Formalising Inductive and Coinductive Containers

Stefania Damato
School of Computer Science, University of Nottingham, UK

Thorsten Altenkirch
School of Computer Science, University of Nottingham, UK

Axel Ljungström
Department of Mathematics, Stockholm University, Sweden

Abstract

Containers capture the concept of strictly positive data types in programming. The original development of containers is done in the internal language of locally cartesian closed categories (LCCCs) with disjoint coproducts and W-types, and uniqueness of identity proofs (UIP) is implicitly assumed throughout. Although it is claimed that these developments can also be interpreted in extensional Martin-Löf type theory, this interpretation is not made explicit. In this paper, we present a formalisation of the results that 'containers preserve least and greatest fixed points' in `Cubical Agda`, thereby giving a formulation in intensional type theory. Our proofs do not make use of UIP and thereby generalise the original results from talking about container functors on Set to container functors on the wild category of types. Our main incentive for using `Cubical Agda` is that its path type restores the equivalence between bisimulation and coinductive equality. Thus, besides developing container theory in a more general setting, we also demonstrate the usefulness of `Cubical Agda`'s path type to coinductive proofs.

2012 ACM Subject Classification Theory of computation → Type theory

Keywords and phrases type theory; container; initial algebra; terminal coalgebra; Cubical Agda

Funding *Axel Ljungström*: This author is supported by the Swedish Research Council (Vetenskapsrådet) under Grant No. 2019-04545.

Acknowledgements The authors would like to thank Anders Mörtberg, the anonymous reviewers, and the Agda Zulip and Discord communities for useful comments and discussions.

1 Introduction

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) [21], with examples including the natural numbers, lists, and finite sets. In order for our inductive definitions to 'make sense', meaning their induction principle can be properly expressed without leading to inconsistencies (see [26, Section 5.6]), we need to impose conditions on the general form of a constructor. The condition we would like to impose on our inductive definitions is that they are *strictly positive*. 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, and examples include the conatural numbers and streams. As is the case for inductive definitions, coinductive definitions also ought to be strictly positive in order to avoid inconsistencies.[1] To ensure our type systems only admit such types, we seek a semantic description of strict positivity — this is precisely what containers provide.

[1] To be precise, in both cases we mean that the type's signature functor should be strictly positive.

The theory of containers [1–4] (also referred to as polynomial functors in the literature [15]) was developed to capture the concept of strictly positive data types in programming, and has been very useful in providing semantics for inductive types and inductive families. The original development of containers [1–4] uses a categorical language, where containers are presented as constructions in the internal language of locally cartesian closed categories (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 [3, Section 2.1] that these developments can also be interpreted as constructions in extensional MLTT. While Seely, Hofmann, and others show that this translation can be done in principle [17, 24], the construction in type theory is not actually carried out.

In this paper, we remedy this by providing a formalisation of existing results by Abbott et al. [3]. While the aforementioned paper implicitly assumes uniqueness of identity proofs (UIP) throughout, we do not assume this principle for any of the types involved in our formalisation. More precisely, the main contributions of this paper are:

- the formalisation of the results stating 'container functors preserve initial algebras' and 'container functors preserve terminal coalgebras' (Theorems 13 and 14), and
- the generalisation of these results by proving them not in the category of sets, but in the wild category of types.

The formalisation is written in `Cubical Agda` [29], an extension of the `Agda` proof assistant which implements (a cubical [12] flavour of) homotopy type theory (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 [25], 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 [14] 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 have taken some liberties concerning the typesetting in this paper and, although it should still be easy to follow, the syntax of the formalisation may differ slightly from the paper.

The paper is organised as follows. In Section 2, we give a brief overview of `Agda` and `Cubical Agda`, W- and M-types, and some container theory. In Section 3 we motivate and develop the constructions to be used in the main proofs, which are then given in Section 4. Lastly, we conclude in Section 5 with some related work and future avenues of study.

2 Background

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

2.1 Agda and Cubical Agda

Agda is a dependently typed functional programming language and proof assistant. It supports inductive data types, e.g. the unit type, the empty type, the natural numbers

```
data ⊤ : Type where
  tt : ⊤
```

```
data ⊥ : Type where
```

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

and record types, e.g. Σ-types and (coinductive) 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 data types, and dually supports copattern matching on record and coinductive data types, as shown below.

```
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)
```

We note that Agda uses the syntax $(a : A) \to B\ a$ rather than $\Pi_{a:A} B\ a$; we will use both notations interchangeably throughout the paper. We also remark that the syntax $\{a : A\} \to B\ a$ is available in Agda and denotes the same construction but with a an implicit argument.

Cubical Agda [29] extends Agda with primitives from cubical type theory [12, 13]. While the original motivation behind this extension was arguably to allow for native support for Voevodsky's univalence axiom [30] and higher inductive types [20], 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.

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 operations, such as a 'reversal' operation $\sim$: I → I, which allow 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 : I → A such that p i0 is definitionally equal to x and p i1 is definitionally equal to y. To showcase some elementary constructions of paths in Cubical Agda, consider e.g. the constructions/proofs of reflexivity and symmetry.

```
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 $(i :$ I$) \to A\ i$ instead of only non-dependent ones I → A. In this paper, we adopt an informal approach and write $a \approxeq b$ 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. However, it will be included in a separate text box for the interested reader, although it can generally be ignored by the reader focusing on the broader picture.

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 $a \approxeq b$ 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 by definition* pointwise equality. In a similar way, and especially important for us, the equality type of a coinductive type *is* bisimulation, modulo copattern matching. We can define id below by first introducing the path variable i, after which we are required to construct an element of Stream A 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, which then allows us to prove path≃bisim stated below [29], where $\simeq$ denotes equivalence (and if we assume univalence, which we do not require for this paper, 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

path≃bisim : ∀ {xs ys : Stream A} →
             (xs ≡ ys) ≃ (xs ≈ ys)
```

2.2 The W-type and the M-type

The type W, due to Martin-Löf [22, 23], 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 : Type and P: $S \to$ Type. 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. The key property of W is that it is the universal type for strictly positive inductive types, i.e. any strictly positive inductive type can be expressed using W.

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

▶ **Example 1.** We encode the type of natural numbers ℕ by defining S and P as below (where $A \uplus B$ is the sum type of A and B with constructors inl and inr).

```
      S = ⊤ ⊎ ⊤
P (inl _) = ⊥
P (inr _) = ⊤
```

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

$$\begin{tikzcd} \mathsf{inl\ tt} \end{tikzcd} \qquad \begin{tikzcd} \mathsf{inr\ tt} \arrow[dr] & \\ & \mathsf{inl\ tt} \end{tikzcd}$$

The type M, first studied by Abbott et al. [3] and van den Berg and De Marchi [27], is the type of non-well-founded, labelled trees. A tree of type M can have both finite and infinite paths. M takes two parameters S : Type and P: $S \rightarrow$ Type, and we think of them similarly as for W. 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.** If we define S and P as in Example 1, then M S P is an encoding of the conatural numbers ℕ∞.

```
record ℕ∞ : Type where
  coinductive
  field
    pred∞ : Maybe 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.

$$\begin{tikzcd} \mathsf{inr\ tt} \arrow[loop right] \end{tikzcd}$$

We will see in Section 4 that it is useful to have an explicit account of the coinduction principle of M, which states that any two m_0, m_1 : 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 ($\lambda\ i \rightarrow Q$ (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
```

$$\begin{array}{l}\mathsf{pos}\ (\mathsf{MCoind}\ R\ \textit{is-bisim}\ \{m_0 = m_0\}\{m_1 = m_1\}\ r\ i)\ q = \\ \quad \mathsf{MCoind}\ R\ \textit{is-bisim}\ \{m_0 = \mathsf{pos}\ m_0\ \mathsf{q}_0\}\ \{m_1 = \mathsf{pos}\ m_1\ \mathsf{q}_1\}\ (\mathsf{p\text{-}eq}\ (\textit{is-bisim}\ r)\ \mathsf{q}_0\ \mathsf{q}_1\ \mathsf{q}_2)\ i\end{array}$$

Above, q_0, q_1, and q_2 are all of the form $\mathsf{transport} \ldots q$. The constructions of these transports use some rather technical cube algebra; we omit the details and refer the interested reader to the formalisation.

2.3 Containers

The container and container functor definitions in this section are adapted from [3] to the wild category of types Type.[2] A wild category is a precategory as in the HoTT book [26], 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 [11]. We observe that Type has triangle and pentagon coherators, and is therefore a 2-coherent category in the sense of [11], and moreover its coherators are trivial [11, 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 [18, Definition 5].

▶ **Definition 3.** *A <u>(unary) container</u> is given by a pair of types $S : \mathsf{Type}$ and $P : S \to \mathsf{Type}$, which we write as $S \lhd P$.*

In practice, many data types are parameterised by one or more types. For example, $\mathsf{List}\ (A : \mathsf{Type}) : \mathsf{Type}$ and $\mathsf{Vec}\ (A : \mathsf{Type}) : \mathbb{N} \to \mathsf{Type}$ 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 parameterised by some (potentially infinite) indexing type I. We call these I-ary containers.

▶ **Definition 4.** *An <u>I-ary container</u> is given by a pair $S : \mathsf{Type}$ and $\mathbf{P} : I \to S \to \mathsf{Type}$, which we write as $S \lhd \mathbf{P}$.*

Above, $\mathbf{P}$ can be thought of as I 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 5.** Given a type A, the (unary) container representation of $\mathsf{List}\ A : \mathsf{Type}$ is given by $\mathbb{N} \lhd \mathsf{Fin}$.[3] 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.

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 $S \lhd \mathbf{P}$, 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 6.** *The <u>container functor</u> associated to an I-ary container $S \lhd \mathbf{P}$ is the wild functor $[\![S \lhd \mathbf{P}]\!] : (I \to \mathsf{Type}) \to \mathsf{Type}$ with the following actions on objects and morphisms.*[4]

- *Given $\mathbf{X} : I \to \mathsf{Type}$, we define $[\![S \lhd \mathbf{P}]\!]\,\mathbf{X} :\equiv \sum_{s:S} \Big(\prod_{i:I} \mathbf{P}\,i\,s \to \mathbf{X}\,i\Big)$.*

[2] We note that we use Type to refer to both the wild category of types as well as `Cubical Agda`'s universe of types.

[3] The precise meaning of this is that $[\![\mathbb{N} \lhd \mathsf{Fin}]\!]\,A \cong \mu X.F_{\mathsf{List}}(A, X)$, see Example 7.

[4] Here, $I \to \mathsf{Type}$ refers to the wild category of I-indexed families of types.

- *Given* $\mathbf{X}, \mathbf{Y} : I \to \mathsf{Type}$*, and a morphism* $f : \prod_{i:I} \mathbf{X}\,i \to \mathbf{Y}\,i$*, we define*

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

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

As a special case of Definition 6, given an $(I+1)$-ary container $F = S \lhd \mathbf{R}$, 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 $F = (S \lhd \mathbf{P}, Q)$. Given $\mathbf{X} : I \to \mathsf{Type}$ and $Y : \mathsf{Type}$, then $S, \mathbf{P}$, and Q satisfy the below.

$$[\![S \lhd \mathbf{P}, Q]\!](\mathbf{X}, Y) = \sum_{s:S} \Big((i : I) \to \mathbf{P}\,i\,s \to \mathbf{X}\,i\Big) \times (Q\,s \to Y) \tag{1}$$

▶ **Example 7.** The signature functor of List, $F_{\mathsf{List}}\,(A, X) = \top \uplus (A \times X)$, is isomorphic to the container functor $[\![\mathsf{S} \lhd \mathsf{P}_0, \mathsf{P}_1]\!]\,(A, X)$ via the below definitions.

$\mathsf{S} : \mathsf{Type}$	$\mathsf{P}_0 : \mathsf{S} \to \mathsf{Type}$	$\mathsf{P}_1 : \mathsf{S} \to \mathsf{Type}$
$\mathsf{S} :\equiv \top \uplus \top$	$\mathsf{P}_0\ (\mathsf{inl}\ \mathsf{tt}) :\equiv \bot$	$\mathsf{P}_1\ (\mathsf{inl}\ \mathsf{tt}) :\equiv \bot$
	$\mathsf{P}_0\ (\mathsf{inr}\ \mathsf{tt}) :\equiv \top$	$\mathsf{P}_1\ (\mathsf{inr}\ \mathsf{tt}) :\equiv \top$

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. Example 5 corresponds to taking the least fixed point of this functor with respect to X.

▶ **Example 8.** Container functors allow us to view strictly positive types simply as memory locations in which data can be stored. The container functor associated to $\mathbb{N} \lhd \mathsf{Fin}$ allows us to represent concrete lists. The list of Chars [‘r’, ‘e’, ‘d’] is represented as $(3, (0 \mapsto \text{‘}r\text{’}; 1 \mapsto \text{‘}e\text{’}; 2 \mapsto \text{‘}d\text{’})) : \sum_{n:\mathbb{N}} (\mathsf{Fin}\,n \to \mathsf{Char})$.

The two main results formalised in this paper 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 9.** *For a (unary) container functor* $[\![F]\!] : \mathsf{Type} \to \mathsf{Type}$*, the* <u>*wild category of*</u> <u>$[\![F]\!]$*-algebras*</u>*, denoted* $\mathsf{Alg}_{[\![F]\!]}$*, is defined as follows.*

- *Objects are* <u>*algebras*</u>*: pairs* $(X : \mathsf{Type}, \alpha : [\![F]\!]\,X \to X)$.[5]
- *A morphism of algebras* $(X, \alpha) \to (Y, \beta)$ *is a function* $f : X \to Y$ *such that the following square commutes.*

$$\begin{tikzcd} {[\![F]\!]\,X} \arrow[r, "{[\![F]\!]\,f}"] \arrow[d, "\alpha"'] & {[\![F]\!]\,Y} \arrow[d, "\beta"] \\ X \arrow[r, "f"'] & Y \end{tikzcd}$$

[5] X is sometimes called the carrier.

▶ **Definition 10.** *The initial algebra of a (unary) container functor* $[\![F]\!]\colon \mathsf{Type} \to \mathsf{Type}$ *is an algebra* (I, ι) *such that for every other 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 9, where the objects, coalgebras, are defined as pairs $(X\colon \mathsf{Type}, \alpha\colon X \to [\![F]\!]\,X)$. The terminal coalgebra is then an object (T, τ) such that for every other coalgebra (X, α), $\mathsf{Coalg}_{[\![F]\!]}((X, \alpha), (T, \tau))$ is contractible.

3 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.

3.1 Calculation of the initial algebra and terminal coalgebra

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

$$[\![A_\mu \lhd \mathbf{B}_\mu]\!]\colon (I \to \mathsf{Type}) \to \mathsf{Type}$$
$$[\![A_\nu \lhd \mathbf{B}_\nu]\!]\colon (I \to \mathsf{Type}) \to \mathsf{Type}$$

such that

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

Above, and for the remainder of the paper, $\cong$ is used to denote an equivalence of types.[6] The symbols μ and ν denote partial operators taking a wild functor to the carrier of its initial algebra or terminal coalgebra respectively, if they exist. The notation $\mu Y.[\![F]\!](\mathbf{X}, Y)$ is shorthand for (the carrier of) the initial algebra of the wild functor G defined by $G\,Y :\equiv [\![F]\!](\mathbf{X}, Y)$, and similarly for ν.

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

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

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

[6] In HoTT, there are several different notions of type equivalence. In our formalisation, we primarily use a definition in terms of *quasi-inverses* [26, 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.

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 $[\![S \lhd Q]\!]$ and construct $\mathbf{B} : I \to \phi \to \mathsf{Type}$. Being a fixed point of $[\![S \lhd Q]\!]$ means that ϕ consists of a carrier $\mathsf{C} : \mathsf{Type}$ together with an isomorphism, $\chi : [\![S \lhd Q]\!]\, \mathsf{C} \cong \mathsf{C}$.

```
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 $[\![\mathsf{C} \lhd \mathbf{B}]\!]\mathbf{X}$ is to be a fixed point of $[\![F]\!](\mathbf{X}, -)$, by Lambek's theorem [19], the following isomorphism is induced.

$$[\![F]\!](\mathbf{X}, [\![\mathsf{C} \lhd \mathbf{B}]\!]\, \mathbf{X}) \cong [\![\mathsf{C} \lhd \mathbf{B}]\!]\, \mathbf{X} \tag{2}$$

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

$$\begin{aligned}
&\sum_{s:S}\Big(\Big(\prod_i (\mathbf{P}\ i\ s \to \mathbf{X}\ i)\Big) \times (Q\ s \to [\![\mathsf{C} \lhd \mathbf{B}]\!]\, \mathbf{X})\Big) && \text{using Equation (1)}\\
=&\sum_{s:S}\Big(\Big(\prod_i (\mathbf{P}\ i\ s \to \mathbf{X}\ i)\Big) \times \Big(Q\ s \to \sum_{c:\mathsf{C}}\Big(\prod_i (\mathbf{B}\ i\ c \to \mathbf{X}\ i)\Big)\Big)\Big) && \text{definition of } [\![\_]\!]\\
\cong&\sum_{s:S}\Big(\Big(\prod_i (\mathbf{P}\ i\ s \to \mathbf{X}\ i)\Big) \times \sum_{f\,:\,Q\ s \to \mathsf{C}}\Big(\prod_{q:Q\ s}\prod_i (\mathbf{B}\ i\ (f\ q) \to \mathbf{X}\ i)\Big)\Big) && \text{distributivity of } \Pi \text{ over } \Sigma\\
\cong&\sum_{(s,f)\,:\,\sum_{s:S}(Q\ s \to \mathsf{C})}\Big(\prod_i \Big(\mathbf{P}\ i\ s + \sum_{q:Q\ s}(\mathbf{B}\ i\ (f\ q))\Big) \to \mathbf{X}\ i\Big) && \text{commutativity of } \times \text{ and } (A \uplus B) \to C \cong (A \to C) \times (B \to C)\\
=&\Big[\!\!\Big[\sum_{s:S}(f : Q\ s \to \mathsf{C}) \lhd \Big(\lambda\, i.\, \mathbf{P}\ i\ s + \sum_{q:Q\ s}(\mathbf{B}\ i\ (f\ q))\Big)\Big]\!\!\Big]\, \mathbf{X} && \text{definition of } [\![\_]\!]
\end{aligned}$$

The induced isomorphism (2) can then be written as

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

We already have the isomorphism $\chi : \sum_{s:S}(f : Q\ s \to \mathsf{C}) \cong \mathsf{C}$ on shapes. We will also need the below isomorphism on positions for $i : I$ and $c : \mathsf{C}$. 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 $(\phi\ \xi_1)\ c : Q\ ((\phi\ \xi_0)\ c) \to \mathsf{C}$.

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

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) (i : I) : φ .C → Type where
  here : {c : φ .C} → P i ((φ ξ₀) c) → Pos φ i c
  below : {c : φ .C} (q : Q ((φ ξ₀) c)) → Pos φ i ((φ ξ₁) c q) → Pos φ i c
```

It turns out that Pos works for both cases: we set $\mathbf{B}_\mu = \mathsf{Pos}\ \mathsf{WAlg}$ and $\mathbf{B}_\nu = \mathsf{Pos}\ \mathsf{MAlg}$. It is not immediately clear that choosing $\mathbf{B}_\nu$ to be an inductive (and not coinductive) family over $\mathsf{M}\, S\, Q$ 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 $\phi = \mathsf{MAlg}, I = \top$, and $\mathbf{P}\ \mathsf{tt}\ s := \top$ (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 Section 2.2 the M trees encoding 0, 1, and ∞ respectively of type $\mathbb{N}\infty$.

$$\begin{tikzcd} \mathsf{inl\ tt} \end{tikzcd} \qquad \begin{tikzcd} \mathsf{inr\ tt} \arrow[dr] & \\ & \mathsf{inl\ tt} \end{tikzcd} \qquad \begin{tikzcd} \mathsf{inr\ tt} \arrow[loop right] \end{tikzcd}$$

Now we look at the elements of PosM (M S P), where recall M S P $\cong \mathbb{N}\infty$, given the different elements $0, 1$, and ∞ of $\mathbb{N}\infty$. 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 here, since P (inr tt) is now $\top$. For the third tree (encoding ∞), PosM consists of here, below here, below (below 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 4.

3.2 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 $i : I$, and a family of types $A : (c : \phi\ .\mathsf{C}) \to \mathsf{Pos}\ \phi\ i\ c \to \mathsf{Type}$ equipped with

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

induces, in the obvious way, a dependent function $(c : \phi\ .\mathsf{C})\ (p : \mathsf{Pos}\ \phi\ i\ c) \to A\ c\ p$. 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 $\mathsf{Pos}\ \phi\ i\ (f\ d)$ 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 entirely analogous to how path induction does not apply to paths with fixed endpoints). There are, of course, 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 11.** *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 to the right exists.*

$$\begin{tikzcd} D \arrow[rrd, "f"] & & \\ Q\ ((\phi\ \xi_0)\ (f\ d)) \arrow[u, dashed, "\widehat{f_d}"] \arrow[rr, "(\phi\ \xi_1)\ (f\ d)"'] & & \mathsf{C} \end{tikzcd}$$

▶ **Lemma 12** (Generalised Pos induction). *Let ϕ be a fixed point, $i : I$ an index, and $f : D \to \phi\ .\mathsf{C}$ a ϕ-retraction. Let $A : (d : D) \to \mathsf{Pos}\ \phi\ i\ (f\ d) \to \mathsf{Type}$ be a dependent type equipped with*

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

where $\widehat{p}$ is p transported along the witness of the fact the diagram in Definition 11 commutes. This data induces a dependent function $(d : D)\ (p : \mathsf{Pos}\ \phi\ i\ (f\ d)) \to A\ d\ p$.

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

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

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

4 Fixed points

Let us now show that the constructions from Section 3 are correct: $[\![\mathsf{W}\ S\ Q \lhd \mathsf{Pos\ WAlg}]\!]\ \mathbf{X}$ is the initial $[\![F]\!](\mathbf{X}, -)$-algebra carrier, and $[\![\mathsf{M}\ S\ Q \lhd \mathsf{Pos\ MAlg}]\!]\ \mathbf{X}$ is the terminal $[\![F]\!](\mathbf{X}, -)$-coalgebra carrier. The proofs in this section mostly follow those given in [3], but in the more general (UIP-free) setting of Type instead of Set.

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

▶ **Theorem 13.** *Let $F = (S \lhd \mathbf{P}, Q)$ 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 $[\![\mathsf{W}\ S\ Q \lhd \mathsf{Pos\ WAlg}]\!]\ \mathbf{X}$ is the carrier of the initial algebra of $[\![F]\!](\mathbf{X}, -) : \mathsf{Type} \to \mathsf{Type}$, i.e.*

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

Proof of Theorem 13. We write W for W S Q and $\mathsf{Pos}\mu$ for Pos WAlg. We construct an $[\![F]\!](\mathbf{X}, -)$-algebra with carrier $[\![\mathsf{W} \lhd \mathsf{Pos}\mu]\!]\ \mathbf{X}$ by defining a morphism

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

by induction on $\mathsf{Pos}\mu$ as follows.[7]

$$\begin{aligned}
&\mathsf{fst}\ (\mathsf{into}\ ((s\ ,\ f)\ ,\ g\ ,\ h)) = \mathsf{sup\text{-}W}\ s\ f\\
&\mathsf{snd}\ (\mathsf{into}\ ((s\ ,\ f)\ ,\ g\ ,\ h))\ ind\ (\mathsf{here}\ p) = g\ ind\ p\\
&\mathsf{snd}\ (\mathsf{into}\ ((s\ ,\ f)\ ,\ g\ ,\ h))\ ind\ (\mathsf{below}\ q\ b) = h\ ind\ q\ b
\end{aligned}$$

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

$$\begin{tikzcd}
{[\![F]\!](\mathbf{X}, [\![\mathsf{W} \lhd \mathsf{Pos}\mu]\!]\ \mathbf{X})} \arrow[r, "\mathsf{into}"] \arrow[d, "{[\![F]\!](\mathbf{X},\overline{\alpha})}"'] & {[\![\mathsf{W} \lhd \mathsf{Pos}\mu]\!]\ \mathbf{X}} \arrow[d, "\overline{\alpha}"] \\
{[\![F]\!](\mathbf{X}, Y)} \arrow[r, "\alpha"'] & Y
\end{tikzcd} \tag{3}$$

[7] We repackage the type of the input of into via Equation (1) and distributivity of functions over Σ. This also applies for the definition of out in Theorem 14.

We define $\overline{\alpha}\colon \sum_{w:\mathsf{W}}((ind : Ind) \to \mathsf{Pos}\mu\ ind\ w \to \mathbf{X}\ ind) \to Y$ by induction on W.[8]

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

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

That (3) commutes then follows definitionally.

The only thing left to show is that $\overline{\alpha}$ is unique. We assume there is another arrow $\tilde{\alpha}\colon [\![\mathsf{W} \triangleleft \mathsf{Pos}\mu]\!]\,\mathbf{X} \to Y$ making (3) commute, i.e.

$$\tilde{\alpha} \circ \mathsf{into} \equiv \alpha \circ [\![F]\!](\mathbf{X}, \tilde{\alpha}), \tag{4}$$

and prove that for $w : \mathsf{W}, k\colon \mathsf{Pos}\mu\ ind\ w \to \mathbf{X}\ ind$, we have $\tilde{\alpha}(w,k) \equiv \overline{\alpha}(w,k)$. By induction on W, we have to show that for $s : S, f\colon Q\ s \to \mathsf{W}$, 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 (4), and our inductive hypothesis. ◀

Next, we show that $[\![\mathsf{M}\ S\ Q \triangleleft \mathsf{Pos\ MAlg}]\!]\,\mathbf{X}$ is the terminal $[\![S \triangleleft \mathbf{P}, Q]\!](\mathbf{X}, -)$-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 14.** *Let $F = (S \triangleleft \mathbf{P}, Q)$ be a container in $Ind + 1$ parameters with $S : \mathsf{Type}, \mathbf{P}\colon Ind \to S \to \mathsf{Type}$, and $Q\colon S \to \mathsf{Type}$. For any fixed $\mathbf{X}\colon Ind \to \mathsf{Type}$, the type $[\![\mathsf{M}\ S\ Q \triangleleft \mathsf{Pos\ MAlg}]\!]\,\mathbf{X}$ is the carrier of the terminal coalgebra of $[\![F]\!](\mathbf{X}, -)\colon \mathsf{Type} \to \mathsf{Type}$, i.e.*

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

Before we prove Theorem 14, we spell out what it is we need to show. We write M for $\mathsf{M}\ S\ Q$ and $\mathsf{Pos}\nu$ for $\mathsf{Pos\ MAlg}$. First, we construct an $[\![F]\!](\mathbf{X}, -)$-coalgebra with carrier $[\![\mathsf{M} \triangleleft \mathsf{Pos}\nu]\!]\,\mathbf{X}$ by defining

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

roughly as into^{-1}, where into is the function from Theorem 13.

```
out (m , k) = (shape m , pos m) , ((λ ind p → k ind (here p)) , (λ ind q b → k ind (below q b)))
```

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

$$\begin{tikzcd}
Y \arrow[r, "\beta"] \arrow[d, "\overline{\beta}"'] & {[\![F]\!](\mathbf{X}, Y)} \arrow[d, "{[\![F]\!](\mathbf{X}, \overline{\beta})}"] \\
{[\![\mathsf{M} \triangleleft \mathsf{Pos}\nu]\!]\,\mathbf{X}} \arrow[r, "\mathsf{out}"'] & {[\![F]\!](\mathbf{X}, [\![\mathsf{M} \triangleleft \mathsf{Pos}\nu]\!]\,\mathbf{X})}
\end{tikzcd} \tag{5}$$

[8] 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.

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

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

To prove Theorem 14 we (i) construct $\overline{\beta} \colon Y \to \sum_{m:\mathsf{M}} ((ind : Ind) \to \mathsf{Pos}\nu\ ind\ m \to \mathbf{X}\ ind)$ such that (5) commutes and (ii) show that this $\overline{\beta}$ is unique. This will be the content of Lemmas 15 and 16.

▶ **Lemma 15.** *There is a function* $\overline{\beta} \colon Y \to \sum_{m:\mathsf{M}} ((ind : Ind) \to \mathsf{Pos}\nu\ ind\ m \to \mathbf{X}\ ind)$ *making* (5) *commute.*

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

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

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

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

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

This construction makes (5) commute by definition. ◀

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

$$\mathsf{out} \circ \tilde{\beta} \equiv [\![F]\!](\mathbf{X}, \tilde{\beta}) \circ \beta, \tag{6}$$

then show that $\tilde{\beta} \equiv \overline{\beta}$. Naming $\tilde{\beta}$'s first and second projections $\tilde{\beta}_1$ and $\tilde{\beta}_2$, assumption (6) 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 $y : Y$.

$$comm_1\ y\ :\ \mathsf{shape}\ (\tilde{\beta}_1\ y) \equiv \beta s\ y$$

$$comm_2\ y\ :\ \mathsf{pos}\ (\tilde{\beta}_1\ y) \approxeq (\lambda\ q \to \tilde{\beta}_1\ (\beta h\ y\ q))$$

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

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

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

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

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

These equations simply express the fact that for $\tilde{\beta}$ to make diagram (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 16.** *The function* $\overline{\beta} : Y \to \sum_{m:\mathsf{M}} ((ind : Ind) \to \mathsf{Pos}\nu\ ind\ m \to \mathbf{X}\ ind)$ *from Lemma 15 is unique. In other words, under the assumption of the existence of* $comm_1$–$comm_4$ *above, we can construct the following paths.*

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

↥ dependent path over $(\lambda i \to (ind : Ind) \to \mathsf{Posv}\ ind\ (\mathsf{fstEq}\ y\ i)\ X\ ind)$

Proof of Lemma 16, part 1: construction of fstEq. Recall the coinduction principle MCoind from Section 2. Using this, we can prove fstEq in a rather straightforward manner. To apply it, 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 $\mathsf{R}\ (\mathsf{pos}\ (\tilde{\beta}_1\ y\ q_0))\ (\overline{\beta}_1\ (\beta h\ y\ q_1))$ while $\mathsf{R\text{-}intro}\ (\beta h\ y\ q_1)$ is of type $\mathsf{R}\ (\tilde{\beta}_1\ (\beta h\ y\ q_1))\ (\overline{\beta}_1\ (\beta h\ y\ q_1))$. This mismatch is adjusted using $comm_2$. Explicitly, we transport over the path of types $(\lambda\, i \to \mathsf{R}\ (comm_2\ y\ ({\sim}\ i)\ (q\text{-}eq\ ({\sim}\ i))))$.

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, let us 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 $\mathsf{pos}\ (\tilde{\beta}_1\ y) \approx \overline{\beta}_1 \circ (\beta h\ y)$ over the path of types $(\lambda\ i\ \to\ Q\ (comm_1\ y\ i) \to \mathsf{M})$. 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

$$\mathsf{pos}\ (\tilde{\beta}_1\ y) \xrightarrow{comm_2\ y} (\lambda\ q \to \tilde{\beta}_1\ (\beta h\ y\ q)) \xrightarrow{\mathsf{fstEq} \circ (\beta h\ y)} (\lambda\ q \to \overline{\beta}_1\ (\beta h\ y\ q)) \qquad (7)$$

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

▶ **Lemma 17.** *For all* $y : Y$*, we have* $\mathsf{fstEqPos}\ y \equiv (\lambda\ i \to \mathsf{pos}\ (\mathsf{fstEq}\ y\ i))$.

Proof sketch of Lemma 17. The lemma is proved by abstracting and applying function extensionality and path induction on $comm_1$. In this special case, i.e. when $comm_1\ y = \mathsf{refl}$, 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 4.1.

Let us now move on to the construction of sndEq. We construct sndEq using Lemma 12 which requires the following fact.

▶ **Lemma 18.** *$\overline{\beta}_1$ is an* MAlg-retraction.

Proof. By unfolding definitions, we see that the statement boils down to constructing, for each $y : Y$, a function $\widehat{f_y} : Q\ (\beta s\ y) \to Y$ such that the following identity holds for each $q : Q\ (\beta s\ y)$.

$$\overline{\beta}_1 y\ (\widehat{f_y}\ q) \equiv_{\mathsf{M}} \overline{\beta}_1 y\ (\beta h\ y\ q) \tag{8}$$

Defining $\widehat{f_y} = \beta h\ y$ makes (8) hold definitionally. ◀

Finally, we are ready to construct sndEq and thereby finish the proof of Lemma 16.

Proof of Lemma 16, part 2: construction of sndEq. For ease of notation, we define, for each $ind : Ind$ and $y : Y$, the function $\mathsf{tr}\ y : \mathsf{Pos}\nu\ ind\ (\overline{\beta}_1\ y) \to \mathsf{Pos}\nu\ ind\ (\tilde{\beta}_1\ y)$ to be the function obtained by transporting via $(\mathsf{fstEq}\ y)^{-1}$.[9] 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

$$\tilde{\beta}_2\ y\ ind\ (\mathsf{tr}\ y\ t) \equiv \overline{\beta}_2\ y\ ind\ t \tag{9}$$

for $ind : Ind$ and $t : \mathsf{Pos}\nu\ ind\ (\overline{\beta}_1\ y)$. In light of Lemma 18, we may apply Lemma 12 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 (9) 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

$$\tilde{\beta}_2\ (\beta h\ y\ q)\ ind\ (\mathsf{tr}\ (\beta h\ y\ q)\ p) \equiv \overline{\beta}_2\ (\beta h\ y\ q)\ ind\ p \tag{10}$$

and the goal is to show that

$$\tilde{\beta}_2\ y\ ind\ (\mathsf{tr}\ y\ (\mathsf{below}\ q\ p)) \equiv \overline{\beta}_2\ y\ ind\ (\mathsf{below}\ q\ p). \tag{11}$$

The RHS of (11) is, by definition, equal to the RHS of (10). By commuting transports with below and using $comm_4$, we can rewrite the LHS of (10) to a term of the form $\tilde{\beta}_2\ y\ ind\ (\mathsf{below}\ (\mathsf{transport}\ \ldots\ q)\ (\mathsf{transport}\ \ldots\ p))$. Commuting transports with below in the LHS of (11), 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 17. This concludes the proof of Lemma 16 and thus also of Theorem 14. ◀

▶ **Example 19.** For a concrete example of Theorems 13 and 14, consider S, P_0, and P_1 as defined in Example 7. Then for a fixed X : Type, $[\![\mathsf{S} \triangleleft \mathsf{P}_0, \mathsf{P}_1]\!](X, -) \colon \mathsf{Type} \to \mathsf{Type}$ 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 and Cofin is the inductive type family over $\mathbb{N}\infty$ of finite and (countably) infinite sets. We note that $[\![\mathsf{S} \triangleleft \mathsf{P}_0, \mathsf{P}_1]\!](X, -) \cong \top \uplus (- \times X)$, the signature functor for List. $\mathbb{N} \triangleleft \mathsf{Fin}$ 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).

[9] Formally, we define $\mathsf{tr}\ y = \mathsf{transport}\ (\lambda\ i \to \mathsf{Pos}\nu\ ind\ ((\mathsf{fstEq}\ y)^{-1}\ i))$.

4.1 The absence of UIP and Agda's termination checker

One of the key contributions of this paper is the fact we were able to formalise Lemma 16 without using UIP. Our main difficulty was proving the technical Lemma 17 which essentially says that fstEq is coinductively defined in the obvious manner. In theory, `Cubical Agda` should allow us to define fstEq as

$$\begin{aligned}&\mathsf{shape}\ (\mathsf{fstEq}\ y\ i) = comm_1\ y\ i\\&\mathsf{pos}\ (\mathsf{fstEq}\ y\ i) = \mathsf{fstEqPos}\ i\end{aligned}$$

where we recall that fstEqPos is defined by coinductively calling fstEq as in (7). This construction would make Lemma 17 hold definitionally, without requiring any form of UIP. There are, however, two issues with it. 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 [6].

If the first issue were to be resolved, our proof of Theorem 14 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 issue 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 16 was to support the claim by Abbott et al. [3] 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 [8]) and coinduction for M-types. The fact that the authors of [3] never mention any results akin to Lemma 17 suggests that they worked under a tacit assumption of UIP, which is further evidence that our formalisation indeed is a generalisation of previous results.

5 Conclusion, Related Work, and Future Work

In this paper, we presented a formalisation of the following results on containers, doing so without making any h-set assumptions, thereby lifting 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. These workarounds, however, suggested that our construction holds not only in *cubical* type theory, but in any type theory with function extensionality and W- and M- types.

Similar results have been formalised in `Lean` by Avigad et al. [9], 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 [28] wrote about the semantics of allowing path abstraction in copattern matching definitions in `Cubical Agda`. Ahrens et al. [5] 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 2.2. It is, however, 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.

The formalisation of Theorems 13 and 14 is part of ongoing work on giving a comprehensive rendition and formalisation of containers in type theory. In terms of the main result of [3] stating that 'each strictly positive type in n variables can be interpreted as an n-ary container', we have proved that 'container functors, without any h-level restriction, are closed under μ and ν'. In our experience, compared to strictly positive types formed using $0, 1, +, \times$, and $\rightarrow$, μ and ν are the most challenging cases. One aspect of the original result in [3] that we are still missing is that they show that '*containers* are closed under μ and ν', i.e. containers, and not their functor interpretations, model strictly positive types. In the h-set level setting of [3], one can easily move closure properties from container functors to containers via the fully faithful functor $[\![\_]\!] : \mathsf{Cont} \rightarrow [\mathsf{I} \rightarrow \mathsf{Set}, \mathsf{Set}]$. In our more general setting of using Type instead of Set, the wild functor on wild categories $[\![\_]\!] : \mathsf{Cont} \rightarrow [\mathsf{I} \rightarrow \mathsf{Type}, \mathsf{Type}]$ being in some sense fully faithful does not immediately follow. To show this, we might need to somehow 'tame' the wild category $[\mathsf{I} \rightarrow \mathsf{Type}, \mathsf{Type}]$ by adding coherences, but stating Type as a higher category in HoTT is still an open problem (see e.g. [10]).

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 [7, 16]. 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.

References

1 Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Categories of containers. In Andrew D. Gordon, editor, *Foundations of Software Science and Computation Structures*, pages 23–38, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. URL: `https://dl.acm.org/doi/10.5555/1754809.1754813`.

2 Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Representing nested inductive types using W-types. In *Automata, Languages and Programming, 31st International Colloqium (ICALP)*, pages 59 – 71, 2004. `doi:10.1007/978-3-540-27836-8_8`.

3 Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Containers: Constructing strictly positive types. *Theoretical Computer Science*, 342(1):3–27, 2005. Applied Semantics: Selected Topics. `doi:10.1016/j.tcs.2005.06.002`.

4 Michael Gordon Abbott. *Categories of Containers.* PhD thesis, University of Leicester, 2003. URL: `https://hdl.handle.net/2381/30102`.

5 Benedikt Ahrens, Paolo Capriotti, and Régis Spadotti. Non-wellfounded trees in homotopy type theory. In Thorsten Altenkirch, editor, *13th International Conference on Typed Lambda Calculi*

and Applications (TLCA 2015), volume 38 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 17–30, Dagstuhl, Germany, 2015. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.TLCA.2015.17.

6 Thorsten Altenkirch. Cubical primitives should preserve guardedness. https://github.com/agda/agda/issues/4740, 2020. Online.

7 Thorsten Altenkirch. Quotient inductive types as categorified containers. https://hott-uf.github.io/2024/abstracts/HoTTUF_2024_paper_10.pdf, 2024. HoTT/UF 2024 abstract.

8 Thorsten Altenkirch, Simon Boulier, Ambrus Kaposi, and Nicolas Tabereau. Setoid type theory—a syntactic translation, 2019. doi:10.1007/978-3-030-33636-3_7.

9 Jeremy Avigad, Mario Carneiro, and Simon Hudon. Data Types as Quotients of Polynomial Functors. In John Harrison, John O'Leary, and Andrew Tolmach, editors, *10th International Conference on Interactive Theorem Proving (ITP 2019)*, volume 141 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 6:1–6:19, Dagstuhl, Germany, 2019. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ITP.2019.6.

10 Ulrik Buchholtz. *Higher Structures in Homotopy Type Theory*, pages 151–172. Springer International Publishing, Cham, 2019. doi:10.1007/978-3-030-15655-8_7.

11 Joshua Chen. 2-coherent internal models of homotopical type theory, 2025. arXiv:2503.05790.

12 Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom. In Tarmo Uustalu, editor, *21st International Conference on Types for Proofs and Programs (TYPES 2015)*, volume 69 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 5:1–5:34, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.TYPES.2015.5.

13 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, page 255–264, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3209108.3209197.

14 Damato, Stefania and Altenkirch, Thorsten and Ljunström, Axel. Formalising inductive and coinductive containers, June 2025. Accompanying formalisation. URL: https://github.com/stefaniatadama/formalising-inductive-coinductive-containers/blob/main/cubical/Cubical/Papers/Containers.agda.

15 Nicola Gambino and Martin Hyland. Wellfounded trees and dependent polynomial functors. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *Types for Proofs and Programs*, pages 210–225, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. doi:10.1007/978-3-540-24849-1_14.

16 Håkon Robbestad Gylterud. Symmetric containers, 2011. Master's thesis. URL: https://staff.math.su.se/gylterud/thesis_gylterud.pdf.

17 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, page 427–441, Berlin, Heidelberg, 1994. Springer-Verlag. doi:10.1007/BFb0022273.

18 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. IEEE Press, 2021. doi:10.5555/3470152.3470159.

19 Joachim Lambek. A fixpoint theorem for complete categories. *Mathematische Zeitschrift*, 103:151–161, 1968. URL: http://eudml.org/doc/170906.

20 Peter LeFanu Lumsdaine and Michael Shulman. Semantics of higher inductive types. *Mathematical Proceedings of the Cambridge Philosophical Society*, 169(1):159–208, 2020. doi:10.1017/S030500411900015X.

21 Per Martin-Löf. *An Intuitionistic Theory of Types.* Clarendon Press, 1972. Appears in 'Twenty-Five Years of Constructive Type Theory'.

22 Per Martin-Löf. Intuitionistic type theory, 1984. Notes by Giovanni Sambin, available at https://archive-pml.github.io/martin-lof/pdfs/Bibliopolis-Book-1984.pdf.

23 Per Martin-Löf. Constructive mathematics and computer programming. In L. Jonathan Cohen, Jerzy Łoś, Helmut Pfeiffer, and Klaus-Peter Podewski, editors, *Logic, Methodology and Philosophy of Science VI*, volume 104 of *Studies in Logic and the Foundations of Mathematics*, pages 153–175. Elsevier, 1982. `doi:10.1016/S0049-237X(09)70189-2`.

24 R.A.G. Seely. Locally cartesian closed categories and type theory. *Mathematical Proceedings of the Cambridge Philosophical Society*, 95:33–48, 1984. `doi:10.1017/S0305004100061284`.

25 The Agda Community. Cubical Agda Library, March 2025. URL: `https://github.com/agda/cubical`.

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

27 Benno van den Berg and Federico De Marchi. Non-well-founded trees in categories. *Annals of Pure and Applied Logic*, 146(1):40–59, 2007. `doi:10.1016/j.apal.2006.12.001`.

28 Andrea Vezzosi. Streams for cubical type theory, 2017. Unpublished note, available at `https://saizan.github.io/streams-ctt.pdf`.

29 Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. Cubical Agda: A dependently typed programming language with univalence and higher inductive types. *Journal of Functional Programming*, 31:e8, 2021. `doi:10.1017/S0956796821000034`.

30 Vladimir Voevodsky. The equivalence axiom and univalent models of type theory. (Talk at CMU on February 4, 2010), 2014. `arXiv:1402.5556`.