{\{x:[\![S\triangleleft P]\!]\,T\}} \prod_{\{f:\Diamond_Q^P\,s\to[\![S\triangleleft P]\!]\ T\}} \Diamond_Q^P\,(\sigma\,s\,f) \to \sum_{p:\Diamond_Q^P\,s} \Diamond_Q^P\,(f\,p)$$

where:

- $((S \triangleleft P) \circ (T \triangleleft Q), (\iota^S, \lambda\_ \to \iota^T), \sigma, \mathrm{pr})$ is a monadic container
- the container morphisms $(\lambda s \to (s, \lambda\_ \to \iota^T)) \triangleleft \pi_1$ and $(\lambda t \to (\iota^S, \lambda\_ \to t)) \triangleleft \pi_2$ are monadic container morphisms, as defined in [Uus17]
- the middle unitary laws hold:

$$(s, f) = \sigma\,(s, \lambda\_ \to \iota^T)\,(\lambda p \to (\iota^S, \lambda\_ \to f\,(\pi_1\,p)))$$
$$(q, p) = (\pi_1\,(\mathrm{pr}_1\,(q, p)), \pi_2\,(\mathrm{pr}_2\,(q, p)))$$

  where the second equality is dependent on the first. ◊

In Beck's paper (i.e. in the monad setting), distributive laws and compatible composites (which he calls composite triples) are shown to be equivalent. In the container setting, we have seen that monadic container distributive laws of $D = (T \triangleleft Q, \iota^T, \sigma^T, \mathrm{pr}^T)$ over $C = (S \triangleleft P, \iota^S, \sigma^S, \mathrm{pr}^S)$ and distributive laws of their monad interpretations $\gamma : [\![C]\!]^{\mathrm{mc}} \cdot [\![D]\!]^{\mathrm{mc}} \Rightarrow [\![D]\!]^{\mathrm{mc}} \cdot [\![C]\!]^{\mathrm{mc}}$ are in bijection. Compatible composite monadic containers of C over D are also in bijection with monad interpretations $\gamma : [\![C]\!]^{\mathrm{mc}} \cdot [\![D]\!]^{\mathrm{mc}} \Rightarrow [\![D]\!]^{\mathrm{mc}} \cdot [\![C]\!]^{\mathrm{mc}}$. This follows because a compatible composite of C over D

$$\sigma \triangleleft \mathrm{pr} : ((S \triangleleft P) \circ (T \triangleleft Q)) \circ ((S \triangleleft P) \circ (T \triangleleft Q)) \to (S \triangleleft P) \circ (T \triangleleft Q)$$

is a container morphism, the collection of which is in bijection with natural transformations between container functors $\gamma : [\![C \circ D]\!] \cdot [\![C \circ D]\!] \Rightarrow [\![C \circ D]\!]$, by $[\![\_]\!]$ being full and faithful. By monoidality γ has the structure of a distributive law and now that we are back in the monad setting, we can use Beck's result that says that this is equivalent to γ being a monad distributive law. Therefore, since we have already seen that

$$\text{monadic container distributive laws} \simeq \text{monad distributive laws}$$

and we just explained how

$$\text{compatible composite monadic containers} \simeq \text{monad distributive laws},$$

by transitivity, we get that monadic container distributive laws and compatible composites are equivalent.

Following the above argument, we would now like to give a direct construction of the compatible composite monadic container we obtain from a monadic container distributive law, and vice versa.

Proposition 4.3.9 (Compatible composite from a distributive law)**.** *Given monadic containers* $(S \triangleleft P, \iota^S, \sigma^S, \mathrm{pr}^S)$ *and* $(T \triangleleft Q, \iota^T, \sigma^T, \mathrm{pr}^T)$*, and a monadic container distributive law* (u_1, u_2, v_1, v_2) *of* $S \triangleleft P$ *over* $T \triangleleft Q$*, we have a compatible composite monadic container given by*

$$\begin{aligned}
\sigma\,(s, f)\,g &\coloneqq (\sigma^S\,s\,(\lambda p \to u_1\,(f\,p)\,(g_1 \circ (p, -))),\\
&\qquad \lambda p \to \sigma^T\,(u_2\,(f\,(\mathrm{pr}_1^S\,p))\,(g_1 \circ (\mathrm{pr}_1^S\,p, -))\,(\mathrm{pr}_2^S\,p))\\
&\qquad\quad (\lambda q \to g_2\,(\mathrm{pr}_1^S\,p, v_1\,(\mathrm{pr}_2^S\,p)\,q)\,(v_2\,(\mathrm{pr}_2^S\,p)\,q)))\\
\mathrm{pr}\,(p, q) &\coloneqq ((\mathrm{pr}_1^S\,p, v_1\,(\mathrm{pr}_2^S\,p)\,(\mathrm{pr}_1^T\,q)),\\
&\qquad (v_2\,(\mathrm{pr}_2^S\,p)\,(\mathrm{pr}_1^T\,q), \mathrm{pr}_2^T\,q))
\end{aligned}$$

where $g_i \coloneqq \pi_i \circ g$*.*

Proof. Derivations of the 'associativity' equalities are in Appendix A. Proofs of the remaining equalities are included in the `Cubical Agda` formalisation. □

Proposition 4.3.10 (Distributive law from a compatible composite)**.** *Given monadic containers* $(S \triangleleft P, \iota^S, \sigma^S, \mathrm{pr}^S)$ *and* $(T \triangleleft Q, \iota^T, \sigma^T, \mathrm{pr}^T)$*, and a compatible composite monadic container* (σ, pr) *of* $S \triangleleft P$ *over* $T \triangleleft Q$*, then we have a monadic container distributive law given by*

$$\begin{aligned}
u\,s\,f &\coloneqq \sigma\,(\iota^T, (\lambda_ \to s))\,(\lambda p \to (f\,(\pi_2\,p), (\lambda_ \to \iota^S)))\\
v\,q\,p &\coloneqq (\pi_2\,(\mathrm{pr}_1\,(q, p)), \pi_1\,(\mathrm{pr}_2\,(q, p))).
\end{aligned}$$

To see that Proposition 4.3.9 generalises the Zappa-Szép product of monoids, we consider the case in Example 4.3.7 of a distributive law between two writer monadic containers $A \triangleleft (\lambda\_ \to \top)$ and $B \triangleleft (\lambda\_ \to \top)$. The resulting compatible composite monadic container (constructed using Proposition 4.3.9) is again a writer monadic container $(A \times B) \triangleleft (\lambda\_ \to \top)$, whose corresponding monoid is a Zappa-Szép product of the monoids corresponding to $A \triangleleft (\lambda\_ \to \top)$ and $B \triangleleft (\lambda\_ \to \top)$.

Example 4.3.11. Given a set A and monadic container $(S \triangleleft P, \iota^S, \sigma^S, \mathrm{pr}^S)$, the compatible composite monadic container of $\top \triangleleft (\lambda\_ \to A)$ over $S \triangleleft P$ arising from the distributive law in Example 4.3.5 is given by

$$\begin{aligned}
\sigma\,(\mathrm{tt}, f)\,g &\coloneqq (\mathrm{tt}, \lambda a \to \sigma^S\,(f\,a)\,(\lambda p \to \pi_2\,(g\,(a, p)\,a)))\\
\mathrm{pr}\,(a, p) &\coloneqq ((a, \mathrm{pr}_1^S\,p), (a, \mathrm{pr}_2^S\,p)).
\end{aligned}$$

◊

On another note, composite monadic containers can be useful for considering extensions of $(1, \Sigma)$-type-universes (modelled by Σ-universes [AP17]). As an example, we consider how to augment a Σ-universe $(\mathcal{U} \triangleleft \mathrm{El}, \iota^{\mathcal{U}}, \mathsf{un}^{\mathcal{U}}, \sigma^{\mathcal{U}}, \mathsf{pr}^{\mathcal{U}})$ with codes for *refinement types* [JV20]. This can be done by considering compatible composites of $\mathcal{U} \triangleleft \mathrm{El}$ over the 'Maybe' Σ-universe (Example 4.2.6 where $E = \top$). In the below, we refer to the function of type $\top + E \to \mathsf{Set}$ mapping $(\mathsf{inl}\ \mathsf{tt}) \mapsto \top$ and $(\mathsf{inr}\ \_) \mapsto \bot$ as $\mathsf{Maybe}\top$.

Example 4.3.12. Given a Σ-universe $(\mathcal{U} \triangleleft \mathrm{El}, \iota^{\mathcal{U}}, \mathsf{un}^{\mathcal{U}}, \sigma^{\mathcal{U}}, \mathsf{pr}^{\mathcal{U}})$, the composite monadic container corresponding to Example 4.3.3 when $E = \top$ is

$$
\begin{aligned}
\iota &\coloneqq (\iota^{\mathcal{U}}, \lambda\_ \to \mathsf{inl}\ \mathsf{tt}) \\
\sigma\,(s, f)\,g &\coloneqq \left(\sigma^{\mathcal{U}}\ s\ \left(\lambda p \to \begin{cases} g_1\,(p, \mathsf{tt}) & \text{if } f\,p \equiv \mathsf{inl}\ \mathsf{tt} \\ \iota^{\mathcal{U}} & \text{if } f\,p \equiv \mathsf{inr}\ \mathsf{tt} \end{cases}\right),\right. \\
&\qquad \left.\lambda p \to \begin{cases} g_2\,(\mathsf{pr}_1^{\mathcal{U}}\,p, \mathsf{tt})\,(\mathsf{pr}_2^{\mathcal{U}}\,p) & \text{if } f\,(\mathsf{pr}_1^{\mathcal{U}}\,p) \equiv \mathsf{inl}\ \mathsf{tt} \\ \mathsf{inr}\ \mathsf{tt} & \text{if } f\,(\mathsf{pr}_1^{\mathcal{U}}\,p) \equiv \mathsf{inr}\ \mathsf{tt} \end{cases}\right) \\
\mathsf{pr}\,(p, \mathsf{tt}) &\coloneqq ((\mathsf{pr}_1^{\mathcal{U}}\,p, \mathsf{tt}), (\mathsf{pr}_2^{\mathcal{U}}\,p, \mathsf{tt})).
\end{aligned}
$$

This composite monadic container is also a Σ-universe, since $\mathsf{pr}^{\mathcal{U}}$ being an isomorphism implies that pr is an isomorphism, and $\Diamond^{\mathrm{El}}_{\mathsf{Maybe}\top}\,\iota \cong \top$, since $\mathsf{un}^{\mathcal{U}}$ is an isomorphism. $\Diamond$

To see how shapes in the above composite encode refinement types, consider a shape $(s, f) : \sum_{s:\mathcal{U}} (\mathrm{El}\,s \to \top + \top)$. The shape s is a code for the type $\mathrm{El}\,s$, and we can see f as a *predicate* on elements of $\mathrm{El}\,s$. The type that (s, f) encodes is $\Diamond^{\mathrm{El}}_{\mathsf{Maybe}\top}\,(s, f) \coloneqq \sum_{x:\mathrm{El}\,s} \mathsf{Maybe}\top\,(f\,x)$, whose elements are elements of $\mathrm{El}\,s$ for which f 'holds', or is equal to $\mathsf{inl}\ \mathsf{tt}$.

This is an 'augmentation' in the sense that all types encoded by $\mathcal{U}$ have unique codes in the composite universe. Given a shape $s : \mathcal{U}$, the composite shape $(s, \lambda\_ \to \mathsf{inl}\ \mathsf{tt})$ codes for the type with elements from $\mathrm{El}\,s$ that satisfy the 'always true' predicate — this is clearly isomorphic to the type $\mathrm{El}\,s$.

4.3.2 Mixed distributive laws

By combining our characterisation of monadic container distributive laws with Ahman and Uustalu's directed container distributive laws [AU13, Section 4], it is straightforward to obtain a characterisation of mixed container distributive laws.

Definition 4.3.13. Let $(T \triangleleft Q, \iota, \sigma, \mathrm{pr})$ be a monadic container and $(S \triangleleft P, o, \oplus, \downarrow)$ be a directed container. We define a *monadic-directed container distributive law* of $T \triangleleft Q$ over $S \triangleleft P$ as the data

$$
\begin{aligned}
u_1 &: \prod_{s:S} (P\,s \to T) \to T \\
u_2 &: \prod_{s:S} \prod_{f:P\,s \to T} Q\,(u_1\,s\,f) \to S
\end{aligned}
$$

$$v_1 : \prod_{\{s:S\}} \prod_{\{f:P\,s\to T\}} \prod_{q:Q\,(u_1\,s\,f)} P\,(u_2\,s\,f\,q) \to P\,s$$
$$v_2 : \prod_{\{s:S\}} \prod_{\{f:P\,s\to T\}} \prod_{q:Q\,(u_1\,s\,f)} \prod_{p:P\,(u_2\,s\,f\,q)} Q\,(f\,(v_1\,q\,p))$$

which satisfy the following equalities.

$$u_1\,s\,f = f\,(o\,\{s\}) \qquad \text{(unit-}oS\text{-}s\text{)}$$
$$v_1\,\{s\}\,\{f\}\,q\,(o\,\{u_2\,s\,f\,q\}) = o\,\{s\} \qquad \text{(unit-}oS\text{-}p_1\text{)}$$
$$v_2\,\{s\}\,\{f\}\,q\,(o\,\{u_2\,s\,f\,q\}) = q \qquad \text{(unit-}oS\text{-}p_2\text{)}$$
$$u_2\,s\,f\,q \downarrow p = u_2\,(s \downarrow v_1\,q\,p)\,(\lambda p' \to f\,(v_1\,q\,p \oplus p'))\,(v_2\,q\,p) \qquad \text{(mul-}S\text{-}s_3\text{)}$$
$$v_1\,q\,(p \oplus p') = v_1\,q\,p \oplus v_1\,(v_2\,q\,p)\,p' \qquad \text{(mul-}S\text{-}p_1\text{)}$$
$$v_2\,q\,(p \oplus p') = v_2\,(v_2\,q\,p)\,p' \qquad \text{(mul-}S\text{-}p_2\text{)}$$

$$u_2\,s\,(\lambda \_ \to \iota^T) = \lambda \_ \to s \qquad \text{(unit-}\iota T\text{-}s_2\text{)}$$
$$v_1\,\{s\}\,\{\lambda \_ \to \iota^T\}\,q\,p = p \qquad \text{(unit-}\iota T\text{-}p_1\text{)}$$
$$v_2\,\{s\}\,\{\lambda \_ \to \iota^T\}\,q\,p = q \qquad \text{(unit-}\iota T\text{-}p_2\text{)}$$
$$u_2\,s\,(\lambda p \to \sigma^T\,(f\,p)\,(g \circ (p, -)))\,q = u_2\,(u_2\,s\,f\,(\mathrm{pr}_1^T\,q))\,(g \circ v\,q)\,(\mathrm{pr}_2^T\,q) \qquad \text{(mul-}T\text{-}s_2\text{)}$$
$$v_1\,(\mathrm{pr}_1^T\,q)\,(v_1\,(\mathrm{pr}_2^T\,q)\,p) = v_1\,q\,p \qquad \text{(mul-}T\text{-}p_1\text{)}$$
$$v_2\,(\mathrm{pr}_1^T\,q)\,(v_1\,(\mathrm{pr}_2^T\,q)\,p) = \mathrm{pr}_1^T\,(v_2\,q\,p) \qquad \text{(mul-}T\text{-}p_{21}\text{)}$$
$$v_2\,(\mathrm{pr}_2^T\,q)\,p = \mathrm{pr}_2^T\,(v_2\,q\,p) \qquad \text{(mul-}T\text{-}p_{22}\text{)}$$

$\Diamond$

We note that the first six equations are taken from directed container distributive laws, and the remaining equations from monadic container distributive laws. As in directed container distributive laws, the unit-oS-s equality fully determines u_1.

Directed-monadic container distributive laws (of a directed container over a monadic container) can be derived in a similar fashion.

Definition 4.3.14. Let $(S \triangleleft P, \iota, \sigma, \mathrm{pr})$ be a monadic container and $(T \triangleleft Q, o, \oplus, \downarrow)$ be a directed container. We define a *directed-monadic container distributive law* of $T \triangleleft Q$ over $S \triangleleft P$ as the data

$$u_1 : \prod_{s:S} (P\,s \to T) \to T$$
$$u_2 : \prod_{s:S} \prod_{f:P\,s\to T} Q\,(u_1\,s\,f) \to S$$
$$v_1 : \prod_{\{s:S\}} \prod_{\{f:P\,s\to T\}} \prod_{q:Q\,(u_1\,s\,f)} P\,(u_2\,s\,f\,q) \to P\,s$$
$$v_2 : \prod_{\{s:S\}} \prod_{\{f:P\,s\to T\}} \prod_{q:Q\,(u_1\,s\,f)} \prod_{p:P\,(u_2\,s\,f\,q)} Q\,(f\,(v_1\,q\,p))$$

which satisfy the following equalities.

$$u_1\ \iota^S\ (\lambda\ \_ \longrightarrow t) = t \qquad \text{(unit-}\iota S\text{-}s_1\text{)}$$
$$u_2\ \iota^S\ (\lambda\ \_ \longrightarrow t) = \lambda\ \_ \longrightarrow \iota^S \qquad \text{(unit-}\iota S\text{-}s_2\text{)}$$
$$v_1\ \{\iota^S\}\ \{\lambda\ \_ \longrightarrow t\}\ q\ p = p \qquad \text{(unit-}\iota S\text{-}p_1\text{)}$$
$$v_2\ \{\iota^S\}\ \{\lambda\ \_ \longrightarrow t\}\ q\ p = q \qquad \text{(unit-}\iota S\text{-}p_2\text{)}$$
$$u_1\ (\sigma^S\ s\ f)\ (g \circ \mathrm{pr}^S) = u_1\ s\ (\lambda p \longrightarrow u_1\ (f\ p)\ (g \circ (p, -))) \qquad \text{(mul-}S\text{-}s_1\text{)}$$
$$u_2\ (\sigma^S\ s\ f)\ (g \circ \mathrm{pr}^S)\ q = \sigma^S\ (u_2\ s\ (\lambda p \longrightarrow u_1\ (f\ p)\ (g \circ (p, -)))\ q)$$
$$(\lambda p \longrightarrow u_2\ (f\ (v_1\ q\ p))\ (g \circ (v_1\ q\ p, -))\ (v_2\ q\ p)) \qquad \text{(mul-}S\text{-}s_2\text{)}$$
$$\mathrm{pr}_1^S\ (v_1\ q\ p) = v_1\ q\ (\mathrm{pr}_1^S\ p) \qquad \text{(mul-}S\text{-}p_1\text{)}$$
$$\mathrm{pr}_2^S\ (v_1\ q\ p) = v_1\ (v_2\ q\ (\mathrm{pr}_1^S\ p))\ (\mathrm{pr}_2^S\ p) \qquad \text{(mul-}S\text{-}p_{21}\text{)}$$
$$v_2\ q\ p = v_2\ (v_2\ q\ (\mathrm{pr}_1^S\ p))\ (\mathrm{pr}_2^S\ p) \qquad \text{(mul-}S\text{-}p_{22}\text{)}$$

$$u_2\ s\ f\ (o\ \{u_1\ s\ f\}) = s \qquad \text{(unit-}oT\text{-}s\text{)}$$
$$v_1\ (o\ \{u_1\ s\ f\})\ p = p \qquad \text{(unit-}oT\text{-}p_1\text{)}$$
$$v_2\ (o\ \{u_1\ s\ f\})\ p = o\ \{s\} \qquad \text{(unit-}oT\text{-}p_2\text{)}$$
$$u_1\ s\ f \downarrow q = u_1\ (u_2\ s\ f\ q)\ (\lambda p \longrightarrow f\ (v_1\ q\ p) \downarrow v_2\ q\ p) \qquad \text{(mul-}T\text{-}s_1\text{)}$$
$$u_2\ s\ f\ (q \oplus q') = u_2\ (u_2\ s\ f\ q)\ (\lambda p \longrightarrow f\ (v_1\ q\ p) \downarrow v_2\ q\ p)\ q' \qquad \text{(mul-}T\text{-}s_2\text{)}$$
$$v_1\ (q \oplus q')\ p = v_1\ q\ (v_1\ q'\ p) \qquad \text{(mul-}T\text{-}p_1\text{)}$$
$$v_2\ (q \oplus q')\ p = v_2\ q\ (v_1\ q'\ p) \oplus v_2\ q'\ p \qquad \text{(mul-}T\text{-}p_2\text{)}$$

$\Diamond$

Example 4.3.15. There is a monadic-directed distributive law of the reader monadic container $\top \triangleleft (\lambda\ \_ \longrightarrow B)$ over the writer directed container $A \triangleleft (\lambda\ \_ \longrightarrow \top)$ for any sets A and B, given by

$$u\ a\ f := (\mathsf{tt}, \lambda\ \_ \longrightarrow a)$$
$$v\ b\ \mathsf{tt} := (\mathsf{tt}, b).$$

$\Diamond$

Lemma 4.3.16. *The mixed distributive law in Example 4.3.15 is the unique one of* $\top \triangleleft (\lambda\ \_ \longrightarrow B)$ *over* $A \triangleleft (\lambda\ \_ \longrightarrow \top)$.

Proof. Follows from the isomorphism $(\top \longrightarrow \top) \cong \top$ and then directly from the unit-ιT equations of monadic-directed distributive laws. □

It is thematically appropriate to check if these mixed distributive laws correspond to any known constructions on monoids. For monadic-directed distributive laws, we can do this by specialising the mixed distributive law to the case of $B \triangleleft (\lambda\ \_ \longrightarrow \top)$ over $\top \triangleleft (\lambda\ \_ \longrightarrow A)$, where $\top \triangleleft (\lambda\ \_ \longrightarrow A)$ is the reader directed container for a monoid $(A, e^A, \oplus^A)$, and $B \triangleleft (\lambda\ \_ \longrightarrow \top)$ is the writer monadic container for a monoid $(B, e^B, \oplus^B)$.

u_2 and v_2 trivialise, u_1 is uniquely defined as $u_1\ s\ f = f\ e^A$, and the only parameter we are left with is

$$v_1 : \prod_{\{\mathsf{tt}:\top\}} \prod_{\{f:A\to B\}} \prod_{\mathsf{tt}:\top} \prod_{a:A} A.$$

Possible values of v_1 in this case are in bijection with functions $\alpha : (A \to B) \to A \to A$, satisfying the equations:

$$\begin{aligned}
\alpha\ f\ e^A &= e^A \\
\alpha\ f\ (a \oplus^A a') &= \alpha\ f\ a \oplus^A \alpha\ (\lambda x \to f\ (\alpha\ f\ a \oplus^A x))\ a' \\
\alpha\ (\lambda _ \to e^B)\ a &= a \\
\alpha\ (\lambda x \to f\ x \oplus^B g\ x)\ a &= \alpha\ f\ (\alpha\ (\lambda x \to g\ (\alpha\ f\ x))\ a).
\end{aligned}$$

Counterintuitively, we do not end up with a matching pair of monoid actions. Instead we have something that we could call a 'functional monoid action' of the function space $A \to B$ on the monoid A. We have not come across this construction in the literature, but we would not be surprised if it was already known to algebraists.

Despite this, when we specialise directed-monadic container distributive laws to those between writer monadic containers and reader directed containers, we find that they *are* in bijection with matching pairs of monoid actions.

In summary, by restricting distributive laws to those between certain monadic and directed containers (those that are in bijection with monoids), we obtain corresponding constructions on monoids. The table below records the constructions obtained by considering distributive laws of each row container over each column container.

	Writer monadic container	Reader directed container
Writer monadic container	Matching pairs	Functional monoid actions
Reader directed container	Matching pairs	Matching pairs

This highlights an interesting asymmetry, however the reason for this asymmetry is not immediately obvious to us.

4.4 A No-Go Theorem

To show that our characterisation in Definition 4.3.2 is amenable to developing theorems concerning non-existence of distributive laws, we look to Zwart and Marsden [ZM22] for approaches to developing no-go theorems that we may be able to emulate.

To do this, since Zwart and Marsden study algebraic theories, we first need to translate properties of and statements about algebraic theories into ones for monadic containers. Our strategy for this translation was to *very loosely* view aspects of a monadic container as analogous to aspects of an algebraic theory:

- shapes are 'terms'
- positions are 'variables within a term'

- σ is the 'substitution operator'
- ι is a 'variable placeholder'
- pr is a map from 'variable positions in a term resulting from a substitution' to 'variable positions in the terms that were involved in that substitution'.

The point of this is not to form a rigorous connection between monadic containers and algebraic theories, but to see if there are common patterns that can be easily taken advantage of. As it turns out, this loose perspective provides sufficient intuition to emulate the "too many constants" theorem about non-existence of certain composite algebraic theories in [ZM22, Theorem 4.6] into our setting.

Definition 4.4.1. A monadic container $(S \triangleleft P, \iota, \sigma, \mathrm{pr})$ has the *singleton property* if $P\,\iota \cong \top$. ◊

Definition 4.4.2. Given a monadic container $(S \triangleleft P, \iota, \sigma, \mathrm{pr})$ we call any shape $s : S$ where $P\,s \cong \bot$ a *constant shape*. ◊

Definition 4.4.3. A monadic container $(S \triangleleft P, \iota, \sigma, \mathrm{pr})$ satisfies the *(S3) property* if there exist $s : S$ and $f : P\,s \to S$ such that

- $P\,s$ is non-empty,
- equality on $P\,s$ is decidable, and
- for any $p : P\,s$

$$\sigma\,s\left(\lambda p' \to \begin{cases} \iota & \text{if } p \equiv p' \\ f\,p' & \text{if } p \not\equiv p' \end{cases}\right) \equiv \iota.$$ ◊

The name of this property is taken from the analogous property of algebraic theories in [ZM22, Section 4].

Example 4.4.4. The list monadic container $\mathbb{N} \triangleleft \mathsf{Fin}$ satisfies (S3) by taking any $n : \mathbb{N}$ except 0, and taking $f := \lambda\_ \to 0$. Since you can pick any $n : \mathbb{N}$ where $\mathsf{Fin}\,n$ is non-empty, this container actually satisfies a stronger property, directly analogous to (S4) in [ZM22, Section 4].

We denote $\sigma\,n\left(\lambda p' \to \begin{cases} \iota & \text{if } p \equiv p' \\ f\,p' & \text{if } p \not\equiv p' \end{cases}\right)$ (i.e. σ for a fixed p) by $\sigma\,n\,f^p$. First notice that $\sigma\,n\,f^p \equiv 1$ for any $p : \mathsf{Fin}\ n$. This means that the shape equality is satisfied, and also that we only have to consider $p' \equiv 0$ for the position equality.

The sum $f^p\,0 + \cdots + f^p\,(i-1)$ becomes 1 exactly when $i \equiv p$, which gives us that $\max\{i \in [0..n) \mid f^p\,0 + \cdots + f^p\,(i-1) \leq 0\} \equiv p$. Therefore, by definition, we have $\mathrm{pr}_1\,\{n\}\,\{f^p\}\,p' \equiv p$. ◊

Lemma 4.4.5 (Composite (S3))**.** *Let $(S \triangleleft P, \iota^S, \sigma^S, \mathrm{pr}^S)$ be a monadic container satisfying the singleton property and (S3) for some $s : S$ and $f : P\,s \to S$, and $(T \triangleleft Q, \iota^T, \sigma^T, \mathrm{pr}^T)$ be a monadic container. Assume we have a monadic container distributive law (u_1, u_2, v_1, v_2)*

of $S \triangleleft P$ over $T \triangleleft Q$. Then, for any $t : T$ and $p : P\,s$,

$$u_1\, s \left(\lambda p' \to \begin{cases} t & \text{if } p' \equiv p \\ \iota^T & \text{if } p' \not\equiv p \end{cases}\right) \equiv t.$$

Proof. By the chain of equalities:

$$\begin{aligned}
& u_1\, s \left(\lambda p' \to \begin{cases} t & \text{if } p' \equiv p \\ \iota^T & \text{if } p' \not\equiv p \end{cases}\right) \\
& \equiv u_1\, s \left(\lambda p' \to \begin{cases} u_1\, \iota^S\, (\lambda _ \to t) & \text{if } p' \equiv p \\ u_1\, (f\, p')\, (\lambda _ \to \iota^T) & \text{if } p' \not\equiv p \end{cases}\right) && [\text{unit-}\iota S\text{-}s_1 \ \&\ \text{unit-}\iota T\text{-}s_1] \\
& = u_1\, s \left(\lambda p' \to u_1 \left(\begin{cases} \iota^S & \text{if } p' \equiv p \\ f\, p' & \text{if } p' \not\equiv p \end{cases}\right) \left(\lambda _ \to \begin{cases} t & \text{if } p' \equiv p \\ \iota^T & \text{if } p' \not\equiv p \end{cases}\right)\right) \\
& \equiv u_1 \left(\sigma^S\, s \left(\lambda p' \to \begin{cases} \iota^S & \text{if } p' \equiv p \\ f\, p' & \text{if } p' \not\equiv p \end{cases}\right)\right) \left(\lambda p'' \to \begin{cases} t & \text{if } \mathrm{pr}_1^S\, p'' \equiv p \\ \iota^T & \text{if } \mathrm{pr}_1^S\, p'' \not\equiv p \end{cases}\right) && [\text{mul-}S\text{-}s_1] \\
& \equiv u_1\, \iota^S \left(\lambda _ \to \begin{cases} t & \text{if } p \equiv p \\ \iota^T & \text{if } p \not\equiv p \end{cases}\right) && [(\text{S3})] \\
& \equiv t && [\text{unit-}\iota S\text{-}s_1] \ \square
\end{aligned}$$

Theorem 4.4.6 (Too many constants)**.** *Let $(S \triangleleft P, \iota^S, \sigma^S, \mathrm{pr}^S)$ be a monadic container satisfying the singleton property and (S3) for some $s : S$ and $f : P\,s \to S$, such that we have two distinct positions $p, p' : P\,s$, and let $(T \triangleleft Q, \iota^T, \sigma^T, \mathrm{pr}^T)$ be a monadic container. Assume we have a monadic container distributive law (u_1, u_2, v_1, v_2) of $S \triangleleft P$ over $T \triangleleft Q$. Then $T \triangleleft Q$ cannot have more than one distinct constant shape.*

Proof. Assume we have two distinct constant shapes $t_0, t_1 : T, t_0 \not\equiv t_1$. We first have the chain of equalities:

$$\begin{aligned}
& t_0 \\
& \equiv \sigma^T\, t_0\, (\lambda _ \to \iota^T) && [\text{Definition } 4.2.1] \\
& \equiv \sigma^T \left(u_1\, s \left(\lambda y \to \begin{cases} t_0 & \text{if } y \equiv p \\ \iota^T & \text{if } y \not\equiv p \end{cases}\right)\right) \\
& \qquad \left(\lambda y \to u_1 \left(u_2\, s \left(\lambda z \to \begin{cases} t_0 & \text{if } z \equiv p \\ \iota^T & \text{if } z \not\equiv p \end{cases}\right) y\right) \left(\lambda z \to \begin{cases} t_1 & \text{if } v_1\, y\, z \equiv p' \\ \iota^T & \text{if } v_1\, y\, z \not\equiv p' \end{cases}\right)\right)
\end{aligned}$$

[Lemma 4.4.5, $Q\, t_0 \to T$ is contractible and transporting over Lemma 4.4.5 preserves this]

$$\begin{aligned}
& \equiv u_1\, s \left(\lambda y \to \sigma^T \left(\begin{cases} t_0 & \text{if } y \equiv p \\ \iota^T & \text{if } y \not\equiv p \end{cases}\right) \left(\lambda _ \to \begin{cases} t_1 & \text{if } y \equiv p' \\ \iota^T & \text{if } y \not\equiv p' \end{cases}\right)\right) && [\text{mul-}T\text{-}s_1] \\
& = u_1\, s \left(\lambda y \to \begin{cases} \sigma^T\, t_0\, (\lambda _ \to \iota^T) & \text{if } y \equiv p \\ \sigma^T\, \iota^T\, (\lambda _ \to t_1) & \text{if } y \equiv p' \\ \sigma^T\, \iota^T\, (\lambda _ \to \iota^T) & \text{if } y \not\equiv p \text{ and } y \not\equiv p' \end{cases}\right) && [p \not\equiv p' \text{ by assumption}]
\end{aligned}$$

$$= u_1\, s \left(\lambda y \longrightarrow \begin{cases} t_0 & \text{if } y \equiv p \\ t_1 & \text{if } y \equiv p' \\ \iota^T & \text{if } y \not\equiv p \text{ and } y \not\equiv p' \end{cases} \right) \qquad \text{[Definition 4.2.1]}$$

Using the same steps, we can derive

$$t_1 \equiv u_1\, s \left(\lambda y \longrightarrow \begin{cases} t_0 & \text{if } y \equiv p \\ t_1 & \text{if } y \equiv p' \\ \iota^T & \text{if } y \not\equiv p \text{ and } y \not\equiv p' \end{cases} \right)$$

and by composing these two equalities, we get that $t_0 \equiv t_1$. This contradicts our assumption that t_0 and t_1 were distinct. □

With this theorem we can obtain the known result that there are no distributive laws of the list monad over the coproduct monad.

Example 4.4.7. Let E be a set with at least two elements $e_1, e_2 : E$ such that $e_1 \not\equiv e_2$. By Theorem 4.4.6, there is no monadic container distributive law of the list monadic container $\mathbb{N} \triangleleft \mathsf{Fin}$ over the coproduct monadic container $(\top + E) \triangleleft \lambda \begin{cases} \mathsf{inl\ tt} \longrightarrow \top \\ \mathsf{inr}\ \_ \longrightarrow \bot \end{cases}$. ◊

The scope of this theorem is slightly different to Zwart and Marsden's theorem of the same name, as it concerns a different class of monads — those presentable as monadic containers rather than algebraic theories. It is possible to represent any monad on **Set** as a *generalised* algebraic theory [Man12], but it is not immediately clear how to transfer no-go theorems for compositions of algebraic theories into that setting.

4.5 Conclusion

We have characterised monadic, monadic-directed, and directed-monadic container distributive laws, and have motivated their use for development of uniqueness and existence proofs. Further, we have shown such proofs to be amenable to mechanisation in a proof assistant.

Looking at the constructions on monoids that each of these distributive laws generalise, we note a curious asymmetry. Monadic-directed container distributive laws correspond to the notion of 'functional monoid action', as opposed to matching pairs of monoid actions, which all other kinds of distributive law we consider correspond to.

Our work dualises that of Ahman and Uustalu in [AU13], completing the set of characterisations for container (co)monads and their distributive laws, but there are clearly further directions to explore. All of our characterisations and those in [ACU12; AU13; Uus17] could be extended to symmetric (or groupoid) containers [Gyl11], or further to categorified containers [Gyl11; Alt24] to describe a larger class of (co)monads and their distributive laws. For example, it seems possible to represent the finite multiset monad using groupoid containers. We leave this for future work.

In Example 4.3.12, we touch on the connection between type universes and monadic containers. This perspective could be further explored, using this work as a starting point. For example, one could consider developing a notion of composition for type universes with not only codes for $\top$ and Σ, but also for Π types.

Chapter 5

Conclusion & Further Work

This thesis focussed on two main themes: exploring the theory of containers in a context without uniqueness of identity proofs, thereby working in a setting compatible with homotopy type theory, and formalising much of this theory in the proof assistant `Agda`. We started by giving an overview of the existing literature on containers and proving some minor results, such as generalising Cartesian closure from ordinary containers to indexed containers, and showing closure under products and coproducts of generalised containers. Then, we generalised and fully formalised the result that 'containers are closed under least and greatest fixed points' into a setting without uniqueness of identity proofs, suggesting that similar closure results should hold for more general kinds of containers, such as groupoid (or symmetric) containers and categorified containers. Next, we constructed the groupoid category with families of containers and gave a careful account of its formalisation, with the main novelty being the three coherence proofs required to make this a *groupoid* model of type theory. We also sketched how this model could play a part in obtaining semantics for quotient inductive-inductive types using containers. Finally, we applied the theory of containers to monad distributive laws, and characterised monadic distributive laws as well as mixed monadic-comonadic distributive laws. These characterisations were formalised, and we presented various examples to showcase the utility of our formalisation in proving uniqueness of certain distributive laws.

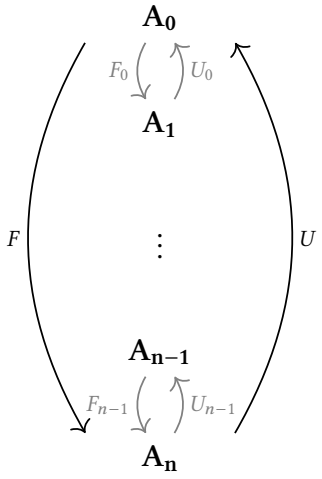
The different projects undertaken in this thesis still have various interesting open questions related to them, and we outline some of them here to act as a starting point for future work.

While we have worked out all the basic details of the container GCwF, type formers for this model of type theory remain mostly unexplored. The question of whether this model has Π-types is of particular interest, since von Glehn's polynomial functor model which has the same contexts and substitutions as our model but different types and

terms [vGle15], was shown to have Π-types but refute function extensionality.

Our original motivation for the container model of type theory has to do with providing semantics for QIITs via containers, as detailed in Section 3.1. To proceed in this direction, one generalisation we should make is to construct a variant of our container model of type theory, where contexts are now generalised containers (as opposed to set-containers). This is because, for our application of giving QIIT semantics, we need to construct the model with respect to an already constructed category of algebras instead of **Set**. As detailed in Section 3.5, we want to restrict the functors L_n and R_n, representing the left- and right-hand sides of a constructor respectively, to be container functors, among other restrictions. This is to ensure that the constructor is strictly positive. However, L_n and R_n are not of type $\underline{\mathbf{Set}} \to \underline{\mathbf{Set}}$ but $\underline{\mathbf{C}} \to \underline{\mathbf{Set}}$, where for L_n, $\underline{\mathbf{C}}$ is a category of algebras, and for R_n, $\underline{\mathbf{C}}$ is the category of elements over L_n. Hence L_n and R_n are required to be *generalised* container functors.

Another pertinent research direction relating to providing semantics for QIITs via containers is the construction of an initial object in the category of algebras corresponding to a QIIT, as defined in [ACD+18]. The construction as described in [ACD+18] of this category $\underline{\mathbf{A}}_n$ of dependent dialgebras representing a QIIT is as follows. We start off by defining a category $\underline{\mathbf{A}}_0$ of the type's sorts. In general, any number of sorts can be reduced to two sorts $\mathrm{U} : \mathrm{Set}, \mathrm{El} \colon \mathrm{U} \to \mathrm{Set}$ (by what is informally known as the 'Szumi trick'), so we can fix $\underline{\mathbf{A}}_0$ to be the category of such families of sets. Next, we define functors L_0 and R_0 representing the first constructor of the type, using which we construct the next category $\underline{\mathbf{A}}_1$ of algebras. We keep going in this way, adding one constructor at a time, until we have added all constructors and get to $\underline{\mathbf{A}}_n$, the category of dependent dialgebras corresponding to the QIIT.



Our ultimate goal is an extistence result for the initial object in $\underline{\mathbf{A}}_n$, provided that all the constructors added along the way were strictly positive, i.e. all the L_n's and R_n's were generalised container functors (as well as some other restrictions). We always have the forgetful functors $U_n \colon \underline{\mathbf{A}}_{n+1} \to \underline{\mathbf{A}}_n$. What we would like to be able to define in general are the free functors F_n as the left adjoints of U_n. If we have these free functors, we can then compose them to get $F \colon \underline{\mathbf{A}}_0 \to \underline{\mathbf{A}}_n$, and if F is the free functor, it is a left adjoint and therefore preserves colimits. So if $\underline{\mathbf{A}}_0$ has an initial object, which can be shown to be

the case if $\underline{\mathbf{A}}_0$ is defined as described above, we get an initial object in $\underline{\mathbf{A}}_n$. One approach to obtaining the free functors F_n is using a constructive version of the adjoint functor theorem — the categories $\underline{\mathbf{A}}_n$ were already shown to be complete in [ACD+18]. This would require some variant of Adámek and Rosický's locally presentable categories [AR94], in which a large collection of objects can be approximated by a small set of objects via filtered colimits.

Appendix A

Omitted Proofs from Section 4.3.1

Associativity proofs of Proposition 4.3.9. Nearly all parts of Proposition 4.3.9 were formalised and can be found at stefaniatadama.com/distr-law-thesis-html, aside from the associativity equalities, which for technical reasons (Agda was taking too long to compile) were not formalised but are proved below instead. Sub-expressions with the same colour were rewritten by applying the same position equality.

$$
\begin{aligned}
&\sigma\ (a_1, a_2)\ (\lambda p.\sigma\ (b_1\ p, b_2\ p)\ \langle c_1\ p, c_2\ p\rangle)\\
:=&(\sigma^S\ a_1\ (\lambda y.u_1\ (a_2\ y)\ (\lambda z.\sigma^S\ (b_1\ (y,z))\ (\lambda y'.u_1\ (b_2\ (y,z)\ y')\ (\lambda z'.c_1\ (y,z)\ (y',z'))))),\\
&\lambda y.\sigma^T\\
&\qquad (u_2\ (a_2\ (\mathrm{pr}_1^S\ y))\\
&\qquad\qquad (\lambda z.\sigma^S\ (b_1\ (\mathrm{pr}_1^S\ y, z))\ (\lambda y'.u_1\ (b_2\ (\mathrm{pr}_1^S\ y, z)\ y')\ (\lambda z'.c_1\ (\mathrm{pr}_1^S\ y, z)\ (y', z'))))\\
&\qquad\qquad (\mathrm{pr}_2^S\ y))\\
&\qquad (\lambda z.\sigma^T\\
&\qquad\qquad (u_2\ (b_2\ (\mathrm{pr}_1^S\ y, v_1\ (\mathrm{pr}_2^S\ y)\ z)\ (\mathrm{pr}_1^S\ (v_2\ (\mathrm{pr}_2^S\ y)\ z)))\\
&\qquad\qquad\qquad (\lambda z'.c_1\ (\mathrm{pr}_1^S\ y, v_1\ (\mathrm{pr}_2^S\ y)\ z)\ (\mathrm{pr}_1^S\ (v_2\ (\mathrm{pr}_2^S\ y)\ z), z'))\\
&\qquad\qquad\qquad (\mathrm{pr}_2^S\ (v_2\ (\mathrm{pr}_2^S\ y)\ z)))\\
&\qquad\qquad (\lambda z'.c_2\ (\mathrm{pr}_1^S\ y, v_1\ (\mathrm{pr}_2^S\ y)\ z)\\
&\qquad\qquad\qquad (\mathrm{pr}_1^S\ (v_2\ (\mathrm{pr}_2^S\ y)\ z), v_1\ (\mathrm{pr}_2^S\ (v_2\ (\mathrm{pr}_2^S\ y)\ z))\ z')\\
&\qquad\qquad\qquad (v_2\ (\mathrm{pr}_2^S\ (v_2\ (\mathrm{pr}_2^S\ y)\ z))\ z'))))\\
=&[\sigma^T \text{ and } \mathrm{pr}^T \text{ associativity}]\\
&(\sigma^S\ a_1\ (\lambda y.u_1\ (a_2\ y)\ (\lambda z.\sigma^S\ (b_1\ (y,z))\ (\lambda y'.u_1\ (b_2\ (y,z)\ y')\ (\lambda z'.c_1\ (y,z)\ (y',z'))))),\\
&\lambda y.\sigma^T\\
&\qquad (\sigma^T
\end{aligned}
$$

$$
\begin{array}{l}
\quad (u_2\ (a_2\ (\mathrm{pr}_1^S\ y)) \\
\qquad (\lambda z.\sigma^S\ (b_1\ (\mathrm{pr}_1^S\ y, z))\ (\lambda y'.u_1\ (b_2\ (\mathrm{pr}_1^S\ y, z)\ y')\ (\lambda z'.c_1\ (\mathrm{pr}_1^S\ y, z)\ (y', z')))) \\
\qquad (\mathrm{pr}_2^S\ y)) \\
\quad (\lambda z.u_2\ (b_2\ (\mathrm{pr}_1^S\ y, v_1\ (\mathrm{pr}_2^S\ y)\ z)\ (\mathrm{pr}_1^S\ (v_2\ (\mathrm{pr}_2^S\ y)\ z))) \\
\qquad\quad (\lambda z'.c_1\ (\mathrm{pr}_1^S\ y, v_1\ (\mathrm{pr}_2^S\ y)\ z)\ (\mathrm{pr}_1^S\ (v_2\ (\mathrm{pr}_2^S\ y)\ z), z')) \\
\qquad\quad (\mathrm{pr}_2^S\ (v_2\ (\mathrm{pr}_2^S\ y)\ z))) \\
\quad (\lambda z.c_2\ (\mathrm{pr}_1^S\ y, v_1\ (\mathrm{pr}_2^S\ y)\ (\mathrm{pr}_1^T\ z)) \\
\qquad\quad (\mathrm{pr}_1^S\ (v_2\ (\mathrm{pr}_2^S\ y)\ (\mathrm{pr}_1^T\ z)), v_1\ (\mathrm{pr}_2^S\ (v_2\ (\mathrm{pr}_2^S\ y)\ (\mathrm{pr}_1^T\ z)))\ (\mathrm{pr}_2^T\ z)) \\
\qquad\quad (v_2\ (\mathrm{pr}_2^S\ (v_2\ (\mathrm{pr}_2^S\ y)\ (\mathrm{pr}_1^T\ z)))\ (\mathrm{pr}_2^T\ z))))
\end{array}
$$

=[mul-T equations]

$$
\begin{array}{l}
(\sigma^S\ a_1\ (\lambda y.\sigma^S \\
\qquad (u_1\ (a_2\ y)\ (\lambda z.b_1\ (y, z))) \\
\qquad (\lambda y'.u_1 \\
\qquad\quad (u_2\ (a_2\ y)\ (\lambda z.b_1\ (y, z))\ y') \\
\qquad\quad (\lambda z.u_1\ (b_2\ (y, v_1\ y'\ z)\ (v_2\ y'\ z))\ (\lambda z'.c_1\ (y, v_1\ y'\ z)\ (v_2\ y'\ z, z'))))), \\
\lambda y.\sigma^T \\
\quad (\sigma^T \\
\qquad (u_2\ (u_2\ (a_2\ (\mathrm{pr}_1^S\ y))\ (\lambda z.b_1\ (\mathrm{pr}_1^S\ y, z))\ (\mathrm{pr}_1^S\ (\mathrm{pr}_2^S\ y))) \\
\qquad\quad (\lambda z.u_1\ (b_2\ (\mathrm{pr}_1^S\ y, v_1\ (\mathrm{pr}_1^S\ (\mathrm{pr}_2^S\ y))\ z)\ (v_2\ (\mathrm{pr}_1^S\ (\mathrm{pr}_2^S\ y))\ z)) \\
\qquad\qquad (\lambda z'.c_1\ (\mathrm{pr}_1^S\ y, v_1\ (\mathrm{pr}_1^S\ (\mathrm{pr}_2^S\ y))\ z)\ (v_2\ (\mathrm{pr}_1^S\ (\mathrm{pr}_2^S\ y))\ z, z'))) \\
\qquad\quad (\mathrm{pr}_2^S\ (\mathrm{pr}_2^S\ y))) \\
\qquad (\lambda z.u_2\ (b_2\ (\mathrm{pr}_1^S\ y, v_1\ (\mathrm{pr}_1^S\ (\mathrm{pr}_2^S\ y))\ (v_1\ (\mathrm{pr}_2^S\ (\mathrm{pr}_2^S\ y))\ z)) \\
\qquad\qquad (v_2\ (\mathrm{pr}_1^S\ (\mathrm{pr}_2^S\ y))\ (v_1\ (\mathrm{pr}_2^S\ (\mathrm{pr}_2^S\ y))\ z))) \\
\qquad\quad (\lambda z'.c_1(\mathrm{pr}_1^S\ y, v_1\ (\mathrm{pr}_1^S\ (\mathrm{pr}_2^S\ y))\ (v_1\ (\mathrm{pr}_2^S\ (\mathrm{pr}_2^S\ y))\ z)) \\
\qquad\qquad (v_2\ (\mathrm{pr}_1^S\ (\mathrm{pr}_2^S\ y))\ (v_1\ (\mathrm{pr}_2^S\ (\mathrm{pr}_2^S\ y))\ z), z')) \\
\qquad\quad (v_2\ (\mathrm{pr}_2^S\ (\mathrm{pr}_2^S\ y))\ z))) \\
\quad (\lambda z.c_2\ (\mathrm{pr}_1^S\ y, v_1\ (\mathrm{pr}_1^S\ (\mathrm{pr}_2^S\ y))\ (v_1\ (\mathrm{pr}_2^S\ (\mathrm{pr}_2^S\ y))\ (\mathrm{pr}_1^T\ z))) \\
\qquad (v_2\ (\mathrm{pr}_1^S\ (\mathrm{pr}_2^S\ y))\ (v_1\ (\mathrm{pr}_2^S\ (\mathrm{pr}_2^S\ y))\ (\mathrm{pr}_1^T\ z)), \\
\qquad\ v_1\ (v_2\ (\mathrm{pr}_2^S\ (\mathrm{pr}_2^S\ y))\ (\mathrm{pr}_1^T\ z))\ (\mathrm{pr}_2^T\ z)) \\
\qquad (v_2\ (v_2\ (\mathrm{pr}_2^S\ (\mathrm{pr}_2^S\ y))\ (\mathrm{pr}_1^T\ z))\ (\mathrm{pr}_2^T\ z))))
\end{array}
$$

=[mul-S equations]

$$
\begin{array}{l}
(\sigma^S\ a_1\ (\lambda y.\sigma^S \\
\qquad (u_1\ (a_2\ y)\ (\lambda z.b_1\ (y, z))) \\
\qquad (\lambda y'.u_1\ (\sigma^T\ (u_2\ (a_2\ y)\ (\lambda z.b_1\ (y, z))\ y')\ (\lambda z.b_2\ (y, v_1\ y'\ z)\ (v_2\ y'\ z))) \\
\qquad\quad (\lambda z.c_1\ (y, v_1\ y'\ (\mathrm{pr}_1^T\ z))\ (v_2\ y'\ (\mathrm{pr}_1^T\ z), \mathrm{pr}_2^T\ z)))), \\
\lambda y.\sigma^T
\end{array}
$$

$$
\begin{aligned}
&(u_2\ (\sigma^T\ (u_2\ (a_2\ (\mathrm{pr}_1^S\ y))\ (\lambda z.b_1\ (\mathrm{pr}_1^S\ y, z))\ (\mathrm{pr}_1^S\ (\mathrm{pr}_2^S\ y)))\\
&\qquad (\lambda z.b_2\ (\mathrm{pr}_1^S\ y, v_1\ (\mathrm{pr}_1^S\ (\mathrm{pr}_2^S\ y))\ z)\ (v_2\ (\mathrm{pr}_1^S\ (\mathrm{pr}_2^S\ y))\ z)))\\
&\quad (\lambda z.c_1\ (\mathrm{pr}_1^S\ y, v_1\ (\mathrm{pr}_1^S\ (\mathrm{pr}_2^S\ y))\ (\mathrm{pr}_1^T\ z))\ (v_2\ (\mathrm{pr}_1^S\ (\mathrm{pr}_2^S\ y))\ (\mathrm{pr}_1^T\ z), \mathrm{pr}_2^T\ z))\\
&\quad (\mathrm{pr}_2^S\ (\mathrm{pr}_2^S\ y)))\\
&(\lambda z.c_2\ (\mathrm{pr}_1^S\ y, v_1\ (\mathrm{pr}_1^S\ (\mathrm{pr}_2^S\ y))\ (\mathrm{pr}_1^T\ (v_1\ (\mathrm{pr}_2^S\ (\mathrm{pr}_2^S\ y))\ z)))\\
&\qquad (v_2\ (\mathrm{pr}_1^S\ (\mathrm{pr}_2^S\ y))\ (\mathrm{pr}_1^T\ (v_1\ (\mathrm{pr}_2^S\ (\mathrm{pr}_2^S\ y))\ z)), \mathrm{pr}_2^T\ (v_1\ (\mathrm{pr}_2^S\ (\mathrm{pr}_2^S\ y))\ z))\\
&\qquad (v_2\ (\mathrm{pr}_2^S\ (\mathrm{pr}_2^S\ y))\ z)))\\
=&[\sigma^S \text{ and } \mathrm{pr}^S \text{ associativity equations}]\\
&(\sigma^S\ (\sigma^S\ a_1\ (\lambda y.u_1\ (a_2\ y)\ (\lambda z.b_1\ (y, z))))\\
&\quad (\lambda y.u_1\ (\sigma^T\ (u_2\ (a_2\ (\mathrm{pr}_1^S\ y))\ (\lambda z.b_1\ (\mathrm{pr}_1^S\ y, z))\ (\mathrm{pr}_2^S\ y))\\
&\qquad (\lambda z.b_2\ (\mathrm{pr}_1^S\ y, v_1\ (\mathrm{pr}_2^S\ y)\ z)\ (v_2\ (\mathrm{pr}_2^S\ y)\ z)))\\
&\qquad (\lambda z.c_1\ (\mathrm{pr}_1^S\ y, v_1\ (\mathrm{pr}_2^S\ y)\ (\mathrm{pr}_1^T\ z))\ (v_2\ (\mathrm{pr}_2^S\ y)\ (\mathrm{pr}_1^T\ z), \mathrm{pr}_2^T\ z))),\\
&\lambda y.\sigma^T\ (u_2\ (\sigma^T\ (u_2\ (a_2\ (\mathrm{pr}_1^S\ (\mathrm{pr}_1^S\ y)))\ (\lambda z.b_1\ (\mathrm{pr}_1^S\ (\mathrm{pr}_1^S\ y), z))\ (\mathrm{pr}_2^S\ (\mathrm{pr}_1^S\ y)))\\
&\qquad (\lambda z.b_2\ (\mathrm{pr}_1^S\ (\mathrm{pr}_1^S\ y), v_1\ (\mathrm{pr}_2^S\ (\mathrm{pr}_1^S\ y))\ z)\ (v_2\ (\mathrm{pr}_2^S\ (\mathrm{pr}_1^S\ y))\ z)))\\
&\quad (\lambda z.c_1\ (\mathrm{pr}_1^S\ (\mathrm{pr}_1^S\ y), v_1\ (\mathrm{pr}_2^S\ (\mathrm{pr}_1^S\ y))\ (\mathrm{pr}_1^T\ z))\\
&\qquad (v_2\ (\mathrm{pr}_2^S\ (\mathrm{pr}_1^S\ y))\ (\mathrm{pr}_1^T\ z), \mathrm{pr}_2^T\ z))\ (\mathrm{pr}_2^S\ y))\\
&\quad (\lambda z.c_2\ (\mathrm{pr}_1^S\ (\mathrm{pr}_1^S\ y), v_1\ (\mathrm{pr}_2^S\ (\mathrm{pr}_1^S\ y))\ (\mathrm{pr}_1^T\ (v_1\ (\mathrm{pr}_2^S\ y)\ z)))\\
&\qquad (v_2\ (\mathrm{pr}_2^S\ (\mathrm{pr}_1^S\ y))\ (\mathrm{pr}_1^T\ (v_1\ (\mathrm{pr}_2^S\ y)\ z)), \mathrm{pr}_2^T\ (v_1\ (\mathrm{pr}_2^S\ y)\ z))\\
&\qquad (v_2\ (\mathrm{pr}_2^S\ y)\ z)))\\
=:&\ \sigma\ (\sigma\ (a_1, a_2)\ (\lambda p.(b_1\ p, b_2\ p)))\ (\lambda p.(c_1\ (\mathrm{pr}_1\ p)\ (\mathrm{pr}_2\ p), c_2\ (\mathrm{pr}_1\ p)\ (\mathrm{pr}_2\ p)))
\end{aligned}
$$

$$
\begin{aligned}
&\mathrm{pr}_1\ (\mathrm{pr}_1\ (p, q))\\
\coloneqq\ &\mathrm{pr}_1^S\ (\mathrm{pr}_1^S\ p), v_1\ (\mathrm{pr}_2^S\ (\mathrm{pr}_1^S\ p))\ (\mathrm{pr}_1^T\ (v_1\ (\mathrm{pr}_2^S\ p)\ (\mathrm{pr}_1^T\ q)))\\
=\ &\mathrm{pr}_1^S\ p, v_1\ (\mathrm{pr}_1^S\ (\mathrm{pr}_2^S\ p))\ (\mathrm{pr}_1^T\ (v_1\ (\mathrm{pr}_2^S\ (\mathrm{pr}_2^S\ p))\ (\mathrm{pr}_1^T\ q)))\qquad [\mathrm{pr}^S \text{ assoc. eqs.}]\\
=\ &\mathrm{pr}_1^S\ p, v_1\ (\mathrm{pr}_1^S\ (\mathrm{pr}_2^S\ p))\ (v_1\ (\mathrm{pr}_2^S\ (\mathrm{pr}_2^S\ p))\ (\mathrm{pr}_1^T\ (\mathrm{pr}_1^T\ q)))\qquad [\text{mul-}S\text{-p}_1]\\
=\ &\mathrm{pr}_1^S\ p, v_1\ (\mathrm{pr}_2^S\ p)\ (\mathrm{pr}_1^T\ (\mathrm{pr}_1^T\ q))\qquad [\text{mul-}T\text{-p}_1]\\
=\ &\mathrm{pr}_1^S\ p, v_1\ (\mathrm{pr}_2^S\ p)\ (\mathrm{pr}_1^T\ q)\qquad [\mathrm{pr}^T \text{ assoc. eqs.}]\\
=:\ &\mathrm{pr}_1\ (p, q)
\end{aligned}
$$

$$
\begin{aligned}
&\mathrm{pr}_2\ (\mathrm{pr}_1\ (p, q))\\
\coloneqq\ &v_2\ (\mathrm{pr}_2^S\ (\mathrm{pr}_1^S\ p))\ (\mathrm{pr}_1^T\ (v_1\ (\mathrm{pr}_2^S\ p)\ (\mathrm{pr}_1^T\ q))), \mathrm{pr}_2^T\ (v_1\ (\mathrm{pr}_2^S\ p)\ (\mathrm{pr}_1^T\ q))\\
=\ &v_2\ (\mathrm{pr}_1^S\ (\mathrm{pr}_2^S\ p))\ (\mathrm{pr}_1^T\ (v_1\ (\mathrm{pr}_2^S\ (\mathrm{pr}_2^S\ p))\ (\mathrm{pr}_1^T\ q))),\\
&\quad \mathrm{pr}_2^T\ (v_1\ (\mathrm{pr}_2^S\ (\mathrm{pr}_2^S\ p))\ (\mathrm{pr}_1^T\ q))\qquad [\mathrm{pr}^S \text{ assoc. eqs.}]\\
=\ &v_2\ (\mathrm{pr}_1^S\ (\mathrm{pr}_2^S\ p))\ (v_1\ (\mathrm{pr}_2^S\ (\mathrm{pr}_2^S\ p))\ (\mathrm{pr}_1^T\ (\mathrm{pr}_1^T\ q))),\\
&\quad v_1\ (v_2\ (\mathrm{pr}_2^S\ (\mathrm{pr}_2^S\ p))\ (\mathrm{pr}_1^T\ (\mathrm{pr}_1^T\ q)))\ (\mathrm{pr}_2^T\ (\mathrm{pr}_1^T\ q))\qquad [\text{mul-}S\text{-p}_1, \text{mul-}S\text{-p}_{21}]\\
=\ &\mathrm{pr}_1^S\ (v_2\ (\mathrm{pr}_2^S\ p)\ (\mathrm{pr}_1^T\ (\mathrm{pr}_1^T\ q))),
\end{aligned}
$$

$$
\begin{aligned}
& v_1\ (\mathrm{pr}_2^S\ (v_2\ (\mathrm{pr}_2^S\ p)\ (\mathrm{pr}_1^T\ (\mathrm{pr}_1^T\ q))))\ (\mathrm{pr}_2^T\ (\mathrm{pr}_1^T\ q)) && [\text{mul-}T\text{-p}_{21}, \text{mul-}T\text{-p}_{22}] \\
=\ & \mathrm{pr}_1^S\ (v_2\ (\mathrm{pr}_2^S\ p)\ (\mathrm{pr}_1^T\ q)), \\
& v_1\ (\mathrm{pr}_2^S\ (v_2\ (\mathrm{pr}_2^S\ p)\ (\mathrm{pr}_1^T\ q)))\ (\mathrm{pr}_1^T\ (\mathrm{pr}_2^T\ q)) && [\mathrm{pr}^T \text{ assoc. eqs.}] \\
=:\ & \mathrm{pr}_1\ (\mathrm{pr}_2\ (p, q))
\end{aligned}
$$

$$
\begin{aligned}
& \mathrm{pr}_2\ (p, q) \\
:=\ & v_2\ (\mathrm{pr}_2^S\ p)\ (\mathrm{pr}_1^T\ q), \mathrm{pr}_2^T\ q \\
=\ & v_2\ (\mathrm{pr}_2^S\ (\mathrm{pr}_2^S\ p))\ (\mathrm{pr}_1^T\ q), \mathrm{pr}_2^T\ q && [\mathrm{pr}^S \text{ assoc. eqs.}] \\
=\ & v_2\ (v_2\ (\mathrm{pr}_2^S\ (\mathrm{pr}_2^S\ p))\ (\mathrm{pr}_1^T\ (\mathrm{pr}_1^T\ q)))\ (\mathrm{pr}_2^T\ (\mathrm{pr}_1^T\ q)), \mathrm{pr}_2^T\ q && [\text{mul-}S\text{-p}_{22}] \\
=\ & v_2\ (\mathrm{pr}_2^S\ (v_2\ (\mathrm{pr}_2^S\ p)\ (\mathrm{pr}_1^T\ (\mathrm{pr}_1^T\ q))))\ (\mathrm{pr}_2^T\ (\mathrm{pr}_1^T\ q)), \mathrm{pr}_2^T\ q && [\text{mul-}T\text{-p}_{22}] \\
=\ & v_2\ (\mathrm{pr}_2^S\ (v_2\ (\mathrm{pr}_2^S\ p)\ (\mathrm{pr}_1^T\ q)))\ (\mathrm{pr}_1^T\ (\mathrm{pr}_2^T\ q)), \mathrm{pr}_2^T\ (\mathrm{pr}_2^T\ q) && [\mathrm{pr}^T \text{ assoc. eqs.}] \\
=:\ & \mathrm{pr}_2\ (\mathrm{pr}_2\ (p, q)) && \blacktriangleleft
\end{aligned}
$$

Bibliography

[AA00] Andreas Abel and Thorsten Altenkirch. "A Predicative Strong Normalisation Proof for a λ-Calculus with Interleaving Inductive Types". In: *Types for Proofs and Programs*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 21–40. ISBN: 978-3-540-44557-9. DOI: 10.1007/3-540-44557-9_2 (cit. on p. 19).

[AAG03] Michael Abbott, Thorsten Altenkirch, and Neil Ghani. "Categories of Containers". In: *Foundations of Software Science and Computation Structures*. Ed. by Andrew D. Gordon. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 23–38. ISBN: 978-3-540-36576-1. DOI: 10.1007/3-540-36576-1_2 (cit. on pp. 2, 17, 38).

[AAG04] Michael Abbott, Thorsten Altenkirch, and Neil Ghani. "Representing Nested Inductive Types using W-types". In: *Automata, Languages and Programming, 31st International Colloqium (ICALP)*. 2004, pp. 59–71. DOI: 10.1007/978-3-540-27836-8_8 (cit. on pp. 2, 17, 38).

[AAG05] Michael Abbott, Thorsten Altenkirch, and Neil Ghani. "Containers: Constructing strictly positive types". In: *Theoretical Computer Science* 342.1 (2005). Applied Semantics: Selected Topics, pp. 3–27. ISSN: 0304-3975. DOI: 10.1016/j.tcs.2005.06.002 (cit. on pp. 2, 17, 38, 41, 48, 54–55, 57).

[Abb03] Michael Gordon Abbott. "Categories of Containers". PhD thesis. University of Leicester, 2003 (cit. on pp. 2, 17, 38).

[ABKT19] Thorsten Altenkirch, Simon Boulier, Ambrus Kaposi, and Nicolas Tabereau. "Setoid Type Theory—A Syntactic Translation". In: Springer, 2019, pp. 155–196. ISBN: 9783030336356. DOI: 10.1007/978-3-030-33636-3_7 (cit. on p. 55).

[ACD+18] Thorsten Altenkirch, Paolo Capriotti, Gabe Dijkstra, Nicolai Kraus, and Fredrik Nordvall Forsberg. "Quotient Inductive-Inductive Types". In: *Foundations of Software Science and Computation Structures*. Springer International Publishing, 2018, pp. 293–310. ISBN: 9783319893662. DOI: 10.1007/978-3-319-89366-2_16 (cit. on pp. 11, 18, 58, 86, 110–111).

[ACH19] Jeremy Avigad, Mario Carneiro, and Simon Hudon. "Data Types as Quotients of Polynomial Functors". In: *10th International Conference on Interactive Theorem Proving (ITP 2019)*. Ed. by John Harrison, John O'Leary, and Andrew Tolmach. Vol. 141. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019, 6:1–6:19. ISBN: 978-3-95977-122-1. DOI: 10.4230/LIPIcs.ITP.2019.6 (cit. on p. 55).

[ACS15] Benedikt Ahrens, Paolo Capriotti, and Régis Spadotti. "Non-Wellfounded Trees in Homotopy Type Theory". In: *13th International Conference on Typed Lambda Calculi and Applications (TLCA 2015)*. Ed. by Thorsten Altenkirch. Vol. 38. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2015, pp. 17–30. ISBN: 978-3-939897-87-3. DOI: 10.4230/LIPIcs.TLCA.2015.17 (cit. on p. 55).

[ACU12] Danel Ahman, James Chapman, and Tarmo Uustalu. "When Is a Container a Comonad?" In: *Foundations of Software Science and Computational Structures*. Ed. by Lars Birkedal. Berlin, Heidelberg: Springer, 2012, pp. 74–88. ISBN: 978-3-642-28729-9. DOI: 10.1007/978-3-642-28729-9_5 (cit. on pp. 91–92, 95, 107).

[AFM+21] Benedikt Ahrens, Dan Frumin, Marco Maggesi, Niccolò Veltri, and Niels van der Weide. "Bicategories in univalent foundations". In: *Mathematical Structures in Computer Science* 31.10 (2021), pp. 1232–1269. DOI: 10.1017/S0960129522000032 (cit. on p. 63).

[Agd25a] The Agda Community. *Cubical Agda Library*. Version 0.7. 2025. URL: https://github.com/agda/cubical (cit. on pp. 39, 91).

[Agd25b] The Agda Development Team. *Agda's documentation: Built-ins*. Version 2.6.1. 2025. URL: https://agda.readthedocs.io/en/v2.6.1/language/built-ins.html#equality (cit. on p. 9).

[Agd25c] The Agda Development Team. *The Agda Programming Language*. 2025. URL: https://wiki.portal.chalmers.se/agda/pmwiki.php (cit. on p. 8).

[AGH+15] Thorsten Altenkirch, Neil Ghani, Peter Hancock, Conor McBride, and Peter Morris. "Indexed containers".
In: *Journal of Functional Programming* 25 (2015), e5.
DOI: 10.1017/S095679681500009X (cit. on pp. 2, 9, 17, 19, 33).

[AGS12] Steve Awodey, Nicola Gambino, and Kristina Sojakova.
"Inductive Types in Homotopy Type Theory". In: *Proceedings of the 2012 27th Annual IEEE/ACM Symposium on Logic in Computer Science*. LICS '12.
New Orleans, Louisiana: IEEE Computer Society, 2012, pp. 95–104.
ISBN: 9780769547695. DOI: 10.1109/LICS.2012.21 (cit. on p. 58).

[Ahm11] Danel Ahman. *An Agda formalisation of the theory of directed containers*.
https://github.com/danelahman/Directed-Containers. Online. 2011 (cit. on p. 92).

[AK16] Thorsten Altenkirch and Ambrus Kaposi.
"Type theory in type theory using quotient inductive types".
In: *SIGPLAN Not.* 51.1 (Jan. 2016), pp. 18–29. ISSN: 0362-1340.
DOI: 10.1145/2914770.2837638 (cit. on pp. 11, 59).

[AK21] Thorsten Altenkirch and Ambrus Kaposi.
A container model of type theory. Talk abstract at TYPES, available at
https://types21.liacs.nl/download/a-container-model-of-type-theory/. 2021 (cit. on pp. 2–3, 58, 66).

[AKX26] Thorsten Altenkirch, Ambrus Kaposi, and Szumi Xie.
"The Groupoid-Syntax of Type Theory Is a Set".
In: *34th EACSL Annual Conference on Computer Science Logic (CSL 2026)*.
Ed. by Stefano Guerrini and Barbara König. Vol. 363.
Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2026, 40:1–40:23. ISBN: 978-3-95977-411-6.
DOI: 10.4230/LIPIcs.CSL.2026.40 (cit. on pp. 3, 62).

[Alt20] Thorsten Altenkirch. *Cubical primitives should preserve guardedness*.
https://github.com/agda/agda/issues/4740. Online.
2020 (cit. on p. 54).

[Alt24] Thorsten Altenkirch. *Quotient inductive types as categorified containers*.
https://hott-uf.github.io/2024/abstracts/HoTTUF_2024_paper_10.pdf. HoTT/UF 2024 abstract. 2024
(cit. on pp. 55, 107).

[Alt25] Thorsten Altenkirch. *Containers in Higher Kinds*.
Talk abstract at WG6 meeting 2025, available at https://europroofnet.github.io/wg6-genoa/programme. 2025
(cit. on p. 37).

[AM09] Thorsten Altenkirch and Peter Morris. "Indexed Containers".
In: *2009 24th Annual IEEE Symposium on Logic In Computer Science*. 2009, pp. 277–285. DOI: 10.1109/LICS.2009.33
(cit. on pp. 2, 9, 17, 19, 33, 57).

[AP17] Thorsten Altenkirch and Gun Pinyo. *Monadic containers and Σ-universes*. TYPES 2017 abstract. 2017.
URL: https://types2017.elte.hu/proc.pdf#page=28
(cit. on pp. 4, 91–93, 101).

[AR94] J. Adamek and J. Rosicky. *Locally Presentable and Accessible Categories*. London Mathematical Society Lecture Note Series.
Cambridge University Press, 1994 (cit. on p. 111).

[Atk20] Robert Atkey. "Interpreting dependent types with containers".
Talk at the MSP101 seminar, University of Strathclyde. 2020.
URL: https://gist.github.com/bobatkey/0d1f04057939905d35699f1b1c323736. (cit. on p. 58).

[AU13] Danel Ahman and Tarmo Uustalu.
"Distributive laws of directed containers".
In: *Progress in Informatics* 10 (Mar. 2013), pp. 3–18. ISSN: 1349-8614.
DOI: 10.2201/NiiPi.2013.10.2 (cit. on pp. 4, 91, 98, 101, 107).

[Awo18] Steve Awodey. "Natural models of homotopy type theory".
In: *Math. Struct. Comput. Sci.* 28.2 (2018), pp. 241–286.
DOI: 10.1017/S0960129516000268 (cit. on pp. 62, 91).

[Bec69] Jon Beck. "Distributive laws".
In: *Seminar on Triples and Categorical Homology Theory*.
Ed. by B. Eckmann. Berlin, Heidelberg: Springer Berlin Heidelberg, 1969, pp. 119–140. ISBN: 978-3-540-36091-9. DOI: 10.1007/BFb0083084
(cit. on pp. 4, 90, 95, 99).

[Bri05] Matthew G. Brin. "On the Zappa-Szép Product".
In: *Communications in Algebra* 33.2 (Feb. 2005), pp. 393–424.
ISSN: 1532-4125. DOI: 10.1081/agb-200047404 (cit. on p. 98).

[Bru16] Guillaume Brunerie.
"On the homotopy groups of spheres in homotopy type theory".
PhD thesis. 2016. arXiv: 1606.05916 [math.AT] (cit. on p. 56).

[Bun94] Alexander Bunkenburg. "The Boom Hierarchy". In:
Functional Programming, Glasgow 1993: Proceedings of the 1993 Glasgow Workshop on Functional Programming, Ayr, Scotland, 5–7 July 1993.
Ed. by John T. O'Donnell and Kevin Hammond.
London: Springer London, 1994, pp. 1–8. ISBN: 978-1-4471-3236-3.
DOI: 10.1007/978-1-4471-3236-3_1 (cit. on p. 91).

[Car86] John Cartmell. "Generalised algebraic theories and contextual categories".
In: *Annals of Pure and Applied Logic* 32 (1986), pp. 209–243.
ISSN: 0168-0072. DOI: 10.1016/0168-0072(86)90053-9
(cit. on p. 59).

[CCHM18] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. "Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom". In: *21st International Conference on Types for Proofs and Programs (TYPES 2015)*. Ed. by Tarmo Uustalu. Vol. 69. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018, 5:1–5:34. ISBN: 978-3-95977-030-9. DOI: 10.4230/LIPIcs.TYPES.2015.5 (cit. on pp. 39–40, 91).

[CH19] Evan Cavallo and Robert Harper. "Higher inductive types in cubical computational type theory". In: *Proc. ACM Program. Lang.* 3.POPL (Jan. 2019). DOI: 10.1145/3290314 (cit. on p. 56).

[Che25] Joshua Chen. *2-Coherent Internal Models of Homotopical Type Theory*. 2025. arXiv: 2503.05790 [cs.LO] (cit. on pp. 3, 43, 62).

[CHM18] Thierry Coquand, Simon Huber, and Anders Mörtberg. "On Higher Inductive Types in Cubical Type Theory". In: *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*. LICS '18. Oxford, United Kingdom: Association for Computing Machinery, 2018, pp. 255–264. ISBN: 9781450355834. DOI: 10.1145/3209108.3209197 (cit. on pp. 39–40, 56).

[CJ95] Aurelio Carboni and Peter Johnstone. "Connected limits, familial representability and Artin glueing". In: *Mathematical Structures in Computer Science* 5.4 (1995), pp. 441–459. DOI: 10.1017/S0960129500001183 (cit. on p. 34).

[DAL25a] Stefania Damato, Thorsten Altenkirch, and Axel Ljungström. *Formalising Inductive and Coinductive Containers*. Accompanying formalisation. June 2025. URL: https://github.com/aljungstrom/containers/blob/main/cubical/Cubical/Papers/Containers.agda (cit. on p. 39).

[DAL25b] Stefania Damato, Thorsten Altenkirch, and Axel Ljungström. "Formalising Inductive and Coinductive Containers". In: *16th International Conference on Interactive Theorem Proving (ITP 2025)*. Ed. by Yannick Forster and Chantal Keller. Vol. 352. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025, 17:1–17:20. ISBN: 978-3-95977-396-6. DOI: 10.4230/LIPIcs.ITP.2025.17 (cit. on pp. 2, 39).

[Dam20] Stefania Damato. *Constructing Simple and Mutual Inductive Types*. Master's thesis. 2020. URL: https://stefaniatadama.com/writings/msc_dissertation.pdf (cit. on p. 1).

[Dam22] Stefania Damato. *Investigating Quotient Inductive-Inductive Types*. First year PhD review report. 2022. URL: https://stefaniatadama.com/writings/first_year_report.pdf (cit. on p. 1).

[Dam25] Stefania Damato. *The Groupoid Category with Families of Containers*. Accompanying formalisation. 2025. URL: https://github.com/stefaniatadama/container-gcwf (cit. on pp. 66, 84–85).

[Dyb96a] Peter Dybjer. "Internal type theory". In: *Types for Proofs and Programs*. Ed. by Stefano Berardi and Mario Coppo. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 120–134 (cit. on p. 59).

[Dyb96b] Peter Dybjer. *Representing Inductively Defined Sets by Wellorderings in Martin-Löf's Type Theory*. 1996 (cit. on p. 57).

[GH04] Nicola Gambino and Martin Hyland. "Wellfounded Trees and Dependent Polynomial Functors". In: *Types for Proofs and Programs*. Ed. by Stefano Berardi, Mario Coppo, and Ferruccio Damiani. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 210–225. ISBN: 978-3-540-24849-1. DOI: 10.1007/978-3-540-24849-1_14 (cit. on p. 17).

[GK13] Nicola Gambino and Joachim Kock. "Polynomial functors and polynomial monads". In: *Mathematical Proceedings of the Cambridge Philosophical Society* 154.1 (Jan. 2013), pp. 153–192. ISSN: 0305-0041, 1469-8064. DOI: 10.1017/S0305004112000394 (cit. on p. 91).

[Gyl11] Håkon Robbestad Gylterud. *Symmetric Containers*. Master's thesis. 2011 (cit. on pp. 37, 55, 107).

[Hag87] Tatsuya Hagino. "A Categorical Programming Language". PhD thesis. University of Edinburgh, 1987 (cit. on p. 18).

[Hof94] Martin Hofmann. "On the Interpretation of Type Theory in Locally Cartesian Closed Categories". In: *Selected Papers from the 8th International Workshop on Computer Science Logic*. CSL '94. Berlin, Heidelberg: Springer-Verlag, 1994, pp. 427–441. ISBN: 3540600175 (cit. on p. 38).

[Hof95] Martin Hofmann. "Extensional concepts in intensional type theory". PhD thesis. University of Edinburgh, 1995. URL: https://www.lfcs.inf.ed.ac.uk/reports/95/ECS-LFCS-95-327/ (cit. on p. 7).

[Hof97] Martin Hofmann. "Syntax and semantics of dependent types". In: *Extensional Constructs in Intensional Type Theory*. London: Springer London, 1997, pp. 13–54. ISBN: 978-1-4471-0963-1. DOI: 10.1007/978-1-4471-0963-1_2 (cit. on p. 59).

[HPW+92] Paul Hudak, Simon Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph Fasel, María M. Guzmán, et al. "Report on the programming language Haskell: a non-strict, purely functional language version 1.2". In: *SIGPLAN Not.* 27.5 (May 1992), pp. 1–164. ISSN: 0362-1340. DOI: 10.1145/130697.130699 (cit. on p. 90).

[Hug21] Jasper Hugunin. "Why Not W?" In: *26th International Conference on Types for Proofs and Programs (TYPES 2020)*. Ed. by Ugo de'Liguoro, Stefano Berardi, and Thorsten Altenkirch. Vol. 188. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, 8:1–8:9. ISBN: 978-3-95977-182-5. DOI: 10.4230/LIPIcs.TYPES.2020.8 (cit. on p. 57).

[Jac93] Bart Jacobs. "Comprehension categories and the semantics of type dependency". In: *Theoretical Computer Science* 107.2 (1993), pp. 169–207. ISSN: 0304-3975. DOI: https://doi.org/10.1016/0304-3975(93)90169-T (cit. on p. 59).

[JV20] Ranjit Jhala and Niki Vazou. *Refinement Types: A Tutorial*. 2020. arXiv: 2010.07763 [cs.PL] (cit. on p. 101).

[Kap13] Ambrus Kaposi. "First year report". An introduction to type theory with a notation using De Bruijn indices and explicit substitutions. 2013. URL: https://akaposi.github.io/fstyearreport.pdf (cit. on p. 59).

[KKA19] Ambrus Kaposi, András Kovács, and Thorsten Altenkirch. "Constructing Quotient Inductive-Inductive Types". In: *Proc. ACM Program. Lang.* 3.POPL (2019). DOI: 10.1145/3290315 (cit. on p. 89).

[Koc09] Joachim Kock. *Notes on Polynomial Functors*. en. Unpublished notes. 2009 (cit. on p. 92).

[Kov20] András Kovács. "The container model of type theory." Talk at the type theory seminar, Eötvös Loránd University. 2020. URL: https://bitbucket.org/akaposi/tipuselmelet/src/master/notes/seminar20200623_container.agda (cit. on p. 58).

[Kov22] András Kovács. "Type-Theoretic Signatures for Algebraic Theories and Inductive Types". en. PhD thesis. Eötvös Loránd University, 2022 (cit. on p. 58).

[Kra21] Nicolai Kraus. "Internal ∞-Categorical Models of Dependent Type Theory : Towards 2LTT Eating HoTT". In: *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. 2021, pp. 1–14. DOI: 10.1109/LICS52264.2021.9470667 (cit. on p. 62).

[KS24] Amin Karamlou and Nihil Shah. "No Go Theorems: Directed Containers That Do Not Distribute Over Distribution Monads". In: *Proceedings of the 39th Annual ACM/IEEE Symposium on Logic in Computer Science*. LICS '24. New York, NY, USA: Association for Computing Machinery, July 2024, pp. 1–13. DOI: 10.1145/3661814.3662137 (cit. on pp. 90, 92).

[KvR21] Nicolai Kraus and Jakob von Raumer. "Path spaces of higher inductive types in homotopy type theory". In: *Proceedings of the 34th Annual ACM/IEEE Symposium on Logic in Computer Science*. LICS '19. Vancouver, Canada: IEEE Press, 2021 (cit. on p. 43).

[LM23] Axel Ljungström and Anders Mörtberg. "Formalizing π4(S3) ≅Z/2Z and Computing a Brunerie Number in Cubical Agda". In: *2023 38th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. 2023, pp. 1–13. DOI: 10.1109/LICS56636.2023.10175833 (cit. on p. 56).

[LS13] Daniel R. Licata and Michael Shulman. *Calculating the Fundamental Group of the Circle in Homotopy Type Theory*. 2013. arXiv: 1301.3443 [math.LO] (cit. on p. 56).

[LS19] Peter LeFanu Lumsdaine and Michael Shulman. "Semantics of higher inductive types". In: *Mathematical Proceedings of the Cambridge Philosophical Society* 169.1 (June 2019), pp. 159–208. DOI: 10.1017/s030500411900015x (cit. on pp. 40, 56).

[Man12] E.G. Manes. *Algebraic Theories*. Springer Science & Business Media, 2012. ISBN: 9781461298601. DOI: 10.1007/978-1-4612-9860-1 (cit. on p. 107).

[Mar72] Per Martin-Löf. "An Intuitionistic Theory of Types". In: *Twenty-Five Years of Constructive Type Theory* (1972) (cit. on p. 9).

[Mar82] Per Martin-Löf. "Constructive Mathematics and Computer Programming". In: *Logic, Methodology and Philosophy of Science VI*. Ed. by L. Jonathan Cohen, Jerzy Łoś, Helmut Pfeiffer, and Klaus-Peter Podewski. Vol. 104. Studies in Logic and the Foundations of Mathematics. Elsevier, 1982, pp. 153–175. DOI: 10.1016/S0049-237X(09)70189-2 (cit. on p. 20).

[Mar84] Per Martin-Löf. *Intuitionistic type theory*. Notes by Giovanni Sambin, available at https://archive-pml.github.io/martin-lof/pdfs/Bibliopolis-Book-1984.pdf. 1984 (cit. on p. 20).

[McB10] Conor McBride. *W-types: good news and bad news*. Online. Online blog post. 2010. URL: https://mazzo.li/epilogue/index.html%3Fp=324.html (cit. on p. 57).

[MM07] Ernie Manes and Philip Mulry. “Monad compositions I: General constructions and recursive distributive laws.” eng. In: *Theory and Applications of Categories* 18 (2007), pp. 172–208. ISSN: 1201-561X (cit. on pp. 90–91).

[MM08] Ernie Manes and Philip Mulry. “Monad compositions II: Kleisli strength”. en. In: *Mathematical Structures in Computer Science* 18.3 (June 2008), pp. 613–643. ISSN: 1469-8072, 0960-1295. DOI: 10.1017/S0960129508006695 (cit. on pp. 90–91).

[Mog91] Eugenio Moggi. “Notions of computation and monads”. In: *Information and Computation* 93.1 (1991). Selections from 1989 IEEE Symposium on Logic in Computer Science, pp. 55–92. ISSN: 0890-5401. DOI: 10.1016/0890-5401(91)90052-4 (cit. on p. 90).

[Nor13] Fredrik Nordvall Forsberg. “Inductive-inductive definitions”. PhD thesis. Swansea University, 2013 (cit. on p. 10).

[Pau93] Christine Paulin-Mohring. “Inductive definitions in the system Coq rules and properties”. In: *Typed Lambda Calculi and Applications*. Ed. by Marc Bezem and Jan Friso Groote. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 328–345. ISBN: 978-3-540-47586-6 (cit. on p. 7).

[PD25a] Chris Purdy and Stefania Damato. *Distributive Laws of Monadic Containers*. Accompanying formalisation. 2025. URL: https://github.com/chrisjpurdy/distr-laws-of-monadic-containers (cit. on p. 91).

[PD25b] Chris Purdy and Stefania Damato. “Distributive Laws of Monadic Containers”. In: *11th Conference on Algebra and Coalgebra in Computer Science (CALCO 2025)*. Ed. by Corina Cîrstea and Alexander Knapp. Vol. 342. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025, 4:1–4:22. ISBN: 978-3-95977-383-6. DOI: 10.4230/LIPIcs.CALCO.2025.4 (cit. on p. 4).

[Rie16] Emily Riehl. *Category Theory in Context*. Dover, 2016 (cit. on p. 26).

[Rij25] Egbert Rijke. *Introduction to Homotopy Type Theory*. Cambridge Studies in Advanced Mathematics, 2025 (cit. on p. 70).

[Sat15] Christian Sattler. “On relating indexed W-types with ordinary ones”. In: *21st International Conference on Types for Proofs and Programs (TYPES 2015)*. Ed. by Tarmo Uustalu. 2015, pp. 71–72. ISBN: 978-9949-430-86-4. URL: https://drive.google.com/file/d/1eqdyodN9fbZi03Eaz3pgBQdke7lb7MoM/view (cit. on p. 58).

[See84] R.A.G. Seely. "Locally cartesian closed categories and type theory". In: *Mathematical Proceedings of the Cambridge Philosophical Society* 95 (1984), pp. 33–48 (cit. on p. 38).

[Ses23] Filippo Sestini. "Bootstrapping Extensionality". PhD thesis. University of Nottingham, 2023 (cit. on p. 57).

[Tro91] A.S. Troelstra. *History of Constructivism in the Twentieth Century*. Jan. 1991. URL: https://eprints.illc.uva.nl/1307/ (cit. on p. 6).

[Uem22] Taichi Uemura. *Normalization and coherence for ∞-type theories*. 2022. arXiv: 2212.11764 [math.LO] (cit. on p. 62).

[Uni13] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study: https://homotopytypetheory.org/book, 2013 (cit. on pp. 11–12, 16, 18–19, 45, 61).

[Uus17] Tarmo Uustalu. "Container Combinatorics: Monads and Lax Monoidal Functors". en. In: *Topics in Theoretical Computer Science*. Ed. by Mohammad Reza Mousavi and Jiří Sgall. Cham: Springer International Publishing, 2017, pp. 91–105. ISBN: 978-3-319-68953-1. DOI: 10.1007/978-3-319-68953-1_8 (cit. on pp. 90–92, 99, 107).

[vD07] Benno van den Berg and Federico De Marchi. "Non-well-founded trees in categories". In: *Annals of Pure and Applied Logic* 146.1 (2007), pp. 40–59. ISSN: 0168-0072. DOI: https://doi.org/10.1016/j.apal.2006.12.001 (cit. on p. 41).

[vdWei25] Niels van der Weide. *The internal languages of univalent categories*. 2025. arXiv: 2411.06636 [math.CT] (cit. on p. 62).

[Vez17] Andrea Vezzosi. *Streams for Cubical Type Theory*. Unpublished note, available at https://saizan.github.io/streams-ctt.pdf. 2017 (cit. on p. 55).

[vGle15] Tamara von Glehn. "Polynomials and models of type theory". PhD thesis. University of Cambridge, 2015 (cit. on pp. 58, 110).

[VMA21] Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. "Cubical Agda: A dependently typed programming language with univalence and higher inductive types". In: *Journal of Functional Programming* 31 (2021), e8. DOI: 10.1017/S0956796821000034 (cit. on pp. 39–40, 91).

[Voe14] Vladimir Voevodsky. *The equivalence axiom and univalent models of type theory. (Talk at CMU on February 4, 2010)*. 2014. arXiv: 1402.5556 [math.LO] (cit. on p. 40).

[vRau19] Jakob von Raumer. "Higher Inductive Types, Inductive Families, and Inductive-Inductive Types". PhD thesis. University of Nottingham, 2019. URL: https://von-raumer.de/phd_vonraumer.pdf (cit. on p. 57).

[Xie19] Szumi Xie. *tt-in-tt*. https://bitbucket.org/szumixie/tt-in-tt/src/master/Cubical/Syntax.agda. A reduction from any inductive-inductive type to one with only two sorts; see https://lists.chalmers.se/pipermail/agda/2019/011176.html. 2019 (cit. on p. 10).

[ZM22] Maaike Zwart and Dan Marsden. "No-Go Theorems for Distributive Laws". en. In: *Logical Methods in Computer Science* Volume 18, Issue 1 (Jan. 2022), p. 6253. ISSN: 1860-5974. DOI: 10.46298/lmcs-18(1:13)2022 (cit. on pp. 90–91, 95, 104–105).

[Zwa20] Maaike Zwart. "On the Non-Compositionality of Monads via Distributive Laws". PhD thesis. University of Oxford, 2020 (cit. on p. 91).

Acronyms

CwF	category with families, 59
GCwF	groupoid category with families, 65
HIT	higher inductive type, 7
HoTT	homotopy type theory, 7
LCCC	locally cartesian closed category, 17
MLTT	Martin-Löf type theory, 4
UIP	uniqueness of identity proofs, 7
ZFC	Zermelo-Fraenkel w/ choice, 4

Index