JFP **25**, e5, 41 pages, 2015.
doi:10.1017/S095679681500009X

Indexed containers

THORSTEN ALTENKIRCH
School of Computer Science, University of Nottingham, Nottingham, UK

NEIL GHANI, PETER HANCOCK, CONOR MCBRIDE
Department of Computer and Information Sciences, University of Strathclyde, Glasgow, UK

PETER MORRIS
School of Computer Science, University of Nottingham, Nottingham, UK

Abstract

We show that the syntactically rich notion of strictly positive families can be reduced to a core type theory with a fixed number of type constructors exploiting the novel notion of indexed containers. As a result, we show indexed containers provide normal forms for strictly positive families in much the same way that containers provide normal forms for strictly positive types. Interestingly, this step from containers to indexed containers is achieved without having to extend the core type theory. Most of the construction presented here has been formalized using the Agda system.

1 Introduction

Inductive datatypes are a central feature of modern type theory (e.g. Coq The Coq Development Team (2008)) or functional programming (e.g. Haskell[1]). Examples include the natural numbers a la Peano:[2]

```
data ℕ : Set where
  zero : ℕ
  suc  : (n : ℕ) → ℕ
```

the set of lists indexed by a given set:

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

and the set of de Bruijn λ-terms:

[1] Here, we shall view Haskell as an approximation of strong functional programming as proposed by Turner (1985) and ignore non-termination.

[2] We are using Agda to represent constructions in type theory. Indeed, the source of this document is a literate Agda file which is available online, Altenkirch *et al.* (2015). For an overview over Agda see The Agda developers (2015), in particular the tutorials and the reference manual which explain how to read the code included in this paper.

```
data Lam : Set where
  var : (n : ℕ)      → Lam
  app : (f a : Lam)  → Lam
  lam : (t : Lam)    → Lam
```

An elegant way to formalize and reason about inductive types is to model them as the initial algebra of an endofunctor.[3] We can define the signature functors corresponding to each of the above examples as follows:

```
F_ℕ : Set → Set
F_ℕ X = ⊤ ⊎ X

F_List : (A : Set) → Set → Set
F_List A X = ⊤ ⊎ (A × X)

F_Lam : Set → Set
F_Lam X = ℕ ⊎ (X × X) ⊎ X
```

This perspective has been very successful in providing a generic approach to programming with and reasoning about inductive types, e.g. see the *Algebra of Programming* (Bird & de Moor 1997).

While the theory of inductive types is well developed, we often want to have a finer, more expressive, notion of type. This allows us, for example, to ensure the absence of runtime errors such as access to arrays out of range or access to undefined variables in the previous example of λ-terms. To model such finer types, we move to the notion of an inductive family in type theory. A family is a type indexed by another, already given, type. Our first example of an inductive family is the family of finite sets Fin which assigns to any natural number n, a type Fin n which has exactly n elements. Fin can be used where, in conventional reasoning, we assume a finite set, e.g. when dealing with a finite address space or a finite set of variables. The inductive definition of Fin refines the type of natural numbers:

```
data Fin : ℕ → Set where
  zero : ∀ {n}          → Fin (suc n)
  suc  : ∀ {n} (i : Fin n) → Fin (suc n)
```

In the same fashion, we can refine the type of lists to the type of vectors which are indexed by a number indicating the length of the vector:

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

Notice how using the inductive family Vec instead of List enables us to write a total projection function projecting the *n*th element out of vector:

```
_!!_ : {A : Set} → {n : ℕ} → Vec A n → Fin n → A
[]     !! ()
```

[3] This requires a type theory with an extensional propositional equality.

```
(a :: as) !! zero  = a
(a :: as) !! suc n = as !! n
```

In contrast, the corresponding function $\_!!\_ : \{A : \mathsf{Set}\} \to \mathsf{List}\ A \to \mathbb{N} \to A$ is not definable in a total language like Agda.

Finally, we can define the family of a well-scoped lambda terms ScLam which assigns to a natural number n the set of λ-terms with at most n free variables ScLam n. DeBruijn variables are now modeled by elements of Fin n replacing Nat in the previous, unindexed definition of λ-terms Lam.

```
data ScLam (n : ℕ) : Set where
  var : (i : Fin n)           → ScLam n
  app : (f a : ScLam n)       → ScLam n
  lam : (t : ScLam (suc n))   → ScLam n
```

Importantly, the constructor lam reduces the number of *free* variables by one. Inductive families may be mutually defined, for example the scoped versions of β (NfLam) normal forms and neutral λ-terms (NeLam):

```
mutual
  data NeLam (n : ℕ) : Set where
    var : (i : Fin n)                  → NeLam n
    app : (f : NeLam n) (a : NfLam n)  → NeLam n
  data NfLam (n : ℕ) : Set where
    lam : (t : NfLam (suc n))          → NfLam n
    ne  : (t : NeLam n)                → NfLam n
```

The initial algebra semantics of inductive types can be extended to model inductive families by replacing functors on the category Set with functors on the category of families indexed by a given type – in the case of all our examples so far this indexing type was Nat. The objects of the category of families indexed over a type $I : \mathsf{Set}$ are I-indexed families of sets, i.e. functions of type $I \to \mathsf{Set}$, and a morphism between I-indexed families $A, B : I \to \mathsf{Set}$ is given by a family of maps $f : (i : I) \to A\ i \to B\ i$. Indeed, this category is easily seen to be isomorphic to the slice category Set/I but the chosen representation is more convenient type theoretically. Using Σ-types and equality types from type theory, we can define the following endofunctors $\mathsf{F}_{\mathsf{Fin}}$, $\mathsf{F}_{\mathsf{Vec}}$ and $\mathsf{F}_{\mathsf{Lam}}$ on the category of families over Nat whose initial algebras are Fin and Lam, respectively:

$$\mathsf{F}_{\mathsf{Fin}} : (\mathbb{N} \to \mathsf{Set}) \to \mathbb{N} \to \mathsf{Set}$$
$$\mathsf{F}_{\mathsf{Fin}}\ X\ n = (m : \mathbb{N}) \times (n \equiv \mathsf{suc}\ m) \times (\top \uplus X\ m)$$
$$\mathsf{F}_{\mathsf{Vec}} : (A : \mathsf{Set}) \to (\mathbb{N} \to \mathsf{Set}) \to \mathbb{N} \to \mathsf{Set}$$
$$\mathsf{F}_{\mathsf{Vec}}\ A\ X\ n = n \equiv \mathsf{zero} \uplus ((m : \mathbb{N}) \times (n \equiv \mathsf{suc}\ m) \times (A \times X\ m))$$
$$\mathsf{F}_{\mathsf{ScLam}} : (\mathbb{N} \to \mathsf{Set}) \to \mathbb{N} \to \mathsf{Set}$$
$$\mathsf{F}_{\mathsf{ScLam}}\ X\ n = \mathsf{Fin}\ n \uplus (X\ n \times X\ n) \uplus (X \circ \mathsf{suc})\ n$$

The equality type expresses the focussed character of the constructors for Fin. The mutual definition of NeLam and NfLam can be represented by two binary functors:

$$
\begin{aligned}
&\mathsf{F}_{\mathsf{NeLam}} : (\mathbb{N} \to \mathsf{Set}) \to (\mathbb{N} \to \mathsf{Set}) \to \mathbb{N} \to \mathsf{Set}\\
&\mathsf{F}_{\mathsf{NeLam}}\, \mathsf{X}\, \mathsf{Y}\, \mathsf{n} = \mathsf{Fin}\, \mathsf{n} \uplus (\mathsf{X}\, \mathsf{n} \times \mathsf{Y}\, \mathsf{n})\\
&\mathsf{F}_{\mathsf{NfLam}} : (\mathbb{N} \to \mathsf{Set}) \to (\mathbb{N} \to \mathsf{Set}) \to \mathbb{N} \to \mathsf{Set}\\
&\mathsf{F}_{\mathsf{NfLam}}\, \mathsf{X}\, \mathsf{Y}\, \mathsf{n} = (\mathsf{Y} \circ \mathsf{suc})\, \mathsf{n} \uplus \mathsf{X}\, \mathsf{n}
\end{aligned}
$$

We can construct NeLam and NfLam as follows: first, we define a parametrized initial algebra $\mathsf{NeLam'} : (\mathbb{N} \to \mathsf{Set}) \to \mathbb{N} \to \mathsf{Set}$ so that $\mathsf{NeLam'}\, \mathsf{Y}$ is the initial algebra of $\lambda\, \mathsf{X} \to \mathsf{F}_{\mathsf{NeLam}}\, \mathsf{X}\, \mathsf{Y}$ and then NfLam is the initial algebra of $\lambda\, \mathsf{Y} \to \mathsf{F}_{\mathsf{NfLam}}\, (\mathsf{NeLam'}\, \mathsf{Y})\, \mathsf{Y}$. Symmetrically we derive NeLam. Compare this with the encoding in Section 8.

This approach extends uniformly to more complicated examples such as the family of typed λ-terms, using lists of types to represent typing contexts:

$$
\begin{aligned}
&\textbf{data}\ \mathsf{Ty} : \mathsf{Set}\ \textbf{where}\\
&\quad \iota : \mathsf{Ty}\\
&\quad \_ \Rightarrow \_ : (\sigma\ \tau : \mathsf{Ty}) \to \mathsf{Ty}\\
&\textbf{data}\ \mathsf{Var}\, (\tau : \mathsf{Ty}) : \mathsf{List}\ \mathsf{Ty} \to \mathsf{Set}\ \textbf{where}\\
&\quad \mathsf{zero} : \forall\, \{\Gamma\} \to \mathsf{Var}\ \tau\ (\tau :: \Gamma)\\
&\quad \mathsf{suc}\ \ : \forall\, \{\sigma\ \Gamma\}\, (\mathsf{i} : \mathsf{Var}\ \tau\ \Gamma) \to \mathsf{Var}\ \tau\ (\sigma :: \Gamma)\\
&\textbf{data}\ \mathsf{STLam}\, (\Gamma : \mathsf{List}\ \mathsf{Ty}) : \mathsf{Ty} \to \mathsf{Set}\ \textbf{where}\\
&\quad \mathsf{var}\ \ : \forall\, \{\tau\}\ \ (\mathsf{i} : \mathsf{Var}\ \tau\ \Gamma) \to \mathsf{STLam}\ \Gamma\ \tau\\
&\quad \mathsf{app} : \forall\, \{\sigma\ \tau\}\, (\mathsf{f} : \mathsf{STLam}\ \Gamma\ (\sigma \Rightarrow \tau))\\
&\qquad\qquad\ \ (\mathsf{a} : \mathsf{STLam}\ \Gamma\ \sigma) \to \mathsf{STLam}\ \Gamma\ \tau\\
&\quad \mathsf{lam} : \forall\, \{\sigma\ \tau\}\, (\mathsf{t} : \mathsf{STLam}\ (\sigma :: \Gamma)\ \tau) \to \mathsf{STLam}\ \Gamma\ (\sigma \Rightarrow \tau)
\end{aligned}
$$

Types like this can be used to implement a tag-free, terminating evaluator (Altenkirch & Chapman 2009). Obtaining the corresponding functors is a laborious but straightforward exercise. As a result of examples such as the above, inductive families have become the backbone of dependently typed programming as present in Epigram or The Agda Team (2015). Coq also supports the definition of inductive families but programming with them is rather hard – a situation which has been improved by the `Program` tactic (Sozeau 2007).

Indexed containers are designed to provide the mathematical and computational infrastructure required to program with inductive families. The remarkable fact about indexed containers, and the fact which underpins their practical usefulness, is that they offer an exceedingly compact way to encapsulate all the information inherent within the definition of functors such as $\mathsf{F}_{\mathsf{Fin}}$, $\mathsf{F}_{\mathsf{Vec}}$ and $\mathsf{F}_{\mathsf{ScLam}}$, $\mathsf{F}_{\mathsf{NeLam}}$ and $\mathsf{F}_{\mathsf{NfLam}}$ and hence within the associated inductive families Fin, Vec, ScLam, NeLam and NfLam. The second important thing about indexed containers is that not only can they be used to represent functors, but the canonical constructions on functors can be internalized to become constructions on the indexed containers which represent those functors. As a result, we get a compositional

combinator language for inductive families as opposed to simply a syntactic definitional format for inductive families.

1.1 Related work

This paper is an expanded and revised version of the LICS paper by the first and 4th author (Morris & Altenkirch 2009). In the present paper, we have integrated the Agda formalization in the main development, which in many instances required extending it. We have made explicit the use of relative monads which was only hinted at in the conference version, based on recent work (Altenkirch *et al.* 2010). We have also dualized the development to terminal coalgebras which required the type of paths to be defined inductively instead of recursively as done in the conference paper (Section 6). We have also formalized the derivation of indexed W-types from ordinary W-types (Section 7.1). The derivation of M-types from W-types (Section 7.2) was already given in Abbott *et al.* (2005) and is revisited here exploiting the indexed W-type derived previously. Moreover, the development is fully formalized in Agda.

Indexed containers are intimately related to *dependent polynomial functors* (Gambino & Hyland 2004a), see also the comprehensive notes (Kock 2009). Indeed, at a very general level one could think of indexed containers as the type theoretic version of dependent polynomials and vice versa. However, the different needs of programmers from category theorists has taken our development of indexed containers in a different direction from that of dependent polynomials. In this vein, an important contribution is the Agda implementation of our ideas which makes our work more useful to programmers than the categorical work on dependent polynomials. We also focus on syntactic constructions such using indexed containers to model mutual and nested inductive definitions. As a consequence, we show that indexed containers are closed under parametrized initial algebras and coalgebras and reduce the construction of parameterized final coalgebras to that of initial algebras. Hence, we can apply both the initial algebra and final coalgebra construction several times. The flexibility of indexed containers allows us to also establish closure under the adjoints of reindexing. This leads directly to a grammar for strictly positive families, which itself is an instance of a strictly positive family (Section 8) – see also our previous work (Morris *et al.* 2007a,b). This generalizes previous results on strictly positive datatypes by Dybjer (1997) which have been further developed in Abbott *et al.* (2005).

Containers are related to Girard's normal functors (Girard 1988) which themselves are a special case of Joyal's analytic functors (Joyal 1987) – those that allow only finite sets of positions. Fiore, Gambino, Hyland and Winskel's work on generalized species (Fiore *et al.* 2008) considers those concepts in a more generic setting – the precise relation of this work to indexed containers remains to be explored but it appears that generalized species can be thought of as indexed containers closed under quotients.

Perhaps the earliest publication related to indexed containers occurs in Petersson and Synek's paper (Petersson & Synek 1989) from 1989. They present rules extending Martin-Löf's type theory with a set constructor for 'tree sets' : families of mutually defined inductive sets, over a fixed index set. Indeed, Petersson–Synek trees are semantically equivalent to the WI-type we define in Section 5 – the difference is that WI-types represent positions as a family indexed over the output positions while the tree type use a set of

positions together with a function which assigns the output position. This is an instance of Grothendieck's well-known inverse image construction. Inspired in part by Petersson and Synek's constructor, Hancock, Hyvernat and Setzer (Hancock & Hyvernat 2006) applied indexed (and unindexed) containers under the name 'interaction structures' to the task of modeling imperative interfaces such as command-response interfaces in a number of publications. The construction of WI-types from W-types in Section 7 is related to the reduction of indexed induction-recursion to induction-recursion in Dybjer & Setzer (2001) and the construction of initial algebras in Gambino & Hyland (2004b). The use of ω-limits to construct final coalgebras in the same section is folklore, e.g. see Adámek & Koubek (1995) and Lindström (1989).

The implementation of Generalized Algebraic Datatypes (GADTs) (Cheney & Hinze *et al.* 2003) allows Fin and Lam to be encoded in Haskell:

```
data Fin a where
  FZero :: Fin (Maybe a)
  FSucc :: Fin a -> Fin (Maybe a)

data Lam a where
  Var :: Fin a -> Lam a
  App :: Lam a -> Lam a -> Lam a
  Abs :: Lam (Maybe a) -> Lam a
```

Here, `Fin` and `Lam` are indexed by types instead of natural numbers; The type constructor `Maybe` serves as a type level copy of the succ constructor for natural numbers. Note that `Lam` is actually just a nested datatype (Altenkirch & Reus 1999) while `Fin` exploits the full power of GADTs because the range of the constructors is constrained. The problem with using GADTs to model inductive families is, however, that the use of type level proxies for say, natural numbers, means that computation must be imported to the type level. This is a difficult problem and probably limits the use of GADTs as a model of inductive families.

Since the publication of the LICS paper, indexed containers have been used as a base for the generic definition of datatypes for Epigram 2 (Chapman *et al.* 2010), and to develop the theory of ornaments (McBride 2010). In recent work, it has been shown that indexed containers are sufficient to express all *small* inductive-recursive definitions (Hancock *et al.* 2013)

1.2 Overview over the paper

We develop our type theoretic and categorical background in Section 2 and also summarize the basic definitions of non-indexed containers. In Section 3, we develop the concept of an indexed functor, showing that this is a relative monad and presenting basic constructions on indexed functors including the definition of a parametrized initial algebra. In Section 4, we develop the basic theory of indexed containers and relate them to indexed functors. Subsequently in Section 5, we construct parametrized initial algebras of indexed containers assuming the existence of indexed W-types, this can be dualized to show the existence of parametrized terminal coalgebras of indexed containers from indexed M-types in Section 6. Both requirements, indexed W-types and indexed M-types can be derived from ordinary

W-types, this is shown in Section 7. Finally, we define a syntax from strictly positive families and interpret this using indexed containers in Section 8.

The source of this paper is a literate Agda file, that is we have formally verified the constructions using Agda. There are some exceptions: Propositions 1–5 are only done on paper and MIext, that bisimilarity of MI trees implies extensional equality is postulated instead of proven. We also have omitted the functor laws and naturality laws from the formal development – we never rely on assuming that something is a functor or a natural transformation. These laws are of a particular simple form for indexed containers and hence we implicitly prove them when needed, e.g. in Proposition 10. The propositions which do rely on these assumptions, e.g. Proposition 3, are only done on paper. The reason for these omissions is that the purpose of the paper is to introduce indexed containers and a complete formalization of these more elementary results would have introduced significant technical complications distracting from our central purpose.

2 Background

2.1 Type theory

Our constructions are all developed in Agda, and so we adopt its syntax, but we will take certain liberties with its type theory. We have Π-types, denoted $(\mathsf{a} : \mathsf{A}) \to \mathsf{B}\,\mathsf{a}$ and Σ-types, which we write as: $(\mathsf{a} : \mathsf{A}) \times \mathsf{B}\,\mathsf{a}$. In fact, this is sugar for the record type:

```
record Σ (A : Set) (B : A → Set) : Set where
  constructor _,_
  field
    π₀ : A
    π₁ : B π₀
```

We will, however assume that the type theory we work in has Σ-types as primitive, and has no native support for datatypes. Instead, we only have W-types, the empty-type $\bot$, the unit type $\mathsf{tt} : \top$ and the booleans $\mathsf{true}, \mathsf{false} : \mathsf{Bool}$. A type theory has W-types if it has a type former $\mathsf{W} : (\mathsf{S} : \mathsf{Set})\,(\mathsf{P} : \mathsf{S} \to \mathsf{Set}) \to \mathsf{Set}$ with a constructor sup and an eliminator wrec:

```
data W (S : Set) (P : S → Set) : Set where
  sup : (s : S) × (P s → W S P) → W S P
wrec : {S : Set} {P : S → Set} (Q : W S P → Set)
       (x : W S P)
       (m : (s : S) (f : P s → W S P)
            (h : (p : P s) → Q (f p))
            → Q (sup (s , f)))
       → Q x
wrec Q (sup (s , f)) m = m s f (λ p → wrec Q (f p) m)
```

Agda comes with a predicative hierarchy of types where $\mathsf{Set} = \mathsf{Set}_0$ is the lowest universe. We sometimes define structures containing sets whose type is Set_1.

As a notational convenience, we will continue to define extra Agda datatypes in the rest of the paper, but in the end we will show how each of these can be reduced to a theory that contains only W. For compactness, and readability we will also define functions using Agda's pattern matching syntax, rather than encoding them using wrec. All of these definitions can be reduced to terms which only use wrec.

We'll also require a notion of propositional equality. To simplify the presentation of some definitions later on, we will employ a heterogeneous equality. This can be defined in Agda via a datatype:

```
data _≅_ {A : Set} (x : A) :
         {B : Set} → B → Set where
  refl : x ≅ x
subst : {A : Set} (P : A → Set) {x y : A} →
        x ≅ y → P x → P y
subst P refl p = p
```

Most of the time our equalities will be homogeneous, however, so we introduce a short hand for this:

```
_≡_ : {A : Set} → A → A → Set
a ≡ b = a ≅ b
```

Alternatively, we could have defined $\_\cong\_$ using Σ and homogenous equality. This is an intensional equality, but we want to work in a setting with extensional type theory, so we extend the propositional equality with this extensionality axiom:

```
postulate ext : {f g : (a : A) → B a} →
                ((a : A) → f a ≡ g a) → f ≡ g
ext⁻¹ : {f g : (a : A) → B a} →
        f ≡ g → ((a : A) → f a ≡ g a)
ext⁻¹ refl a = refl
syntax ext (λ a → b) = λ≡ a → b
```

We'll also need a heterogeneous version of the extensionality principle – this says that two functions of different types are equal iff, when applied to equal arguments they produce equal results. Note that to exploit a heterogeneous equality between functions we must provide a guarantee that the functions have equal domains, and co-domains:

```
postulate exteq : {f : (a : A) → B a}
                  {g : (a′ : A′) → B′ a′} →
                  ({a : A} {a′ : A′} →
                    a ≅ a′ → f a ≅ g a′) →
                  f ≅ g
syntax exteq (λ a → b) = λ≅ a → b
exteq⁻¹ : ∀ {l l′} {A A′ : Set l}
          {B : A → Set l′} {B′ : A′ → Set l′}
          {f : (a : A) → B a} {g : (a′ : A′) → B′ a′} →
```

```
                A ≡ A′ → B ≅ B′ → f ≅ g →
                {a : A} {a′ : A′} → a ≅ a′ → f a ≅ g a′
exteq⁻¹ refl refl refl {a} {.a} refl = refl
```

This creates non-canonical elements of $\_\cong\_$, *i.e.* closed terms in equality types which are not refl. In order to deal with these non-canonical elements, we also rely on axiom K, or the uniqueness of identity proofs:

```
UIP : {a b : A} {p : a ≅ b} {q : a ≅ b} → p ≅ q
UIP {p = refl} {q = refl} = refl
```

With these ingredients, we obtain a theory which captures extensional type theory in the sense that any intensional type which is inhabited in extensional type theory is also inhabited in intensional type theory with extensionality and K (Hofmann 1996).

We will also need to use a notion of Set isomorphism, which we denote $\_ \Longleftrightarrow \_$ and which exploits our extensional equality:

```
record _⟺_ (A B : Set) : Set where
field
  φ : A → B
  ψ : B → A
  φψ : φ ∘ ψ ≡ id
  ψφ : ψ ∘ φ ≡ id
```

We are going to use type theoretic versions of certain category theoretic concepts. For example, we represent functors by packing up their definition as an Agda record. An endofunctor on Set is given by:

```
record Func : Set₁ where
  field
    obj  : Set → Set
    mor : ∀ {A B} → (A → B) → obj A → obj B
```

It would also be possible to pack up the functor laws as extra fields in these records. We use *ends* (Mac Lane 1998) to capture natural transformations. Given a bifunctor $\mathsf{F} : \mathsf{Set}^{\mathsf{op}} \to \mathsf{Set} \to \mathsf{Set}$, an element of $\prod \mathsf{X} \,.\, \mathsf{F}\,\mathsf{X}\,\mathsf{X}$ is equivalent to an element of $\mathsf{f} : \{\mathsf{X} : \mathsf{Set}\} \to \mathsf{F}\,\mathsf{X}\,\mathsf{X}$, along with a proof: [4]

$$\{\mathsf{A}\,\mathsf{B} : \mathsf{Set}\}\,(\mathsf{g} : \mathsf{A} \to \mathsf{B}) \to \mathsf{F}\,\mathsf{g}\,\mathsf{B}\,(\mathsf{f}\,\{\mathsf{B}\}) \equiv \mathsf{F}\,\mathsf{A}\,\mathsf{g}\,(\mathsf{f}\,\{\mathsf{A}\})$$

The natural transformations between functors F and G are ends $\prod \mathsf{X} \,.\, \mathsf{F}\,\mathsf{X} \to \mathsf{G}\,\mathsf{X}$. We will often ignore the presence of the proofs, and use such ends directly as polymorphic functions. In this setting, the Yoneda lemma can be stated as follows, for any functor F:

$$\mathsf{F}\,\mathsf{X} \cong \prod \mathsf{Y} \,.\, (\mathsf{X} \to \mathsf{Y}) \to \mathsf{F}\,\mathsf{Y}$$

we will make use of this fact later on.

[4] For simplicity, we adopt here the standard convention to overload the object and morphism part of F.

Finally, for readability we will elide certain artifacts in Agda's syntax; for instance, the quantification of implicit arguments when their types can be inferred from the context. We will often leave out record projections from notions such as Func, allowing the functor to stand for both its action on object and morphism, just as would happen in the category theory literature.

2.2 Containers in a nutshell

Initial algebra semantics is useful for providing a generic analysis of inductive types based upon concepts such as constructors, functorial map and structured recursion operators. However, it does not cover the question which inductive types actually exist, and it falls short of providing a systematic characterization of generic operations such as equality or the zipper (Huet 1997; McBride 2001). To address this problem, we proposed in previous work to consider only a certain class of functors, namely those arising from containers (Abbott *et al.* 2003; Abbott *et al.* 2005). Since indexed containers build upon containers, we recall the salient information about containers. A (unary) container is given by a set of shapes S and a family of positions P assigning, to each shape, the set of positions where data can be stored in a data structure of that shape.

```
record Cont : Set₁ where
  constructor _◁_
  field
    S : Set
    P : S → Set
```

This shapes and positions metaphor is very useful in developing intuitions about containers. For example, every container $S \lhd P$ gives rise to a functor which maps a set A to the set of pairs consisting of a choice of a shape $s : S$ and a function assigning to every position $p : P\ s$ for that shape, an element of A to be stored at that position. This intuition is formalized by the following definition.

```
⟦_⟧ : Cont → Func
⟦ S ◁ P ⟧ = record { obj = λ A → (s : S) × (P s → A)
                   ; mor = λ m → λ { (s , f) → (s , m ∘ f) }
                   }
```

For example, the list functor arises from a container whose shapes are given by the natural numbers (representing the list's length) and the positions for a shape $n : \mathbb{N}$ are given by Fin n. This reflects the fact that a list of length $n : \mathbb{N}$ has Fin n locations or positions where data may be stored.

The motivation for containers was to find a representation of well-behaved functors. Since natural transformations are the semantic representation of polymorphic functions, it is also natural to seek a representation of natural transformations in the language of containers. We showed in our previous work that a natural transformation between two functors arising as containers can be represented as a morphism between containers as follows.

```
record _⇒_ (C D : Cont) : Set where
  constructor _◁_
  field
    f : C.S → D.S
    r : (s : C.S) → D.P (f s) → C.P s
```

As promised, such container morphisms represent natural transformations as the following definition shows:

```
⟦_⟧⃗ : ∀ {C D} → C ⇒ D → ∏ A . (⟦ C ⟧ A → ⟦ D ⟧ A)
⟦ f ◁ r ⟧⃗ (s , g) = f s , g ∘ r s
```

Rather surprisingly we were able to prove that the representation of natural transformations as container morphisms is a bijection, that is every natural transformation between functors arising from containers is uniquely represented as a container morphism. Technically, this can be stated by saying that container and their morphisms form a category which is a full and faithful sub-category of the functor category. We have also shown that the category of containers is cartesian closed (Altenkirch *et al.* 2010c), and is closed under formation of co-products, products and a number of other constructions. Most important of these is the fact that container functors (i.e. functors arising from containers) have initial algebras. Indeed, these are exactly the W-types we know well from type theory, which we can be equivalently defined to be:

```
data W (S : Set) (P : S → Set) : Set where
  sup : ⟦ S ◁ P ⟧ (W S P) → W S P
```

However, we have also shown that for n-ary containers (containers with n position sets) which we denote as Cont n. It is possible to define a *parameterized* initial algebra construction $\mu : \forall \{n\} \to \mathsf{Cont}\ (\mathsf{suc}\ n) \to \mathsf{Cont}\ n$. This allows us to model a broad range of nested and mutual types as containers. Further details can be found in the paper on containers cited above.

3 Indexed functors

While containers provide a robust framework for studying datatypes arising as initial algebras of functors over sets, indexed containers provide an equally robust framework for studying the more refined datatypes which arise as initial algebras of functors over indexed sets. Indeed, just as the essence of containers is a compact representation of well-behaved functors over sets, so the essence of indexed containers will be an equally compact representation of functors over indexed sets. Given $I : \mathsf{Set}$ we begin by considering the category of families over I. Its objects are I-indexed families of sets $A : I \to \mathsf{Set}$ and its morphisms are given by I-indexed families of functions. The definitions of morphisms, identity morphisms and composition of morphisms in this category are

```
_→⋆_ : {I : Set} → (A B : I → Set) → Set
_→⋆_ {I} A B = (i : I) → A i → B i

id⋆ : {I : Set} {A : I → Set} → A →⋆ A
id⋆ i a = a
```

```
_∘⋆_ : {I : Set} {A B C : I → Set} →
       (B →⋆ C) → (A →⋆ B) → (A →⋆ C)
f ∘⋆ g = λ i → (f i) ∘ (g i)
```

We call this category Fam I.[5] An I-indexed functor is then a functor from Fam I to Set, given by:

```
record IFunc (I : Set) : Set₁ where
  field
    obj : (A : Fam I) → Set
    mor : ∀ {A B} → (A →⋆ B) → obj A → obj B
```

such that both $\mathsf{id}^\star$ is mapped to id and $\_\circ^\star\_$ to $\_\circ\_$ under the action of mor. We adopt the convention that the projections obj and mor are silent, *i.e.* depending on the context F : IFunc I can stand for either the functor's action on objects, or on morphisms. A morphism between two such indexed functors is a natural transformation:

```
_⇒ᶠ_ : ∀ {I} → (F G : IFunc I) → Set₁
F ⇒ᶠ G = ∏ A . F A → G A
```

Our goal is, eventually, to give a representation for indexed functors as indexed containers. In doing this, we will also wish to represent structure on indexed functors as structure on indexed containers. To achieve this, we next look at the structure possessed by indexed functors. The main structure we wish to highlight for IFunc is the following monad-like structure:

```
ηᶠ : ∀ {I} → I → IFunc I
ηᶠ i = record {obj = λ A → A i; mor = λ f → f i}
_>>=ᶠ_ : ∀ {I J} → IFunc I → (I → IFunc J) → IFunc J
F >>=ᶠ H =
  record {obj = λ A → F (λ i → (H i) A)
          ; mor = λ f → F (λ i → (H i) f )}
```

It's clear that IFunc cannot be a monad in the usual sense, since it is not an endofunctor, but a functor from the category of small sets whose objects are elements of Set to the category of large sets whose objects are Set_1. However, it is an *relative monad* in the sense of Altenkirch *et al.* (2010) relative to the embedding functor $\uparrow : \mathsf{Set} \to \mathsf{Set}_1$. That is we have

```
ηᶠ : ∀ {I} → ↑ I → IFunc I
_>>=ᶠ_ : ∀ {I J} → IFunc I → (↑ I → IFunc J) → IFunc J
```

Note that in the code above we have elided the use of the lifting functor. The usual monad laws can be stated almost verbatim in this setting. On a more conceptual level, a relative monad is a monoid in the category of functors similar to ordinary monads being monoids

[5] This should not be confused with the usual notion of the category of families over a given base category, i.e. the families fibration.

in the category of endofunctors – for details please see Altenkirch *et al.* (2010). And indeed we can show:

Proposition 3.1
$(\mathsf{IFunc}, \eta^{\mathsf{F}}, \_ \gg\!=^{\mathsf{F}} \_)$ is a *relative monad* (Altenkirch *et al.* 2010) on the lifting functor $\uparrow : \mathsf{Set} \to \mathsf{Set}_1$.

Proof
To prove the structure is a relative monad, we observe that the following equalities hold up to Agda's $\beta\eta$-equality, and our postulate ext. (IFunc^* is defined below).

For $\mathsf{F} : \mathsf{IFunc}\ \mathsf{I}, \mathsf{G} : \mathsf{IFunc}^*\ \mathsf{J}\ \mathsf{I}, \mathsf{H} : \mathsf{IFunc}^*\ \mathsf{K}\ \mathsf{J}$:

$$\mathsf{H}\ \mathsf{i} \equiv (\eta^{\mathsf{F}}\ \mathsf{i}) \gg\!=^{\mathsf{F}} \mathsf{H} \tag{1}$$

$$\mathsf{F} \equiv \mathsf{F} \gg\!=^{\mathsf{F}} \eta^{\mathsf{F}} \tag{2}$$

$$(\mathsf{F} \gg\!=^{\mathsf{F}} \mathsf{G}) \gg\!=^{\mathsf{F}} \mathsf{H} \equiv \mathsf{F} \gg\!=^{\mathsf{F}} (\lambda\ \mathsf{i} \to (\mathsf{G}\ \mathsf{i}) \gg\!=^{\mathsf{F}} \mathsf{H}) \tag{3}$$

□

So far our indexed functors represent functors $\mathsf{Fam}\ \mathsf{I}$ to Set. However, since we want to construct fixpoints we are really interested in functors from $\mathsf{Fam}\ \mathsf{I}$ to $\mathsf{Fam}\ \mathsf{I}$, or actually to be able to take partial fixpoints, in functors from $\mathsf{Fam}\ (\mathsf{J} \uplus \mathsf{I})$ to $\mathsf{Fam}\ \mathsf{I}$. Hence to be appropriately general, we want to study functors $\mathsf{Fam}\ \mathsf{I}$ to $\mathsf{Fam}\ \mathsf{J}$. We will therefore define a type $\mathsf{IFunc}^\star$ of such doubly indexed functors and then investigate the structure possessed by such functors. Fortunately $\mathsf{IFunc}^\star$ can easily be derived from IFunc as follows. First, note that the opposite of the Kleisli category of the relative monad associated with IFunc has objects $\mathsf{I}, \mathsf{J} : \mathsf{Set}$ and morphisms given by J-indexed families of I-indexed functors. We denote this notion of indexed functor $\mathsf{IFunc}^\star$ and note that, as required, inhabitants of $\mathsf{IFunc}^\star$ are functors mapping indexed sets to indexed sets.

$$\begin{aligned}
&\mathsf{IFunc}^\star : (\mathsf{I}\ \mathsf{J} : \mathsf{Set}) \to \mathsf{Set}_1 \\
&\mathsf{IFunc}^\star\ \mathsf{I}\ \mathsf{J} = \mathsf{J} \to \mathsf{IFunc}\ \mathsf{I} \\
&\mathsf{obj}^\star : \forall\ \{\mathsf{I}\ \mathsf{J}\} \to \mathsf{IFunc}^\star\ \mathsf{I}\ \mathsf{J} \to \mathsf{Fam}\ \mathsf{I} \to \mathsf{Fam}\ \mathsf{J} \\
&\mathsf{obj}^\star\ \mathsf{F}\ \mathsf{A}\ \mathsf{j} = (\mathsf{F}\ \mathsf{j})\ \mathsf{A} \\
&\mathsf{mor}^\star : \forall\ \{\mathsf{I}\ \mathsf{J}\ \mathsf{A}\ \mathsf{B}\}\ (\mathsf{F} : \mathsf{IFunc}^\star\ \mathsf{I}\ \mathsf{J}) \to \\
&\qquad \mathsf{A} \to^\star \mathsf{B} \to \mathsf{obj}^\star\ \mathsf{F}\ \mathsf{A} \to^\star \mathsf{obj}^\star\ \mathsf{F}\ \mathsf{B} \\
&\mathsf{mor}^\star\ \mathsf{F}\ \mathsf{m}\ \mathsf{j} = (\mathsf{F}\ \mathsf{j})\ \mathsf{m}
\end{aligned}$$

Again, we will omit $\mathsf{obj}^\star$ and $\mathsf{mor}^\star$ when inferable from the context in which they appear. Natural transformations extend to this doubly indexed setting, too:

$$\begin{aligned}
&_ \Rightarrow^{\mathsf{F}^\star} _ : \forall\ \{\mathsf{I}\ \mathsf{J}\} \to (\mathsf{F}\ \mathsf{G} : \mathsf{IFunc}^\star\ \mathsf{I}\ \mathsf{J}) \to \mathsf{Set}_1 \\
&\mathsf{F} \Rightarrow^{\mathsf{F}^\star} \mathsf{G} = \prod \mathsf{A}\ .\ \mathsf{F}\ \mathsf{A} \to^\star \mathsf{G}\ \mathsf{A}
\end{aligned}$$

Turning to the structure on $\mathsf{IFunc}^\star$, clearly, the Kleisli structure gives rise to identities and composition in $\mathsf{IFunc}^\star$. Indeed, composition gives rise to a *re-indexing* operation which we denote Δ^{F}:

$$\begin{aligned}
&\Delta^{\mathsf{F}} : \forall\ \{\mathsf{I}\ \mathsf{J}\ \mathsf{K}\} \to (\mathsf{J} \to \mathsf{K}) \to \mathsf{IFunc}^\star\ \mathsf{I}\ \mathsf{K} \to \mathsf{IFunc}^\star\ \mathsf{I}\ \mathsf{J} \\
&\Delta^{\mathsf{F}}\ \mathsf{f}\ \mathsf{F} = \mathsf{F} \circ \mathsf{f}
\end{aligned}$$

This construction is used, for instance, in building the pattern functor for ScLam as in the introduction; Concentrating only on the abs case we want to build $\mathsf{ScLam}'\ \mathsf{X}\ \mathsf{n} = (\mathsf{X} \circ \mathsf{suc})\ \mathsf{n}$. Or simply $\mathsf{ScLam}'\ \mathsf{X} = \Delta^{\mathsf{F}}\ \mathsf{suc}\ \mathsf{X}$. In general, this combinator restricts the functor X to the indices in the image of the function f.

What if the restriction appears on the right of such an equation? As an example, consider the successor constructor for Fin; here we want to build the pattern functor: $\mathsf{FFin}'\ \mathsf{X}\ (1 + \mathsf{n}) = \mathsf{X}\ \mathsf{n}$. To do this, we observe that this is equivalent to the equation $\mathsf{FFin}'\ \mathsf{X}\ \mathsf{n} = (\mathsf{m} : \mathbb{N}) \times (\mathsf{n} \equiv 1 + \mathsf{m} \times \mathsf{X}\ \mathsf{m})$. We denote the general construction Σ^{F}, so the 2nd equation can be written $\mathsf{FFin}'\ \mathsf{X} = \Sigma^{\mathsf{F}}\ \mathsf{suc}\ \mathsf{X}$:

$$
\begin{array}{l}
\Sigma^{\mathsf{F}} : \forall\ \{\mathsf{J}\ \mathsf{I}\ \mathsf{K}\} \to (\mathsf{J} \to \mathsf{K}) \to \mathsf{IFunc}^{\star}\ \mathsf{I}\ \mathsf{J} \to \mathsf{IFunc}^{\star}\ \mathsf{I}\ \mathsf{K} \\
\Sigma^{\mathsf{F}}\ \{\mathsf{J}\}\ \mathsf{f}\ \mathsf{F}\ \mathsf{k} = \\
\quad \mathbf{record}\ \{\mathsf{obj} = \lambda\ \mathsf{A} \to (\mathsf{j} : \mathsf{J}) \times (\mathsf{f}\ \mathsf{j} \equiv \mathsf{k} \times \mathsf{F}\ \mathsf{A}\ \mathsf{j}) \\
\qquad\quad ; \mathsf{mor} = \lambda\ \{\mathsf{m}\ (\mathsf{j}, \mathsf{p}, \mathsf{x}) \to (\mathsf{j}, \mathsf{p}, \mathsf{F}\ \mathsf{m}\ \mathsf{j}\ \mathsf{x})\} \\
\qquad\quad \}
\end{array}
$$

Perhaps unsurprisingly, Σ^{F} turns out to be the left adjoint to reindexing (Δ^{F}). Its right adjoint, we denote Π^{F}:

$$
\begin{array}{l}
\Pi^{\mathsf{F}} : \forall\ \{\mathsf{J}\ \mathsf{I}\ \mathsf{K}\} \to (\mathsf{J} \to \mathsf{K}) \to \mathsf{IFunc}^{\star}\ \mathsf{I}\ \mathsf{J} \to \mathsf{IFunc}^{\star}\ \mathsf{I}\ \mathsf{K} \\
\Pi^{\mathsf{F}}\ \{\mathsf{J}\}\ \mathsf{f}\ \mathsf{F}\ \mathsf{k} = \\
\quad \mathbf{record}\ \{\mathsf{obj} = \lambda\ \mathsf{A} \to (\mathsf{j} : \mathsf{J}) \to \mathsf{f}\ \mathsf{j} \equiv \mathsf{k} \to \mathsf{F}\ \mathsf{A}\ \mathsf{j} \\
\qquad\quad ; \mathsf{mor} = \lambda\ \mathsf{m}\ \mathsf{g}\ \mathsf{j}\ \mathsf{p} \to \mathsf{F}\ \mathsf{m}\ \mathsf{j}\ (\mathsf{g}\ \mathsf{j}\ \mathsf{p})\}
\end{array}
$$

Proposition 3.2
Σ^{F} and Π^{F} are left and right adjoint to reindexing (Δ^{F}).

Proof
To show this, we need to show that for all $\mathsf{f} : \mathsf{J} \to \mathsf{K}$, $\mathsf{g} : \mathsf{K} \to \mathsf{J}$, $\mathsf{F} : \mathsf{IFunc}^{\star}\ \mathsf{I}\ \mathsf{J}$ and $\mathsf{G} : \mathsf{IFunc}^{\star}\ \mathsf{I}\ \mathsf{K}$:

$$
\frac{\Sigma^{\mathsf{F}}\ \mathsf{f}\ \mathsf{F} \Rightarrow^{\mathsf{F}^{\star}} \mathsf{G}}{\mathsf{F} \Rightarrow^{\mathsf{F}^{\star}} \Delta^{\mathsf{F}}\ \mathsf{f}\ \mathsf{G}} \qquad \frac{\Delta^{\mathsf{F}}\ \mathsf{f}\ \mathsf{F} \Rightarrow^{\mathsf{F}^{\star}} \mathsf{G}}{\mathsf{F} \Rightarrow^{\mathsf{F}^{\star}} \Pi^{\mathsf{F}}\ \mathsf{f}\ \mathsf{G}}
$$

We can build the components of these isomorphisms easily:

$$
\begin{array}{l}
\Sigma \dashv \Delta : (\mathsf{f} : \mathsf{J} \to \mathsf{K}) \to (\Sigma^{\mathsf{F}}\ \mathsf{f}\ \mathsf{F} \Rightarrow^{\mathsf{F}^{\star}} \mathsf{G}) \to (\mathsf{F} \Rightarrow^{\mathsf{F}^{\star}} \Delta^{\mathsf{F}}\ \mathsf{f}\ \mathsf{G}) \\
\Sigma \dashv \Delta\ \mathsf{f}\ \mathsf{m}\ \mathsf{j}\ \mathsf{x} = \mathsf{m}\ (\mathsf{f}\ \mathsf{j})\ (\mathsf{j}, \mathsf{refl}, \mathsf{x}) \\
\Sigma \dashv \Delta^{-1} : (\mathsf{f} : \mathsf{J} \to \mathsf{K}) \to (\mathsf{F} \Rightarrow^{\mathsf{F}^{\star}} \Delta^{\mathsf{F}}\ \mathsf{f}\ \mathsf{G}) \to (\Sigma^{\mathsf{F}}\ \mathsf{f}\ \mathsf{F} \Rightarrow^{\mathsf{F}^{\star}} \mathsf{G}) \\
\Sigma \dashv \Delta^{-1}\ \mathsf{f}\ \mathsf{m}\ .\ (\mathsf{f}\ \mathsf{j})\ (\mathsf{j}, \mathsf{refl}, \mathsf{x}) = \mathsf{m}\ \mathsf{j}\ \mathsf{x} \\
\Delta \dashv \Pi : (\mathsf{g} : \mathsf{K} \to \mathsf{J}) \to (\Delta^{\mathsf{F}}\ \mathsf{g}\ \mathsf{F} \Rightarrow^{\mathsf{F}^{\star}} \mathsf{G}) \to (\mathsf{F} \Rightarrow^{\mathsf{F}^{\star}} \Pi^{\mathsf{F}}\ \mathsf{g}\ \mathsf{G}) \\
\Delta \dashv \Pi\ \mathsf{g}\ \mathsf{m}\ .\ (\mathsf{g}\ \mathsf{k})\ \mathsf{x}\ \mathsf{k}\ \mathsf{refl} = \mathsf{m}\ \mathsf{k}\ \mathsf{x} \\
\Delta \dashv \Pi^{-1} : (\mathsf{g} : \mathsf{K} \to \mathsf{J}) \to (\mathsf{F} \Rightarrow^{\mathsf{F}^{\star}} \Pi^{\mathsf{F}}\ \mathsf{g}\ \mathsf{G}) \to (\Delta^{\mathsf{F}}\ \mathsf{g}\ \mathsf{F} \Rightarrow^{\mathsf{F}^{\star}} \mathsf{G}) \\
\Delta \dashv \Pi^{-1}\ \mathsf{g}\ \mathsf{m}\ \mathsf{k}\ \mathsf{x} = \mathsf{m}\ (\mathsf{g}\ \mathsf{k})\ \mathsf{x}\ \mathsf{k}\ \mathsf{refl}
\end{array}
$$

It only remains to observe that these pairs of functions are mutual inverses, which is a simple proof. □

In abstracting over all possible values for the extra indexing information Π^{F} allows for the construction of infinitely branching trees, such as rose trees. We also note that finite co-products and products can be derived from Σ^{F} and Π^{F}, respectively:

```
⊥ᶠ : ∀ {I} → IFunc⋆ I ⊤
⊥ᶠ = Σᶠ {J = ⊥} _ λ ()
_⊎ᶠ_ : ∀ {I} → (F G : IFunc I) → IFunc⋆ I ⊤
F ⊎ᶠ G = Σᶠ _ λ b → if b then F else G
⊤ᶠ : ∀ {I} → IFunc⋆ I ⊤
⊤ᶠ = Πᶠ {J = ⊥} _ λ ()
_×ᶠ_ : ∀ {I} → (F G : IFunc I) → IFunc⋆ I ⊤
F ×ᶠ G = Πᶠ _ λ b → if b then F else G
```

Clearly these are simply the constantly $\top$ and $\bot$ valued functors, and the pointwise product and co-product of functors. However, encoding them using Σ^{F} and Π^{F} allows us to keep to a minimum the language of indexed functors (and hence indexed containers) with obvious benefits in terms of tractability.

3.1 Initial algebras of indexed functors

We have seen that an `F : IFunc⋆ I I` is an endofunctor on the category `Fam I`. Using this observation, we know that an algebra of such a functor is a family `A : Fam I` and a map α : `F A →⋆ A`. A morphism, then, between two such algebras (A, α) and (B, β) is a map `f : A →⋆ B` such that the follow diagram commutes:

$$\begin{array}{ccc} \mathsf{F\,A} & \xrightarrow{\alpha} & \mathsf{A} \\ {\scriptstyle \mathsf{F\,f}}\downarrow & & \downarrow{\scriptstyle \mathsf{f}} \\ \mathsf{F\,B} & \xrightarrow[\beta]{} & \mathsf{B} \end{array}$$

This defines the category of F-algebras. If it exists, then the initial algebra of F is the initial object of the category of F-algebras spelled out above. It follows from the fact that not all functors in `Set → Set` (for instance `F A = (A → Bool) → Bool`) have initial algebras that neither do all indexed functors.

We also know that we cannot iterate the construction of initial algebras given above. That is, an endofunctor `IFunc⋆ I I` gives rise to an initial algebra in `Fam I`, and we cannot take the initial algebra of something in `Fam I`. This prevents us from being able to define nested inductive families in this way.

We finish our study of indexed functors by tackling this problem. Our strategy is as follows: First note that for a singly indexed functor over a co-product we can eliminate the

co-product and curry the resulting definition in this way:

$$\begin{aligned} \mathsf{IFunc}\ (\mathsf{I} \uplus \mathsf{J}) \equiv & \quad (\mathsf{I} \uplus \mathsf{J} \to \mathsf{Set}) \to \mathsf{Set} \\ \iff & \quad (\mathsf{I} \to \mathsf{Set}) \times (\mathsf{J} \to \mathsf{Set}) \to \mathsf{Set} \\ \iff & \quad (\mathsf{I} \to \mathsf{Set}) \to (\mathsf{J} \to \mathsf{Set}) \to \mathsf{Set} \end{aligned}$$

This gives us partial application for indexed functors of the form $\mathsf{IFunc}\ (\mathsf{I} \uplus \mathsf{J})$. Spelled out concretely we have:

$$\begin{aligned} &\_[\_]^{\mathsf{F}} : \forall\, \{\mathsf{I}\ \mathsf{J}\} \to \mathsf{IFunc}\ (\mathsf{I} \uplus \mathsf{J}) \to \mathsf{IFunc}^\star\ \mathsf{I}\ \mathsf{J} \to \mathsf{IFunc}\ \mathsf{I} \\ &\mathsf{F}\ [\ \mathsf{G}\ ]^{\mathsf{F}} \quad = \\ &\quad \mathbf{record}\ \{\ \mathsf{obj}\ = \lambda\ \mathsf{A} \to \mathsf{F}\ [\mathsf{A},\ \mathsf{G}\ \mathsf{A}] \\ &\qquad\quad ;\ \mathsf{mor} = \lambda\ \mathsf{f}\ \ \to \mathsf{F}\ [\mathsf{f}\ ,\ \mathsf{G}\ \mathsf{f}\ ]\} \end{aligned}$$

This construction is functorial:

$$\begin{aligned} &\_[\_]^{\mathsf{F}} : \forall\, \{\mathsf{I}\ \mathsf{J}\}\ (\mathsf{F} : \mathsf{IFunc}\ (\mathsf{I} \uplus \mathsf{J}))\ \{\mathsf{G}\ \mathsf{H} : \mathsf{IFunc}^\star\ \mathsf{I}\ \mathsf{J}\} \\ &\qquad\quad \to \quad \mathsf{G} \quad \Rightarrow^{\mathsf{F}^\star} \quad \mathsf{H} \\ &\qquad\quad \to \mathsf{F}\ [\ \mathsf{G}\ ]^{\mathsf{F}} \Rightarrow^{\mathsf{F}} \mathsf{F}\ [\ \mathsf{H}\ ]^{\mathsf{F}} \\ &\mathsf{F}\ [\ \gamma\ ]^{\mathsf{F}} = \mathsf{F}\ [(\lambda\ \_\,\mathsf{a} \to \mathsf{a}), \gamma] \end{aligned}$$

Each of these definitions generalizes to $\mathsf{IFunc}^\star$:

$$\begin{aligned} &\_[\_]^{\mathsf{F}^\star} : \forall\, \{\mathsf{I}\ \mathsf{J}\ \mathsf{K}\} \to \mathsf{IFunc}^\star\ (\mathsf{I} \uplus \mathsf{J})\ \mathsf{K} \to \mathsf{IFunc}^\star\ \mathsf{I}\ \mathsf{J} \to \mathsf{IFunc}^\star\ \mathsf{I}\ \mathsf{K} \\ &\mathsf{F}\ [\ \mathsf{G}\ ]^{\mathsf{F}^\star} = \lambda\ \mathsf{k} \to (\mathsf{F}\ \mathsf{k})\ [\ \mathsf{G}\ ]^{\mathsf{F}} \\ &\_[\_]^{\mathsf{F}^\star} : \forall\, \{\mathsf{I}\ \mathsf{J}\ \mathsf{K}\}\ (\mathsf{F} : \mathsf{IFunc}^\star\ (\mathsf{I} \uplus \mathsf{J})\ \mathsf{K})\ \{\mathsf{G}\ \mathsf{H} : \mathsf{IFunc}^\star\ \mathsf{I}\ \mathsf{J}\} \\ &\qquad\quad \to \quad \mathsf{G} \quad \Rightarrow^{\mathsf{F}^\star} \quad \mathsf{H} \\ &\qquad\quad \to \mathsf{F}\ [\ \mathsf{G}\ ]^{\mathsf{F}^\star} \Rightarrow^{\mathsf{F}^\star} \mathsf{F}\ [\ \mathsf{H}\ ]^{\mathsf{F}^\star} \\ &\_[\_]^{\mathsf{F}^\star}\ \mathsf{F}\ \{\mathsf{G}\}\ \{\mathsf{H}\}\ \gamma = \lambda\ \mathsf{k} \to \_[\_]^{\mathsf{F}}\ (\mathsf{F}\ \mathsf{k})\ \{\mathsf{G}\}\ \{\mathsf{H}\}\ \gamma \end{aligned}$$

A parametrized F-algebra for $\mathsf{F} : \mathsf{IFunc}^\star\ (\mathsf{I} \uplus \mathsf{J})\ \mathsf{J}$ is then simply an algebra for the functor $\mathsf{F}\ [\ \_\ ]^{\mathsf{F}^\star}$. That is, a parameterized F-algebra consists of a pair of an indexed-functor $\mathsf{G} : \mathsf{IFunc}\ \mathsf{I}\ \mathsf{J}$ and a natural transformation $\alpha : \mathsf{F}\ [\ \mathsf{G}\ ]^{\mathsf{F}^\star} \Rightarrow^{\mathsf{F}^\star} \mathsf{G}$. A morphism between two such algebras (G, α) and (H, β) is a natural transformation $\gamma : \mathsf{G} \Rightarrow^{\mathsf{F}^\star} \mathsf{H}$ such that the follow diagram commutes:

$$\begin{array}{ccc} \mathsf{F}\ [\ \mathsf{G}\ ]^{\mathsf{F}^\star} & \xrightarrow{\ \alpha\ } & \mathsf{G} \\ {\scriptstyle \mathsf{F}\ [\ \gamma\ ]^{\mathsf{F}^\star}}\downarrow & & \downarrow{\scriptstyle \gamma} \\ \mathsf{F}\ [\ \mathsf{H}\ ]^{\mathsf{F}^\star} & \xrightarrow[\ \beta\ ]{} & \mathsf{H} \end{array}$$

As you might expect, a parametrized initial algebra for F, if it exists, will be the initial object in the category of parametrized F-algebras. Alternatively, it is the initial $\mathsf{F}\ [\ \_\ ]^{\mathsf{F}^\star}$-algebra. Either way, the parameterized initial algebra construction will map indexed functors to indexed functors and hence can be iterated. This means that we can define nested and mutual families of datatypes, such as the tuple of neutral and normal λ-terms outlined in the introduction.

However, it is still the case that not all indexed functors in $\mathsf{IFunc}^\star\ (\mathsf{I} \uplus \mathsf{J})\ \mathsf{I}$ have parameterized initial algebras. In the analogous situation for functors on Set, we solved this problem by limiting ourselves to those functors which can be represented by containers. We follow a similar approach in the indexed setting, that is, we restrict our attention to those indexed functors which can be represented by indexed containers. We show that all indexed containers have parameterized initial algebras and that, surprisingly, initial algebras may be constructed using only the W-types used to construct initial algebras of containers.

4 Indexed containers

Following the structure of the previous section, we first define singly indexed containers which will represent singly indexed functors, and then we define doubly indexed containers which will represent doubly indexed functors. To this end, we define an I-indexed container to be given by a set of shapes, and an I-indexed *family* of positions:

```
record ICont (I : Set) : Set₁ where
  constructor _◁_
  field
    S : Set
    P : S → I → Set
```

The above definition shows that an I-indexed container is similar to a container in that it has a set of shapes whose elements can be thought of as constructors. However, the difference between an I-indexed container and a container lies in the notion of the positions associated to a given shape. In the case of a container, the positions for a given shape simply form a set. In the case of an I-indexed container, the positions for a given shape form an I-indexed set. If we think of I as a collection of sorts, then not only does the constructor require input to be stored at its positions, but each of these positions is tagged with an $\mathsf{i} : \mathsf{I}$ and will only store data of sort $\mathsf{i} : \mathsf{I}$ at that position. This intuition is formalized by the following definition which shows how singly indexed containers represent singly indexed functors

```
⟦_⟧ : ∀ {I} → ICont I → IFunc I
⟦_⟧ {I} (S ◁ P) =
  record {obj = λ A → (s : S) × (P s →* A)
          ; mor = λ {m (s , f) → (s , m ∘* f)}
          }
```

Notice how the extension of an indexed container is very similar to the extension of a container. In particular, an element of $\llbracket\ \mathsf{S} \lhd \mathsf{P}\ \rrbracket\ \mathsf{A}$ consists of a shape $\mathsf{s} : \mathsf{S}$ and a morphism $\mathsf{P\ s} \to^\star \mathsf{A}$ of I-indexed sets. This latter function assigns to each $\mathsf{i} : \mathsf{I}$, and each position $\mathsf{p} : \mathsf{P\ s\ i}$ an element of $\mathsf{A\ i}$. If we think of I as a collection of sorts, then this function assigns to each $\mathsf{i} : \mathsf{I}$-sorted position, an i-sorted piece of data, i.e. an element of $\mathsf{A\ i}$.

Analogously to the generalization of singly indexed functors to doubly indexed functors, we can generalize singly indexed containers to doubly indexed containers. Indeed, a doubly indexed container, that is an element of $\mathsf{ICont}^\star\ \mathsf{I}\ \mathsf{J}$, is simply a function from J to ICont I.

Unpacking the definition of such a function gives us the following definition of a doubly indexed container and its extension as a doubly indexed functor:

```
record ICont* (I J : Set) : Set₁ where
  constructor _◁*_
  field
    S : J → Set
    P : (j : J) → S j → I → Set
⟦_⟧* : ∀ {I J} → ICont* I J → IFunc* I J
⟦ S ◁* P ⟧* j = ⟦ S j ◁ P j ⟧
```

We will denote the two projections for an ICont postfix as _.S and _.P. Our methodology of reflecting structure on indexed functors as structure on indexed containers means we must next consider how to reflect morphisms between indexed functors which can be represented by indexed containers as morphisms between those indexed containers. We begin by considering what constitutes a natural transformation between the extension of an indexed container and an arbitrary indexed functor. We do this in the singly indexed case as follows:

$$
\begin{aligned}
& \llbracket\, S \lhd P \,\rrbracket \Rightarrow^{F} F && (1)\\
\equiv\ & \prod X\,.\,(s : S) \times (P\,s \to^{\star} X) \to F\,X && \{\text{by definition}\}\\
\Longleftrightarrow\ & \prod X\,.\,(s : S) \to (P\,s \to^{\star} X) \to F\,X && \{\text{currying}\}\\
\Longleftrightarrow\ & (s : S) \to \prod X\,.\,(P\,s \to^{\star} X) \to F\,X && \{\text{commuting end and pi}\}\\
\Longleftrightarrow\ & (s : S) \to F\,(P\,s) && \{\text{Yoneda}\}
\end{aligned}
$$

Now, if F is the extension of an indexed container $T \lhd Q$, we have:

$$
\begin{aligned}
& \llbracket\, S \lhd P \,\rrbracket \Rightarrow^{F} \llbracket\, T \lhd Q \,\rrbracket && (2)\\
\Longleftrightarrow\ & (s : S) \to (t : T) \times (Q\,t \to^{\star} P\,s)\\
\Longleftrightarrow\ & (f : S \to T) \times ((s : S) \to Q\,(f\,s) \to^{\star} P\,s)
\end{aligned}
$$

We will use this last line as the definition for indexed container morphisms. This definition can be implemented by the following record type, containing a function on shapes and a family of contravariant indexed functions on positions:

```
record _⇒ᶜ_ {I} (C D : ICont I) : Set where
  constructor _◁_
  field
    f : C.S → D.S
    r : (s : C.S) → (D.P (f s)) →* (C.P s)
```

ICont I forms a category, with morphisms given by $\_ \Rightarrow^{c} \_$, the identity and composition morphisms are given as follows:

```
id^c : ∀ {I} {C : ICont I} → C ⇒^c C
id^c = id ◁ (λ _ _ → id)
_∘^c_ : ∀ {I} {C D E : ICont I} →
          D ⇒^c E → C ⇒^c D → C ⇒^c E
(f ◁ r) ∘^c (g ◁ q) = (f ∘ g) ◁ (λ s → q s ∘* r (g s))
```

That id^{c} is the left and right unit of $\circ^{\mathsf{c}}$, and that $\circ^{\mathsf{c}}$ is associative follows immediately from the corresponding properties of id and $\_\circ\_$.

We will use a notion of equality for container morphisms that includes a proof that their shape and position functions are pointwise equal:

```
record _ ≡^⇒ _ {I} {C D : ICont I} (m n : C ⇒^c D) : Set where
  constructor _◁_
  field
    feq : (s : C.S) → m.f s ≡ n.f s
    req : (s : C.S) (i : I) (p : D.P (m.f s) i) →
      m.r s i p ≡
        n.r s i (subst (λ s' → D.P s' i) (feq s) p)
```

In the presence of extensional equality, we can prove that this is equivalent to the propositional equality on $\_ \Rightarrow^{\mathsf{c}} \_$, but it will prove simpler later to use this definition.

We witness the construction of a natural transformation from an indexed container morphisms as follows:

```
⟦_⟧^⇒ : ∀ {I} {C D : ICont I} (m : C ⇒^c D) →
          ∏ A . ⟦ C ⟧ A → ⟦ D ⟧ A
⟦ f ◁ r ⟧^⇒ (s , g) = f s , g ∘* r s
```

The representation of natural transformations between indexed functors arising from indexed containers and morphisms between the indexed containers themselves is actually a bijection. This opens the way to reasoning about natural transformations by reasoning about indexed container morphisms. Technically, this bijection is captured by the following statement:

Proposition 4.1
The functor $(\llbracket \_ \rrbracket, \llbracket \_ \rrbracket^{\Rightarrow}) : \mathsf{ICont\ I} \to \mathsf{IFunc\ I}$ is full and faithful.

Proof
The isomorphism is proved in Equations (1) and (2).
□

Having dealt with indexed container morphisms in the singly indexed setting, we now turn to the doubly indexed setting. First of all, we define the morphisms between two doubly indexed containers.

```
record _ ⇒^c* _ {I J} (C D : ICont* I J) : Set₁ where
  constructor _◁*_
  field
    f : C.S →* D.S
```

```
      r : {j : J} (s : C.S j) → (D.P j (f j s)) →⋆ (C.P j s)
⟦_⟧⇒* : ∀ {I J} {C D : ICont⋆ I J} (m : C ⇒c⋆ D) →
          ∏ A . (⟦ C ⟧⋆ A →⋆ ⟦ D ⟧⋆ A)
⟦ f ◁⋆ r ⟧⇒⋆ j = ⟦ f j ◁ r ⟧⇒
```

Having defined indexed containers and indexed container morphisms as representations of indexed functors and the natural transformations between them, we now turn our attention to the relative monad structure on indexed functors, reindexing of indexed functors (and the associated adjoints), and parameterized initial algebras of indexed functors. Our goal in the rest of this section is to encode each of these structures within indexed containers. We begin by showing that, as with IFunc, we can equip ICont with a relative monadic structure:

```
ηc : ∀ {I} → I → ICont I
ηc i = ⊤ ◁ λ _ i′ → i ≡ i′
_>>=c_ : ∀ {I J} → ICont I → ICont⋆ J I → ICont J
_>>=c_ {I} (S ◁ P) (T ◁⋆ Q) =
    (⟦ S ◁ P ⟧ T)
  ◁ λ { (s , f) j → Σ ((i : I) × P s i) (λ { (i , p) → Q i (f i p) j }) }
```

Proposition 4.2
The triple $(\mathsf{ICont}, \eta^{\mathsf{c}}, \_ \gg\!=^{\mathsf{c}} \_)$ is a relative monad.

Proof
Instead of proving this directly, we observe that the η^{c} and $\_ \gg\!=^{\mathsf{c}} \_$ are preserved under the extension functor, that is that the following natural isomorphisms hold:

$$\prod X\,.\,[\![\, \eta^{\mathsf{c}}\, i \,]\!]\, X \iff \eta^{\mathsf{F}}\, i\, X$$
$$\prod X\,.\,[\![\, C \gg\!=^{\mathsf{c}} D \,]\!]\, X \iff ([\![C]\!]^{\star} \gg\!=^{\mathsf{F}} [\![D]\!])\, X$$

Which follows from the extensionality of our propositional equality, the associativity of Σ and the terminality of $\top$. By the full and faithful nature of the embedding $[\![\_]\!]$, we can then reuse the result that $(\mathsf{IFunc}, \eta^{\mathsf{F}}, \_ \gg\!=^{\mathsf{F}} \_)$ is a relative monad to establish the theorem.
□

As with indexed functors, the re-indexing functor Δ^{c} on indexed containers is defined by composition, and it has left and right adjoints Σ^{c} and Π^{c}. As we shall see, our proof of this fact uses the full and faithfulness of the embedding of indexed containers as indexed functors and the fact that reindexing of indexed functors has left and right adjoints.

```
Δc : (J → K) → ICont⋆ I K → ICont⋆ I J
Δc f F = λ k → F (f k)
Σc : (J → K) → ICont⋆ I J → ICont⋆ I K
Σc f (S ◁⋆ P) = λ k →
      ((j : J) × (f j ≡ k × S j))
    ◁ λ { (j , eq , s) → P j s }
Πc : (J → K) → ICont⋆ I J → ICont⋆ I K
```

$$\begin{aligned}&\Pi^{\mathsf{c}}\,\mathsf{f}\,(\mathsf{S} \lhd^{\star} \mathsf{P}) = \lambda\,\mathsf{k} \to \\ &\quad ((\mathsf{j} : \mathsf{J}) \to \mathsf{f}\,\mathsf{j} \equiv \mathsf{k} \to \mathsf{S}\,\mathsf{j}) \\ &\lhd\ \lambda\,\mathsf{s}\,\mathsf{i} \to (\mathsf{j} : \mathsf{J}) \times ((\mathsf{eq} : \mathsf{f}\,\mathsf{j} \equiv \mathsf{k}) \times \mathsf{P}\,\mathsf{j}\,(\mathsf{s}\,\mathsf{j}\,\mathsf{eq})\,\mathsf{i})\end{aligned}$$

Proposition 4.3
Σ^{c} and Π^{c} are left and right adjoint to reindexing (Δ^{c}).

Proof
Again, we appeal to the full and faithfulness of $[\![\_]\!]$ and show instead that $[\![\_]\!]$ also preserves these constructions. That is we want to show that the following three natural isomorphisms hold:

$$\begin{aligned}&\textstyle\prod \mathsf{X} . [\![\, \Sigma^{\mathsf{c}}\,\mathsf{f}\,\mathsf{F}\,\mathsf{j}\,]\!]\,\mathsf{X} \iff \Sigma^{\mathsf{F}}\,\mathsf{f}\,[\![\,\mathsf{F}\,]\!]^{\star}\,\mathsf{j}\,\mathsf{X} \\ &\textstyle\prod \mathsf{X} . [\![\, \Delta^{\mathsf{c}}\,\mathsf{f}\,\mathsf{F}\,\mathsf{j}\,]\!]\,\mathsf{X} \iff \Delta^{\mathsf{F}}\,\mathsf{f}\,[\![\,\mathsf{F}\,]\!]^{\star}\,\mathsf{j}\,\mathsf{X} \\ &\textstyle\prod \mathsf{X} . [\![\, \Pi^{\mathsf{c}}\,\mathsf{f}\,\mathsf{F}\,\mathsf{j}\,]\!]\,\mathsf{X} \iff \Pi^{\mathsf{F}}\,\mathsf{f}\,[\![\,\mathsf{F}\,]\!]^{\star}\,\mathsf{j}\,\mathsf{X}\end{aligned}$$

These can be proved simply by employing the associativity of Σ. □

Before we build the initial algebras of indexed containers, it will help to define their partial application.

$$\begin{aligned}&\_[\_]^{\mathsf{c}} : \forall\,\{\mathsf{I}\,\mathsf{J}\} \to \mathsf{ICont}\,(\mathsf{I} \uplus \mathsf{J}) \to \mathsf{ICont}^{\star}\,\mathsf{I}\,\mathsf{J} \to \mathsf{ICont}\,\mathsf{I} \\ &\_[\_]^{\mathsf{c}}\,\{\mathsf{I}\}\,\{\mathsf{J}\}\,(\mathsf{S} \lhd \mathsf{P})\,(\mathsf{T} \lhd^{\star} \mathsf{Q}) = \\ &\quad \textbf{let}\ \mathsf{P}^{\mathsf{I}} : \mathsf{S} \to \mathsf{I} \to \mathsf{Set}; \mathsf{P}^{\mathsf{I}}\,\mathsf{s}\,\mathsf{i} = \mathsf{P}\,\mathsf{s}\,(\mathsf{inl}\,\mathsf{i}) \\ &\qquad\ \ \mathsf{P}^{\mathsf{J}} : \mathsf{S} \to \mathsf{J} \to \mathsf{Set}; \mathsf{P}^{\mathsf{J}}\,\mathsf{s}\,\mathsf{j} = \mathsf{P}\,\mathsf{s}\,(\mathsf{inr}\,\mathsf{j}) \\ &\quad \textbf{in}\ \ [\![\, \mathsf{S} \lhd \mathsf{P}^{\mathsf{J}}\,]\!]\,\mathsf{T} \lhd \\ &\qquad\qquad (\lambda\,\{(\mathsf{s},\mathsf{f})\,\mathsf{i} \to \mathsf{P}^{\mathsf{I}}\,\mathsf{s}\,\mathsf{i} \\ &\qquad\quad \uplus\ ((\mathsf{j} : \mathsf{J}) \times ((\mathsf{p} : \mathsf{P}^{\mathsf{J}}\,\mathsf{s}\,\mathsf{j}) \times \mathsf{Q}\,\mathsf{j}\,(\mathsf{f}\,\mathsf{j}\,\mathsf{p})\,\mathsf{i}))\})\end{aligned}$$

The composite container has shapes given by a shape $\mathsf{s} : \mathsf{S}$ and an assignment of T shapes to $\mathsf{P}^{\mathsf{J}}\,\mathsf{s}$ positions. Positions are then a choice between a I-indexed position, or a pair of an J-indexed position, and a Q position *underneath* the appropriate T shape.
As with indexed functors, this construction is functorial in its second argument, and lifts container morphisms in this way:

$$\begin{aligned}&\_[\_]^{\mathsf{c}} : \forall\,\{\mathsf{I}\,\mathsf{J}\}\,(\mathsf{C} : \mathsf{ICont}\,(\mathsf{I} \uplus \mathsf{J}))\,\{\mathsf{D}\,\mathsf{E} : \mathsf{ICont}^{\star}\,\mathsf{I}\,\mathsf{J}\} \to \\ &\qquad\qquad \mathsf{D} \quad \Rightarrow^{\mathsf{c}\star} \quad \mathsf{E} \\ &\qquad\quad \to \mathsf{C}\,[\,\mathsf{D}\,]^{\mathsf{c}} \Rightarrow^{\mathsf{c}} \mathsf{C}\,[\,\mathsf{E}\,]^{\mathsf{c}} \\ &\mathsf{C}\,[\,\gamma\,]^{\mathsf{c}} = \\ &\quad (\lambda\,\{(\mathsf{s},\mathsf{f}) \to (\mathsf{s}, \lambda\,\mathsf{j}\,\mathsf{p} \to \gamma.\mathsf{f}\,\mathsf{j}\,(\mathsf{f}\,\mathsf{j}\,\mathsf{p}))\}) \lhd \\ &\quad\ \lambda\,\{(\mathsf{s},\mathsf{f})\,\mathsf{i} \to \mathsf{id} \uplus \lambda\,\{(\mathsf{j},\mathsf{p},\mathsf{q}) \to (\mathsf{j},\mathsf{p},\gamma.\mathsf{r}\,\mathsf{j}\,(\mathsf{f}\,\mathsf{j}\,\mathsf{p})\,\mathsf{i}\,\mathsf{q})\}\}\end{aligned}$$

5 Initial algebras of indexed containers

We will now examine how to construct the parameterized initial algebra of an indexed container of the form $\mathsf{F} : \mathsf{ICont}^{\star}\,(\mathsf{I} \uplus \mathsf{J})\,\mathsf{J}$. The shapes of such a container are an J-indexed family of Sets and the positions are indexed by $\mathsf{I} \uplus \mathsf{J}$; we will treat these position as two

separate entities, those positions indexed by J – the recursive positions – and those by I – the payload positions.

The shapes of the initial algebra we are constructing will be trees with S shapes at the nodes and which branch over the recursive P^{J} positions. We call these trees *indexed* W-types, denoted WI, and they are the initial algebra of the functor $[\![\, \mathsf{S} \lhd \mathsf{P}^{\mathsf{J}} \,]\!]^{\star}$. In Agda, we can implement the WI constructor and its associated iteration operator WIfold as follows:

```
data WI {J : Set} (S : J → Set)
        (P^J : (j : J) → S j → J → Set) : J → Set where
  sup : ⟦ S ◁* P^J ⟧* (WI S P^J) →* WI S P^J
```

Proposition 5.1
$(\mathsf{WI}\ \mathsf{S}\ \mathsf{P}^{\mathsf{J}}, \mathsf{sup})$ is the initial object in the category of $[\![\, \mathsf{S} \lhd \mathsf{P}^{\mathsf{J}} \,]\!]$-algebras.

Proof
We show this by constructing the iteration operator WIfold, a morphism in the category of $[\![\, \mathsf{S} \lhd \mathsf{P}^{\mathsf{J}} \,]\!]$-algebras from our candidate initial algebra to any other algebra such that the following diagram commutes:

$$\begin{array}{ccc} [\![\, \mathsf{S} \lhd^{\star} \mathsf{P}^{\mathsf{J}} \,]\!]^{\star}\ (\mathsf{WI}\ \mathsf{S}\ \mathsf{P}^{\mathsf{J}}) & \xrightarrow{\mathsf{sup}} & \mathsf{WI}\ \mathsf{S}\ \mathsf{P}^{\mathsf{J}} \\ \Big\downarrow {\scriptstyle [\![\, \mathsf{S} \lhd^{\star} \mathsf{P}^{\mathsf{J}} \,]\!]^{\star}\ (\mathsf{WIfold}\ \alpha)} & & \Big\downarrow {\scriptstyle \mathsf{WIfold}\ \alpha} \\ [\![\, \mathsf{S} \lhd^{\star} \mathsf{P}^{\mathsf{J}} \,]\!]^{\star}\ \mathsf{X} & \xrightarrow{\alpha} & \mathsf{X} \end{array}$$

In fact we can use this specification as the definition of WIfold:

```
WIfold : ∀ {J} {S X : J → Set} {P^J} →
         ⟦ S ◁* P^J ⟧* X →* X →
         WI S P^J →* X
WIfold {S = S} {P^J = P^J} α j (sup ._ x) =
  α j ( ⟦ S ◁* P^J ⟧* (WIfold α) j x)
```

We also require that WIfold is *unique*, that is we must show that any morphism β which makes the diagram above commute must be equal to WIfold α:

```
WIfoldUniq : ∀ {J} {X : J → Set} {S : J → Set}
             {P^J : (j : J) → S j → J → Set}
             (α : ⟦ S ◁* P^J ⟧* X →* X)
             (β : WI S P^J →* X) →
             ((j : J) (s : ⟦ S ◁* P^J ⟧* (WI S P^J) j) →
               (β j (sup j s)) ≡ (α j ( ⟦ S ◁* P^J ⟧* β j s))) →
             (j : J) (x : WI S P^J j) → β j x ≡ WIfold α j x
WIfoldUniq α β commβ j (sup .j (s , g)) = begin
    β j (sup j (s , g))
  ≅⟨ commβ j (s , g) ⟩
    α j (s , (λ j' p' → β j' (g j' p')))
  ≅⟨ cong (λ f → α j (s , f))
          (λ≡ j' → λ≡ p' → WIfoldUniq α β commβ j' (g j' p')) ⟩
```

$\alpha\ \mathsf{j}\ (\mathsf{s}, (\lambda\ \mathsf{j}'\ \mathsf{p}'\ \rightarrow\ \mathsf{WIfold}\ \alpha\ \mathsf{j}'\ (\mathsf{g}\ \mathsf{j}'\ \mathsf{p}')))$

■

The above definition proves that β and WIfold α are pointwise equal, by employing ext we can show that WIfoldUniq′ implies that they are extensionally equal. □

This proof mirrors the construction for ordinary containers, where we can view ordinary W-types as the initial algebra of a container functor. Positions in an indexed W-type are given by the paths through such a tree which terminate in a non-recursive P^{I}-position:

```
data Path {I J : Set} (S : J → Set)
          (P^I : (j : J) → S j → I → Set)
          (P^J : (j : J) → S j → J → Set)
          : (j : J) → WI S P^J j → I → Set where
  path : ∀ {j s f i} →
           P^I j s i
         ⊎ ((j' : J) × ((p : P^J j s j') × Path S P^I P^J j' (f j' p) i))
         → Path S P^I P^J j (sup _ (s , f)) i
pathh : ∀ {I J : Set} (S : J → Set)
  (P^I : (j : J) → S j → I → Set)
  (P^J : (j : J) → S j → J → Set)
  {j s f i} →
           P^I j s i
         ⊎ ((j' : J) × ((p : P^J j s j') × Path S P^I P^J j' (f j' p) i))
         → Path S P^I P^J j (sup _ (s , f)) i
pathh S P^I P^J x = path x
```

Again this mirrors the partial application construction where positions were given by a P^{I} position at the top level, or a pair of a P^{J} position and a recursive Path position. This reflects the fact that a WI-type can be thought of as iterated partial application. We can now use WI-types, or equivalently initial algebras of indexed containers, to construct the parametrized initial algebra of an indexed container. First, we construct the carrier of the parameterized initial algebra:

```
μ^c : {I J : Set} → ICont* (I ⊎ J) J → ICont* I J
μ^c {I} {J} (S ◁* P) =
  let P^I : (j : J) → S j → I → Set; P^I j s i = P j s (inl i)
      P^J : (j : J) → S j → J → Set; P^J j s j' = P j s (inr j')
  in WI S P^J ◁* Path S P^I P^J
```

Next, we note that the structure map for this parameterized initial algebra is a container morphism from the partial application of F and its parametrized initial algebra to the parameterized initial algebra. This structure map is given by the constructor sup of WI and the deconstructor for Path:

```
in^c : ∀ {I J} → (F : ICont* (I ⊎ J) J) → F [ μ^c F ]^c* ⇒^c* μ^c F
in^c F = sup ◁* λ {_ _ (path p) → p}
```

Proposition 5.2
$(\mu^{\mathsf{c}}\ \mathsf{F}, \mathsf{in}^{\mathsf{c}}\ \mathsf{F})$ is initial in the category of parameterized F-algebras of indexed containers. Further, by full and faithfulness, $(\llbracket\ \mu^{\mathsf{c}}\ \mathsf{F}\ \rrbracket^{\star}, \llbracket\ \mathsf{in}^{\mathsf{c}}\ \mathsf{F}\ \rrbracket^{\Rightarrow^{\star}})$ will also be initial in the indexed functor case.

To show this, we must define an operator $\mathsf{fold}^{\mathsf{c}}$ from the initial algebra to an arbitrary algebra. The shape map employs the fold for WI directly. For the position map we apply the position map for the algebra, which maps Q positions to either a P position in the first layer, or a recursive Q position – it is straightforward to recursively employ this position map to construct the corresponding Path to a P position *somewhere* in the tree.

```
fold^c : ∀ {I J} (F : ICont* (I ⊎ J) J) {G : ICont* I J} →
         F [ G ]^c* ⇒^c* G → μ^c F ⇒^c* G
fold^c {I} {J} (S ◁* P) {T ◁* Q} (f ◁* r) = ffold ◁* rfold
  where P^I : (j : J) → S j → I → Set; P^I j s i = P j s (inl i)
        P^J : (j : J) → S j → J → Set; P^J j s j' = P j s (inr j')
        ffold = WIfold f
        rfold : {j : J} (s : WI S P^J j)
                (i : I) → Q j (ffold j s) i → Path S P^I P^J j s i
        rfold (sup ._ (s, g)) i p =
          path ((id ⊎ (λ jpq → (_, π0 (π1 jpq)
                               , rfold _ _ (π1 (π1 jpq))))) (r (s, _) i p))
```

We also need to show that the following diagram commutes for any parametrized F-algebra (G, α):

$$\begin{array}{ccc} \mathsf{F}\ [\ \mu^{\mathsf{c}}\ \mathsf{F}\ ]^{\mathsf{c}\star} & \xrightarrow{\mathsf{in}^{\mathsf{c}}\ \mathsf{F}} & \mu^{\mathsf{c}}\ \mathsf{F} \\ \Big\downarrow{\scriptstyle \mathsf{F}\ [\ (\mathsf{fold}^{\mathsf{c}}\ \mathsf{F}\ \alpha)\ ]^{\mathsf{F}^{\star}}} & & \Big\downarrow{\scriptstyle \mathsf{fold}^{\mathsf{c}}\ \mathsf{F}\ \alpha} \\ \mathsf{F}\ [\ \mathsf{G}\ ]^{\mathsf{c}\star} & \xrightarrow{\alpha} & \mathsf{G} \end{array}$$

Or, equivalently:

```
foldComm : ∀ {I J} {F : ICont* (I ⊎ J) J} (G : ICont* I J)
           (α : F [ G ]^c* ⇒^c* G) →
           (fold^c F α ∘^c* in^c F) ≡⇒* (α ∘^c* F [ (fold^c F α) ]^c*)
foldComm {F} G α = (λ j x → refl) ◁* (λ j x i p → refl)
```

All that remains for us to show in order to prove that $(\mu^{\mathsf{c}}\ \mathsf{F}, \mathsf{in}^{\mathsf{c}}\ \mathsf{F})$ is the initial parametrized F-algebra is to show that $\mathsf{fold}^{\mathsf{c}}\ \mathsf{F}\ \alpha$ is *unique* for any α. That is any morphism $\beta : \mu^{\mathsf{c}}\ \mathsf{F} \Rightarrow^{\mathsf{c}\star} \mathsf{G}$, that makes the above diagram commute, must be $\mathsf{fold}^{\mathsf{c}}\ \mathsf{F}\ \alpha$:

```
foldUniq : ∀ {I J} {F : ICont* (I ⊎ J) J} (G : ICont* I J)
           (α : F [ G ]^c* ⇒^c* G) (β : μ^c F ⇒^c* G) →
           (β ∘^c* in^c F) ≡⇒* (α ∘^c* F [ β ]^c*) →
           β ≡⇒* (fold^c F α)
foldUniq {I} {J} {S ◁* P} (T ◁* Q)
         (αf ◁* αr) (βf ◁* βr) (feq ◁* req) =
  WIfoldUniq αf βf feq ◁* rfoldUniq
```

```
  where
    P^I : (j : J) → S j → I → Set; P^I j s i  = P j s (inl i)
    P^J : (j : J) → S j → J → Set; P^J j s j' = P j s (inr j')
```

That the shape maps of β and $\mathsf{fold^c}\ \mathsf{F}\ \alpha$ agree follows from the uniqueness of WIfold; while the proof that the position maps agree follows the same inductive structure as rfold in the definition of $\mathsf{fold^c}$. [6]

```
rfoldUniq : (j : J) (s : WI S P^J j) (i : I)
            (p : Q j (βf j s) i) →
            βr s i p ≅
                rfold S P^I P^J (T ◁* Q) αf αr s i
                  (subst (λ s → Q j s i)
                         (WIfoldUniq αf βf feq j s) p)
rfoldUniq j (sup ._ y) i p with req j y i p
rfoldUniq j (sup ._ y) i p | reqjyip with βr (sup j y) i p
rfoldUniq j (sup ._ y) i p | reqjyip | path q = begin
    path q    -- βr (sup j y) i p
  ≅⟨ cong path reqjyip ⟩
    path ((id ⊎ (λ jpq → (π₀ jpq, π₀ (π₁ jpq)
                         , βr (π₁ y (π₀ jpq) (π₀ (π₁ jpq))) i
                           (π₁ (π₁ jpq)))))
                 (αr (π₀ y, (λ j' p' → βf j' (π₁ y j' p'))) i
                   (subst (λ s' → Q j s' i) (feq j y) p)))
  ≅⟨ cong ... (λ≅ j' → λ≅ p' → λ≅ q' → begin
               βr (π₁ y _ _) _ _
             ≅⟨ rfoldUniq _ (π₁ y _ _) i _ ⟩
               rfold S P^I P^J (T ◁* Q) αf αr (π₁ y _ _) i
                 (subst (λ s → Q _ s i)
                        (WIfoldUniq αf βf feq _ (π₁ y _ _)) _)
             ≅⟨ ... ⟩
               rfold S P^I P^J (T ◁* Q) αf αr (π₁ y _ _) i _ ■) ... ⟩
    path ((id ⊎ (λ jpq → (π₀ jpq, π₀ (π₁ jpq)
                         , rfold S P^I P^J (T ◁* Q) αf αr
                             (π₁ y (π₀ jpq) (π₀ (π₁ jpq))) i
                             (π₁ (π₁ jpq)))))
                 (αr (π₀ y, (λ j p → WIfold αf j (π₁ y j p))) i
                   (subst (λ s → Q j s i)
                          (WIfoldUniq αf βf feq _ (sup _ y)) p)))
  ≅⟨ refl ⟩
    rfold S P^I P^J (T ◁* Q) αf αr (sup _ y) i
      (subst (λ s → Q _ s i)
             (WIfoldUniq αf βf feq _ (sup _ y)) p) ■
```

[6] Some parts of the Agda proof are hidden and denoted by ....

6 Terminal coalgebras of indexed containers

Dually to the initial algebra construction outlined above, we can also show that indexed containers are closed under parameterized terminal coalgebras. We proceed in much the same way as before, by first constructing the dual of the indexed W-type, which we refer to as an indexed M-type. As you might expect this is in fact the plain (as opposed to parametrized) terminal coalgebra of an indexed container:

```
data MI {I : Set} (S : I → Set)
        (P^I : (i : I) → S i → I → Set) : I → Set where
  sup : ⟦ S ◁* P^I ⟧* (λ i → ∞ (MI S P^I i)) →* MI S P^I
sup⁻¹ : ∀ {I S} {P^I : (i : I) → S i → I → Set} →
        MI S P^I →* ⟦ S ◁* P^I ⟧* (MI S P^I)
sup⁻¹ (sup (s,f)) = s, λ i p → ♭ (f i p)
```

Here, we employ Agda's approach to coprogramming (e.g. see (Danielsson & Altenkirch 2010)), where we mark (possibly) infinite subtrees with ∞. The type ∞ A is a suspended computation of type A, and ♯ : A → ∞ A delays a value of type A and ♭ : ∞ A → A forces a computation. A simple syntactic test then ensures that co-recursive programs are total – recursive calls must be immediately *guarded* by a ♯ constructor.

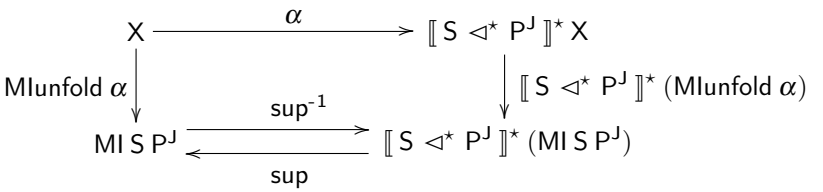
The equality between infinite objects will be bi-simulation, for instance MI. Types are bi-similar if they have the same node shape, and all their sub-trees are bi-similar:

```
data _ ≈^MI _ {J S P^J} {j : J} : (x y : MI S P^J j) → Set where
  sup : ∀ {s f g} → (∀ {j'} (p : P^J j s j') →
                     ∞ (♭ (f j' p) ≈^MI ♭ (g j' p))) →
                   sup (s,f) ≈^MI sup (s,g)
```

It is simple to show that this bi-simulation is an equivalence relation.

Proposition 6.1
(MI S P^J, sup⁻¹) is the terminal object in the category of ⟦ S ◁* P^J ⟧-coalgebras.

We must construct a co-iteration operator MIunfold, a morphism in the category of ⟦ S ◁* P^J ⟧-coalgebras to our candidate terminal coalgebra from any other coalgebra such that the following diagram commutes:



The following definition of MIunfold makes the diagram commute up-to bi-simulation.

```
MIunfold : ∀ {J S P^J} {X : J → Set} →
           X →* ⟦ S ◁* P^J ⟧* X → X →* MI S P^J
MIunfold α j x with α j x
MIunfold α j x | s,f = sup (s, λ j' p → ♯ MIunfold α j' (f j' p))
```

We also require that MIunfold is unique, *i.e.* any morphism that makes the diagram above commute should be provably equal (again upto bi-simulation) to MIunfold α. To state this property we need to lift the bi-simulation $\_ \approx^{\mathsf{MI}} \_$ through the extension of an indexed container, to say what is it for two elements in the extension to be bi-similar:

$$
\begin{aligned}
&\_ \approx^{[\![\_]\!]^{\mathsf{MI}}} \_ : \forall \{J : \mathsf{Set}\}\,\{S : J \to \mathsf{Set}\} \\
&\qquad \{P^J : (j : J) \to S\,j \to J \to \mathsf{Set}\}\,\{j : J\} \to \\
&\qquad (x\,y : [\![\, S \lhd^\star P^J \,]\!]^\star\,(\mathsf{MI}\,S\,P^J)\,j) \to \mathsf{Set} \\
&\_ \approx^{[\![\_]\!]^{\mathsf{MI}}} \_\,\{J\}\,\{S\}\,\{P^J\}\,\{j\}\,(s,f)\,(s',f') = \\
&\quad \Sigma\,(s \equiv s')\,\lambda\,\mathsf{eq} \to \{j' : J\}\,(p : P^J\,j\,s\,j') \to \\
&\qquad\qquad f\,\_\,p \approx^{\mathsf{MI}} f'\,\_\,(\mathsf{subst}\,(\lambda\,s \to P^J\,j\,s\,j')\,\mathsf{eq}\,p)
\end{aligned}
$$

The uniqueness property is then given by:

$$
\begin{aligned}
&\mathsf{MIunfoldUniq} : \forall \{J\}\,\{X : J \to \mathsf{Set}\}\,\{S\,P^J\} \\
&\quad (\alpha : X \to^\star [\![\, S \lhd^\star P^J \,]\!]^\star\,X) \to (\beta : X \to^\star \mathsf{MI}\,\{J\}\,S\,P^J) \to \\
&\quad ((j : J)\,(x : X\,j) \to \\
&\qquad (\mathsf{sup}^{-1}\,(\beta\,j\,x)) \approx^{[\![\_]\!]^{\mathsf{MI}}} ((\,[\![\, S \lhd^\star P^J \,]\!]^\star\,\beta \circ^\star \alpha)\,j\,x)) \to \\
&\quad (j : J)\,(x : X\,j) \to \beta\,j\,x \approx^{\mathsf{MI}} \mathsf{MIunfold}\,\alpha\,j\,x \\
&\mathsf{MIunfoldUniq}\,\alpha\,\beta\,\mathsf{comm}\beta\,i\,x \;\mathbf{with}\; \mathsf{comm}\beta\,i\,x \\
&\mathsf{MIunfoldUniq}\,\alpha\,\beta\,\mathsf{comm}\beta\,i\,x \mid \mathsf{commix} \;\mathbf{with}\; \beta\,i\,x \\
&\mathsf{MIunfoldUniq}\,\alpha\,\beta\,\mathsf{comm}\beta\,i\,x \mid (\mathsf{refl},y) \mid \mathsf{sup}\,(.\,(\pi_0\,(\alpha\,i\,x)),g) = \\
&\quad \mathsf{sup}\,(\lambda\,p \to \sharp\,{\approx}\mathsf{MItrans}\,(y\,p)\,(\mathsf{MIunfoldUniq}\,\alpha\,\beta\,\mathsf{comm}\beta\,\_\,\_))
\end{aligned}
$$

However, Agda rejects this definition due to the recursive call not being guarded immediately by the $\sharp$, however, it is productive due to the fact that the proof of transitivity of bi-simulation is contractive. We can persuade the system this is productive by fusing the definition of ≈MItrans with this MIunfoldUniq in a cumbersome but straightforward way. The paths to positions in an indexed M-tree are always finite – in fact modulo the use of $\flat$, this Path is the same as the definition for the initial algebra case.

$$
\begin{aligned}
&\mathbf{data}\ \mathsf{Path}\,\{I\,J : \mathsf{Set}\}\,(S : J \to \mathsf{Set}) \\
&\qquad (P^I : (j : J) \to S\,j \to I \to \mathsf{Set}) \\
&\qquad (P^J : (j : J) \to S\,j \to J \to \mathsf{Set}) \\
&\qquad : (j : J) \to \mathsf{MI}\,S\,P^J\,j \to I \to \mathsf{Set}\ \mathbf{where} \\
&\quad \mathsf{path} : \forall \{j\,s\,f\,i\} \to \\
&\qquad\qquad P^I\,j\,s\,i \\
&\qquad\qquad \uplus\,((j' : J) \times \\
&\qquad\qquad\quad ((p : P^J\,j\,s\,j') \times \mathsf{Path}\,S\,P^I\,P^J\,j'\,(\flat\,(f\,j'\,p))\,i)) \\
&\qquad\qquad \to \mathsf{Path}\,S\,P^I\,P^J\,j\,(\mathsf{sup}\,(s,f))\,i
\end{aligned}
$$

Just as parameterized initial algebras of indexed containers are built from WI-types, so parameterized terminal coalgebras of indexed containers are built from MI-types as follows:

$$
\begin{aligned}
&\nu^{\mathsf{c}} : \{I\,J : \mathsf{Set}\} \to \mathsf{ICont}^\star\,(I \uplus J)\,J \to \mathsf{ICont}^\star\,I\,J \\
&\nu^{\mathsf{c}}\,\{I\}\,\{J\}\,(S \lhd^\star P) = \\
&\quad \mathbf{let}\ P^I : (j : J) \to S\,j \to I \to \mathsf{Set}; P^I\,j\,s\,i = P\,j\,s\,(\mathsf{inl}\,i)
\end{aligned}
$$

```
    P^J : (j : J) → S j → J → Set; P^J j s j' = P j s (inr j')
  in MI S P^J ◁* Path S P^I P^J

out^C : ∀ {I J} → (F : ICont* (I ⊎ J) J) → ν^C F ⇒^C* F [ ν^C F ]^C*
out^C {I} {J} (S ◁* P) = (λ _ → sup⁻¹) ◁* outr
  where outr : {j : J} (s : (ν^C (S ◁* P) .S) j) →
               ((((S ◁* P) [ ν^C (S ◁* P) ]^C*) .P) j (sup⁻¹ s)) →*
               ((ν^C (S ◁* P)) .P j s)
        outr (sup s) i' p = path p
```

Proposition 6.2
$(\nu^{\mathsf{C}}\ \mathsf{F}, \mathsf{out}^{\mathsf{C}}\ \mathsf{F})$ is the terminal object in the category of parametrized F-coalgebras of indexed containers. By full and faithfulness, $(\llbracket\ \nu^{\mathsf{C}}\ \mathsf{F}\ \rrbracket^{\star}, \llbracket\ \mathsf{out}^{\mathsf{C}}\ \mathsf{F}\ \rrbracket^{\Rightarrow^{\star}})$ will also be terminal in the indexed functor case.

Proof
Mirroring the case of initial algebras, the coiteration for this terminal co-algebra employs the coiteration of MI for the shape maps. The position map takes a Path and builds a Q position by applying the position map from the coalgebra at every step in the path – note that this position map is *inductive* in its path argument.

```
unfold^C : ∀ {I J} (F : ICont* (I ⊎ J) J) {G : ICont* I J} →
           G ⇒^C* F [ G ]^C* → G ⇒^C* ν^C F
unfold^C {I} {J} (S ◁* P) {T ◁* Q} (f ◁* r) = funfold ◁* runfold
  where P^I : (j : J) → S j → I → Set; P^I j s i = P j s (inl i)
        P^J : (j : J) → S j → J → Set; P^J j s j' = P j s (inr j')
        funfold = MIunfold f
        runfold : {j : J} (t : T j)
                  (i : I) → Path S P^I P^J j (funfold j t) i → Q j t i
        runfold t i (path p) =
          r t i ([inl
                 , (λ y → inr (_ , π₀ (π₁ y)
                               , runfold (π₁ (f _ t) _ _) i (π₁ (π₁ y))))] p)
```

We must then show that $\mathsf{unfold}^{\mathsf{C}}$ is the unique morphism that makes the following diagram commute:

$$\begin{array}{ccc} \mathsf{G} & \xrightarrow{\ \alpha\ } & \mathsf{F}\ [\ \mathsf{G}\ ]^{\mathsf{C}\star} \\ {\scriptstyle \mathsf{F}\ [\ (\mathsf{unfold}^{\mathsf{C}}\ \mathsf{F}\ \alpha)\ ]^{\mathsf{F}^\star}}\downarrow & & \downarrow{\scriptstyle \mathsf{unfold}^{\mathsf{C}}\ \mathsf{F}\ \alpha} \\ \nu^{\mathsf{C}}\ \mathsf{F} & \xrightarrow{\ \mathsf{out}^{\mathsf{C}}\ \mathsf{F}\ } & \mathsf{F}\ [\ \nu^{\mathsf{C}}\ \mathsf{F}\ ]^{\mathsf{C}\star} \end{array}$$

As with the initial algebra case, this follows immediately from the definition:

```
unfoldComm : ∀ {I J} {F : ICont* (I ⊎ J) J} (G : ICont* I J)
             (α : G ⇒^C* F [ G ]^C*) →
             (out^C F ∘^C* unfold^C F α) ≡⇒*
```

$$((\mathsf{F}\ [\ (\mathsf{unfold}^{\mathsf{c}}\ \mathsf{F}\ \alpha)\ ]^{\mathsf{c}\star})\ \circ^{\mathsf{c}\star}\ \alpha)$$
$$\mathsf{unfoldComm}\ (\mathsf{S} \lhd^{\star} \mathsf{P})\ (\mathsf{f} \lhd^{\star} \mathsf{r}) = (\lambda\ \mathsf{j}\ \mathsf{s} \rightarrow \mathsf{refl}) \lhd^{\star} (\lambda\ \mathsf{j}\ \mathsf{s}\ \mathsf{i}\ \mathsf{p} \rightarrow \mathsf{refl})$$

We also have to show that the $\mathsf{unfold}^{\mathsf{c}}$ is *unique*; that is, any morphism that makes the above diagram commute must be equal to $\mathsf{unfold}^{\mathsf{c}}\ \mathsf{F}\ \alpha$.

In order to show this in Agda, we are assuming a second extensionality principle namely that if two MI trees are bi-similar, then they are in fact equal:

$$\begin{aligned}&\textbf{postulate}\ \mathsf{MIext} : \forall\ \{\mathsf{J}\ \mathsf{S}\ \mathsf{P}^{\mathsf{J}}\}\ \{\mathsf{j} : \mathsf{J}\}\ \{\mathsf{x}\ \mathsf{y} : \mathsf{MI}\ \mathsf{S}\ \mathsf{P}^{\mathsf{J}}\ \mathsf{j}\} \rightarrow \\ &\qquad\qquad \mathsf{x} \approx^{\mathsf{MI}} \mathsf{y} \rightarrow \mathsf{x} \cong \mathsf{y}\end{aligned}$$

The inverse of this principle is obviously true:

$$\begin{aligned}&\mathsf{MIext}^{-1} : \forall\ \{\mathsf{J}\ \mathsf{S}\ \mathsf{P}^{\mathsf{J}}\}\ \{\mathsf{j} : \mathsf{J}\}\ \{\mathsf{x}\ \mathsf{y} : \mathsf{MI}\ \mathsf{S}\ \mathsf{P}^{\mathsf{J}}\ \mathsf{j}\} \rightarrow \\ &\qquad\qquad \mathsf{x} \cong \mathsf{y} \rightarrow \mathsf{x} \approx^{\mathsf{MI}} \mathsf{y} \\ &\mathsf{MIext}^{-1}\ \mathsf{refl} = \approx\mathsf{MIrefl}\end{aligned}$$

It is reasonable to assume that any language with fully-fledged support for co-inductive types *and* extensional equality would admit such an axiom.

We can now state the property that $\mathsf{unfold}^{\mathsf{c}}$ is, indeed, unique:

$$\begin{aligned}&\mathsf{unfoldUniq} : \forall\ \{\mathsf{I}\ \mathsf{J}\}\ \{\mathsf{F} : \mathsf{ICont}^{\star}\ (\mathsf{I} \uplus \mathsf{J})\ \mathsf{J}\}\ (\mathsf{G} : \mathsf{ICont}^{\star}\ \mathsf{I}\ \mathsf{J}) \\ &\qquad (\alpha : \mathsf{G} \Rightarrow^{\mathsf{c}\star} \mathsf{F}\ [\ \mathsf{G}\ ]^{\mathsf{c}\star})\ (\beta : \mathsf{G} \Rightarrow^{\mathsf{c}\star} \nu^{\mathsf{c}}\ \mathsf{F}) \rightarrow \\ &\qquad (\mathsf{out}^{\mathsf{c}}\ \mathsf{F} \circ^{\mathsf{c}\star} \beta) \equiv\Rightarrow^{*} (\mathsf{F}\ [\ \beta\ ]^{\mathsf{c}\star} \circ^{\mathsf{c}\star} \alpha) \rightarrow \\ &\qquad \beta \equiv\Rightarrow^{*} (\mathsf{unfold}^{\mathsf{c}}\ \mathsf{F}\ \alpha)\end{aligned}$$

The proof that the shape maps agree follows from the proof that MIunfold is unique, and the proof that the position maps agree follows the same inductive structure as runfold. Unfortunately, because Agda lacks full support for both co-induction and extensional equality it is not feasible to complete the proof terms for these propositions in our Agda development. The main obstacle remains mediating between bi-simulation, the (functional) extensional equality and Agda's built-in notion of equality. We have completed this proof on paper, however, and we are hopeful that soon we may be in a position to complete these proof terms in a system where the built-in equality is sensible for both functions and co-inductive types. □

7 W is still enough

So far, we have developed a theory of indexed containers using a rich type theory with features such as WI- and MI-types. We claimed in the introduction, however, that the theory of indexed containers could be developed even when one only has W-types. In this section, we will outline the translation of many of the definitions above into such a spartan theory. First, we will show how to obtain indexed WI-types from W-types, and by analogy MI-types from M-types, and then we will revisit our proof of how to derive M-types from W-types.

7.1 WI from W

How, then, can we build WI-types from W-types? The initial step is to create a type of *pre*-WI trees, with nodes containing a shape *and* its index, and branching over positions *and their* indices:

$$\begin{aligned}
&\mathsf{WI'} : \{\mathsf{I} : \mathsf{Set}\}\,(\mathsf{S} : \mathsf{I} \to \mathsf{Set}) \\
&\qquad (\mathsf{P} : (\mathsf{i} : \mathsf{I})\,(\mathsf{s} : \mathsf{S\,i}) \to \mathsf{I} \to \mathsf{Set}) \to \mathsf{Set} \\
&\mathsf{WI'}\,\{\mathsf{I}\}\,\mathsf{S}\,\mathsf{P} = \mathsf{W}\,((\mathsf{i} : \mathsf{I}) \times \mathsf{S\,i})\,(\lambda\,\{(\mathsf{i},\mathsf{s}) \to (\mathsf{i'} : \mathsf{I}) \times \mathsf{P\,i\,s\,i'}\})
\end{aligned}$$

Given such a tree, we want to express the property that the subtrees of such a pre-tree have the correct index in their node information. In order to do this we need a second W-type, which is similar to WI′, but with an extra copy of the index information stored in that node:

$$\begin{aligned}
&\mathsf{WII} : \{\mathsf{I} : \mathsf{Set}\}\,(\mathsf{S} : \mathsf{I} \to \mathsf{Set}) \\
&\qquad (\mathsf{P} : (\mathsf{i} : \mathsf{I})\,(\mathsf{s} : \mathsf{S\,i}) \to \mathsf{I} \to \mathsf{Set}) \to \mathsf{Set} \\
&\mathsf{WII}\,\{\mathsf{I}\}\,\mathsf{S}\,\mathsf{P} = \mathsf{W}\,(\mathsf{I} \times ((\mathsf{i} : \mathsf{I}) \times \mathsf{S\,i})) \\
&\qquad\qquad (\lambda\,\{(\mathsf{i'},\mathsf{i},\mathsf{s}) \to (\mathsf{i'} : \mathsf{I}) \times \mathsf{P\,i\,s\,i'}\})
\end{aligned}$$

There are two canonical ways to turn an element of WI' S P into an element of WII S P, both of which involve filling in this extra indexing information: (i) we can simply copy the index already stored at the node; or (ii) we can push the indexes down from parent nodes to child nodes:

$$\begin{aligned}
&\mathsf{lup} : \mathsf{WI'\,S\,P} \to \mathsf{WII\,S\,P} \\
&\mathsf{lup}\,(\mathsf{sup}\,((\mathsf{i},\mathsf{s}),\mathsf{f})) = \mathsf{sup}\,((\mathsf{i},(\mathsf{i},\mathsf{s})),(\lambda\,\mathsf{p} \to \mathsf{lup}\,(\mathsf{f\,p})))
\end{aligned}$$

$$\begin{aligned}
&\mathsf{ldown} : \mathsf{I} \to \mathsf{WI'\,S\,P} \to \mathsf{WII\,S\,P} \\
&\mathsf{ldown\,i}\,(\mathsf{sup}\,(\mathsf{s},\mathsf{f})) = \mathsf{sup}\,((\mathsf{i},\mathsf{s}),\lambda\,\{(\mathsf{i'},\mathsf{p}) \to \mathsf{ldown\,i'}\,(\mathsf{f}\,(\mathsf{i'},\mathsf{p}))\})
\end{aligned}$$

The property of a pre-tree being type correct can be stated as its two possible labelings being equal. That is we can use W-types to define the WI-type as follows:

$$\begin{aligned}
&\mathsf{WI} : \{\mathsf{I} : \mathsf{Set}\}\,(\mathsf{S} : \mathsf{I} \to \mathsf{Set}) \\
&\qquad (\mathsf{P} : (\mathsf{i} : \mathsf{I})\,(\mathsf{s} : \mathsf{S\,i}) \to \mathsf{I} \to \mathsf{Set}) \to \mathsf{I} \to \mathsf{Set} \\
&\mathsf{WI\,S\,P\,i} = \\
&\quad (\mathsf{x} : (\mathsf{WI'\,S\,P}))\times \\
&\qquad \mathsf{lup}\,\{_\}\,\{\mathsf{S}\}\,\{\mathsf{P}\}\,\mathsf{x} \equiv \mathsf{ldown}\,\{_\}\,\{\mathsf{S}\}\,\{\mathsf{P}\}\,\mathsf{i\,x}
\end{aligned}$$

Having built the WI-type from the W-type, we must next build the constructor sup which makes elements of WI-types. We rely on function extensionality to define the constructor sup:

$$\begin{aligned}
&\mathsf{sup} : \forall\,\{\mathsf{J\,S\,P^J}\} \to [\![\,\mathsf{S} \lhd^\star \mathsf{P^J}\,]\!]^\star\,(\mathsf{WI}\,\{\mathsf{J}\}\,\mathsf{S\,P^J}) \to^\star \mathsf{WI\,S\,P^J} \\
&\mathsf{sup}\,\{\mathsf{J}\}\,\{\mathsf{S}\}\,\{\mathsf{P^J}\}\,\mathsf{j}\,(\mathsf{s},\mathsf{f}) = \\
&\quad (\mathsf{sup}\,((_,\mathsf{s}),\lambda\,\{(\mathsf{j},\mathsf{p}) \to \pi_0\,(\mathsf{f\,j\,p})\})) \\
&\quad ,\mathsf{cong}\,(\lambda\,\mathsf{x} \to \mathsf{sup}\,((\mathsf{j},\mathsf{j},\mathsf{s}),\mathsf{x}))\,(\lambda^{=}\,\mathsf{ip} \to \pi_1\,(\mathsf{f}\,_\,(\pi_1\,\mathsf{ip})))
\end{aligned}$$

Proposition 7.1
$(\mathsf{WI\,S\,P^J},\mathsf{sup})$ is the initial object in the category of $[\![\,\mathsf{S} \lhd^\star \mathsf{P^J}\,]\!]$-algebras.

Proof
We must once again show that for any $[\![\ \mathsf{S} \lhd^\star \mathsf{P}^\mathsf{J}\ ]\!]$-algebra (X, α) where $\alpha : [\![\ \mathsf{S} \lhd^\star \mathsf{P}^\mathsf{J}\ ]\!]^\star\ \mathsf{X} \to^\star \mathsf{X}$ there is a unique mediating morphism $\mathsf{WIfold} : \mathsf{WI\ S\ P}^\mathsf{J} \to^\star \mathsf{X}$. It is simple enough to define WIfold:

```
WIfold : ∀ {J} {S X : J → Set} {P^J} →
           [[ S ◁* P^J ]]* X →* X →
           WI S P^J →* X
WIfold α j (sup ((j′,s),f),ok) with cong (π₀ ∘ π₀ ∘ sup⁻¹) ok
WIfold α j (sup ((.j,s),f),ok) | refl =
α j (s,(λ j′ p → WIfold α j′ (f (j′,p),ext⁻¹ (cong (π₁ ∘ sup⁻¹) ok) (j′,p))))
```

In the form below, WIfold does not pass Agda's termination checker; the direct encoding via Wfold would avoid this problem, at the expense of being even more verbose.

To show that WIfold makes the initial algebra diagram commute, we must employ the UIP principle, that any two proofs of an equality are equal:

```
WIcomm : ∀ {J} {S X : J → Set} {P^J}
           (α : [[ S ◁* P^J ]]* X →* X)
           (j : J) → (x : [[ S ◁* P^J ]]* (WI S P^J) j) →
           WIfold α j (sup {J} {S} {P^J} j x) ≡
             α j ([[ S ◁* P^J ]]* (WIfold α) j x)
WIcomm α j (s,f) with
  (cong (π₀ ∘ π₀ ∘ sup⁻¹)
    (cong (λ x → sup ((j,j,s),x))
      (λ≡ ip → π₁ (f (π₀ ip) (π₁ ip)))))
WIcomm α j (s,f) | refl =
  cong (λ g → α j (s,g))
    (λ≡ j′ → λ≡ p →
      cong (λ eq → WIfold α j′ (π₀ (f j′ p),eq)) UIP)
```

We can also show that the fold is unique:

```
WIfoldUniq′ : ∀ {J} {X : J → Set} {S : J → Set}
                {P^J : (j : J) → S j → J → Set}
                (α : [[ S ◁* P^J ]]* X →* X)
                (β : WI S P^J →* X) →
                (β ∘* sup) ≡ (α ∘* [[ S ◁* P^J ]]* β) →
                (j : J) (x : WI S P^J j) → β j x ≡ WIfold α j x
WIfoldUniq′ α β commβ j (sup ((j′,s),f),ok)
  with cong (π₀ ∘ π₀ ∘ sup⁻¹) ok
WIfoldUniq′ α β commβ j (sup ((.j,s),f),ok) | refl = begin
    β j (sup ((j,s),f),ok)
  ≅⟨ cong (λ ok′ → β j (sup ((j,s),f),ok′)) UIP ⟩
    β j (sup ((j,s),f)
        ,cong (λ p → sup ((j,j,s),p))
            (ext (ext⁻¹ (cong (π₁ ∘ sup⁻¹) ok))))
```

$$
\begin{array}{l}
\cong\langle\ \mathsf{ext^{-1}}\ (\mathsf{ext^{-1}}\ \mathsf{comm}\beta\ \mathsf{j})\ (\mathsf{s},\_)\ \rangle \\
\quad \alpha\ \mathsf{j}\ (\mathsf{s}, \lambda\ \mathsf{j}\ \mathsf{p}\ \rightarrow\ \beta\ \mathsf{j}\ (\mathsf{f}\ (\mathsf{j},\mathsf{p}) \\
\qquad\qquad\qquad\qquad , \mathsf{ext^{-1}}\ (\mathsf{cong}\ (\pi_1 \circ \mathsf{sup^{-1}})\ \mathsf{ok})\ (\mathsf{j},\mathsf{p}))) \\
\cong\langle\ (\mathsf{cong}\ (\lambda\ \mathsf{n}\ \rightarrow\ \alpha\ \mathsf{j}\ (\mathsf{s},\mathsf{n})) \\
\qquad\quad (\lambda^{\equiv}\ \mathsf{j}\ \rightarrow\ \lambda^{\equiv}\ \mathsf{p}\ \rightarrow \\
\qquad\qquad \mathsf{WIfoldUniq'}\ \alpha\ \beta\ \mathsf{comm}\beta\ \mathsf{j} \\
\qquad\qquad\quad (\mathsf{f}\ (\mathsf{j},\mathsf{p}), \mathsf{ext^{-1}}\ (\mathsf{cong}\ (\pi_1 \circ \mathsf{sup^{-1}})\ \mathsf{ok})\ (\mathsf{j},\mathsf{p})))\ \rangle \\
\quad \alpha\ \mathsf{j}\ (\mathsf{s}, \lambda\ \mathsf{j}\ \mathsf{p}\ \rightarrow\ \mathsf{WIfold}\ \alpha\ \mathsf{j}\ (\mathsf{f}\ (\mathsf{j},\mathsf{p}) \\
\qquad\qquad\qquad\qquad , \mathsf{ext^{-1}}\ (\mathsf{cong}\ (\pi_1 \circ \mathsf{sup^{-1}})\ \mathsf{ok})\ (\mathsf{j},\mathsf{p})))
\end{array}
$$

■

□

We can use this proof that WI-types can be encoded by W to explain where Path fits in, since it is straightforwardly encoded as a WI:

$$
\begin{array}{l}
\mathsf{Path} : \{\mathsf{I}\ \mathsf{J} : \mathsf{Set}\}\ (\mathsf{S} : \mathsf{J} \rightarrow \mathsf{Set}) \\
\qquad (\mathsf{P^I} : (\mathsf{j} : \mathsf{J}) \rightarrow \mathsf{S}\ \mathsf{j} \rightarrow \mathsf{I} \rightarrow \mathsf{Set}) \\
\qquad (\mathsf{P^J} : (\mathsf{j} : \mathsf{J}) \rightarrow \mathsf{S}\ \mathsf{j} \rightarrow \mathsf{J} \rightarrow \mathsf{Set}) \\
\qquad (\mathsf{j} : \mathsf{J}) \rightarrow \mathsf{WI}\ \mathsf{S}\ \mathsf{P^J}\ \mathsf{j} \rightarrow \mathsf{I} \rightarrow \mathsf{Set} \\
\mathsf{Path}\ \{\mathsf{I}\}\ \{\mathsf{J}\}\ \mathsf{S}\ \mathsf{P^I}\ \mathsf{P^J}\ \mathsf{j}\ \mathsf{w}\ \mathsf{i} = \mathsf{WI}\ \mathsf{PathS}\ \mathsf{PathP}\ (\mathsf{j},\mathsf{w}) \\
\quad \textbf{where}\ \mathsf{PathS} : (\mathsf{j} : \mathsf{J}) \times \mathsf{WI}\ \mathsf{S}\ \mathsf{P^J}\ \mathsf{j} \rightarrow \mathsf{Set} \\
\qquad\quad \mathsf{PathS}\ (\mathsf{j}, \mathsf{sup}\ (\mathsf{s},\mathsf{f})) = \mathsf{P^I}\ \mathsf{j}\ \mathsf{s}\ \mathsf{i} \uplus \Sigma\ \mathsf{J}\ (\mathsf{P^J}\ \mathsf{j}\ \mathsf{s}) \\
\qquad\quad \mathsf{PathP} : (\mathsf{jw} : (\mathsf{j} : \mathsf{J}) \times \mathsf{WI}\ \mathsf{S}\ \mathsf{P^J}\ \mathsf{j})\ (\mathsf{s} : \mathsf{PathS}\ \mathsf{jw}) \rightarrow \\
\qquad\qquad\quad (\mathsf{j} : \mathsf{J}) \times \mathsf{WI}\ \mathsf{S}\ \mathsf{P^J}\ \mathsf{j} \rightarrow \mathsf{Set} \\
\qquad\quad \mathsf{PathP}\ (\mathsf{j}, \mathsf{sup}\ (\mathsf{s},\mathsf{f}))\ (\mathsf{inl}\ \mathsf{p})\ (\mathsf{j'},\mathsf{w'}) = \bot \\
\qquad\quad \mathsf{PathP}\ (\mathsf{j}, \mathsf{sup}\ (\mathsf{s},\mathsf{f}))\ (\mathsf{inr}\ (\mathsf{j''},\mathsf{p}))\ (\mathsf{j'},\mathsf{w'}) = \\
\qquad\qquad (\mathsf{j''} \equiv \mathsf{j'}) \times (\mathsf{f}\ \mathsf{j''}\ \mathsf{p} \cong \mathsf{w'})
\end{array}
$$

The reader will be unsurprised to learn that a similar construction to the above allows us to derive MI-types from M-types. The details are, once again, somewhat obfuscated by the experimental treatment of co-induction in Agda, but are in the spirit of the dual of the proof above.

7.2 M from W

Since we have shown that both WI and MI types can be reduced to their non-indexed counterparts, we can finish the reduction of the logical theory of indexed containers to W-types by showing that M types can be reduced to W-types. This is a result from our previous work on containers (Abbott *et al.* 2005), though in the setting of indexed WI-types, we can give a better explanation. Before tackling this question directly, we first introduce the basic definitions pertaining to final coalgebras and our implementation of them within Agda.

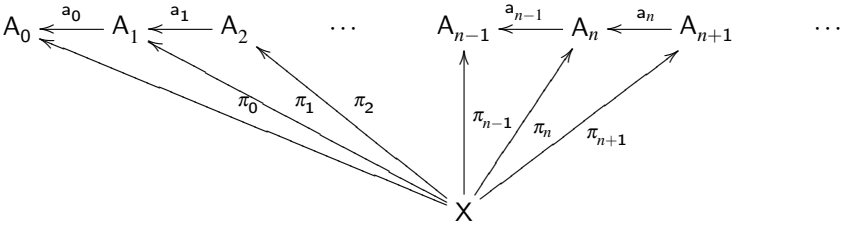
In category theory, an ω-chain is an infinite diagram:

$$
\mathsf{A}_0 \xleftarrow{\mathsf{a}_0} \mathsf{A}_1 \xleftarrow{\mathsf{a}_1} \mathsf{A}_2 \qquad \cdots \qquad \mathsf{A}_{n-1} \xleftarrow{\mathsf{a}_{n-1}} \mathsf{A}_n \xleftarrow{\mathsf{a}_n} \mathsf{A}_{n+1} \qquad \cdots
$$

In type theory, we can represent such a chain as a pair of functions:

$$\begin{aligned}&\mathsf{Chain} : \mathsf{Set}_1\\&\mathsf{Chain} = (\mathsf{A} : (\mathbb{N} \to \mathsf{Set})) \times ((\mathsf{n} : \mathbb{N}) \to \mathsf{A}\,(\mathsf{suc}\,\mathsf{n}) \to \mathsf{A}\,\mathsf{n})\end{aligned}$$

A cone for a chain is an object X and family of projections $\pi_n \in \mathsf{X} \to \mathsf{A}_n$ such that, in the following diagram, all the small triangles commute:



The limit of a chain is the cone which is terminal amongst all cones for that chain. This terminality condition is called the universal property of the limit. We can encode the limit of a chain, including its projections and its universal property as follows:

$$\begin{aligned}&\mathsf{LIM} : \mathsf{Chain} \to \mathsf{Set}\\&\mathsf{LIM}\,(\mathsf{A},\mathsf{a}) = (\mathsf{f} : ((\mathsf{n} : \mathbb{N}) \to \mathsf{A}\,\mathsf{n})) \times\\&\qquad\qquad ((\mathsf{n} : \mathbb{N}) \to \mathsf{a}\,\mathsf{n}\,(\mathsf{f}\,(\mathsf{suc}\,\mathsf{n})) \equiv \mathsf{f}\,\mathsf{n})\\&\pi : \{\mathsf{c} : \mathsf{Chain}\} \to (\mathsf{n} : \mathbb{N}) \to \mathsf{LIM}\,\mathsf{c} \to \pi_0\,\mathsf{c}\,\mathsf{n}\\&\pi\,\mathsf{n}\,(\mathsf{f},\mathsf{p}) = \mathsf{f}\,\mathsf{n}\\&\mathsf{comm} : \{\mathsf{c} : \mathsf{Chain}\}\,(\mathsf{n} : \mathbb{N})\,(\mathsf{l} : \mathsf{LIM}\,\mathsf{c}) \to\\&\qquad \pi_1\,\mathsf{c}\,\mathsf{n}\,(\pi\,\{\mathsf{c}\}\,(\mathsf{suc}\,\mathsf{n})\,\mathsf{l}) \equiv \pi\,\{\mathsf{c}\}\,\mathsf{n}\,\mathsf{l}\\&\mathsf{comm}\,\mathsf{n}\,(\mathsf{f},\mathsf{p}) = \mathsf{p}\,\mathsf{n}\\&\mathsf{univ} : \quad \{\mathsf{c} : \mathsf{Chain}\}\,\{\mathsf{X} : \mathsf{Set}\}\,(\mathsf{pro} : (\mathsf{n} : \mathbb{N}) \to \mathsf{X} \to \pi_0\,\mathsf{c}\,\mathsf{n})\\&\qquad (\mathsf{com} : \ (\mathsf{n} : \mathbb{N})\,(\mathsf{x} : \mathsf{X}) \to\\&\qquad\qquad \pi_1\,\mathsf{c}\,\mathsf{n}\,(\mathsf{pro}\,(\mathsf{suc}\,\mathsf{n})\,\mathsf{x}) \equiv \mathsf{pro}\,\mathsf{n}\,\mathsf{x}) \to\\&\qquad \mathsf{X} \to \mathsf{LIM}\,\mathsf{c}\\&\mathsf{univ}\,\mathsf{pro}\,\mathsf{com}\,\mathsf{x} = (\lambda\ \mathsf{n} \to \mathsf{pro}\,\mathsf{n}\,\mathsf{x}), (\lambda\ \mathsf{n} \to \mathsf{com}\,\mathsf{n}\,\mathsf{x})\end{aligned}$$

We are interested in certain ω-chains which can be constructed from a functor F as follows (where ! is the unique morphism from any object into the terminal object $\top$):

$$\top \xleftarrow{!} \mathsf{F}\top \xleftarrow{\mathsf{F}!} \mathsf{F}^2\top \xleftarrow{\mathsf{F}^2!} \mathsf{F}^3\top \qquad \cdots$$

For the moment, denote this chain $\mathsf{F}^\omega = ((\lambda\ \mathsf{n} \to \mathsf{F}^\mathsf{n}\,\top), \lambda\ \mathsf{n} \to \mathsf{F}^\mathsf{n}\,!)$. We know from Asperti & Longo (1991) that if F is ω-continuous, *i.e.* that for any chain (A,a):

$$\mathsf{F}\,(\mathsf{LIM}\,(\mathsf{A},\mathsf{a})) \approx \mathsf{LIM}\,((\mathsf{F}\circ\mathsf{A}),(\mathsf{F}\circ\mathsf{a}))$$

then the limit of F^ω will be the terminal co-algebra of F. To see this, we first observe that there is an isomorphism between the limit of a chain, and the limit of any of its *tails*:

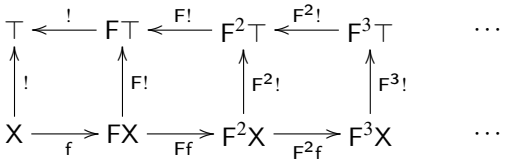
$$\begin{aligned}&\mathsf{tail} : \mathsf{Chain} \to \mathsf{Chain}\\&\mathsf{tail}\,(\mathsf{A},\mathsf{a}) = (\mathsf{A}\circ\mathsf{suc}, \mathsf{a}\circ\mathsf{suc})\\&\mathsf{tailLIM} : (\mathsf{c} : \mathsf{Chain}) \to \mathsf{LIM}\,\mathsf{c} \to \mathsf{LIM}\,(\mathsf{tail}\,\mathsf{c})\\&\mathsf{tailLIM}\,(\mathsf{A},\mathsf{a})\,(\mathsf{f},\mathsf{p}) = \mathsf{f}\circ\mathsf{suc}, \mathsf{p}\circ\mathsf{suc}\end{aligned}$$

```
tailLIM⁻¹ : (c : Chain) → LIM (tail c) → LIM c
tailLIM⁻¹ (A, a) (f, p) = f′, p′
  where f′ : (n : ℕ) → A n
        f′ zero = a _ (f zero)
        f′ (suc n) = f n
        p′ : (n : ℕ) → a _ (f n) ≅ f′ n
        p′ zero = refl
        p′ (suc n) = p n
```

We also note that the tail of F^ω is $((\lambda\ \mathsf{n} \to \mathsf{F}\ (\mathsf{F}^{\mathsf{n}}\ \top)), \lambda\ \mathsf{n} \to \mathsf{F}\ (\mathsf{F}^{\mathsf{n}}\ !))$, which allows us to construct the isomorphism between $\mathsf{F}\ (\mathsf{LIM}\ \mathsf{F}^\omega)$ and $\mathsf{LIM}\ \mathsf{F}^\omega$:

$$
\begin{array}{lll}
 & \mathsf{F}\ (\mathsf{LIM}\ \mathsf{F}^\omega) & \\
\approx & \mathsf{LIM}\ (\mathsf{F} \circ (\lambda\ \mathsf{n} \to \mathsf{F}^{\mathsf{n}}\ \top), \mathsf{F} \circ (\lambda\ \mathsf{n} \to \mathsf{F}^{\mathsf{n}}\ !)) & \{\mathsf{F} \text{ is } \omega\text{-continuous}\} \\
\equiv & \mathsf{LIM}\ ((\lambda\ \mathsf{n} \to \mathsf{F}\ (\mathsf{F}^{\mathsf{n}}\ \top)), (\lambda\ \mathsf{n} \to \mathsf{F}\ (\mathsf{F}^{\mathsf{n}}\ !))) & \{\text{definition}\} \\
\approx & \mathsf{LIM}\ \mathsf{F}^\omega & \{\ \mathsf{tailLIM}\ \}
\end{array}
$$

This isomorphism is witnessed from right to left by the co-algebra map out. To show that the co-algebra is terminal, we employ the universal property of LIM. Given a co-algebra for $\alpha : \mathsf{X} \to \mathsf{F}\ \mathsf{X}$, we construct an F^ω cone:



We now turn to the specific task at hand, namely the construction of M-types from W-types, that is the capacity to construct final coalgebras of container functors from the capacity to construct the initial algebras of container functors. In order to do this, we must construct the iteration of container functors (to build the chain) and show that all container functors are ω-continuous. Since we only need to build iterations of container functors applied to the terminal object $\top$, we build that directly. We define the following variation of W, cut-off at a known depth:

```
data WM (S : Set) (P : S → Set) : ℕ → Set where
  wm⊤ : WM S P zero
  sup : ∀ {n} → ⟦ S ◁ P ⟧ (WM S P n) → WM S P (suc n)
```

Note that WM is itself encodable as an indexed WI-type (and, by the final result in Section 7.1, a W-type):

```
WM′ : (S : Set) (P : S → Set) → ℕ → Set
WM′ S P = WI S′ P′
  where
    S′ : ℕ → Set
```

```
S′ zero = ⊤
S′ (suc n) = S
P′ : (n : ℕ) → S′ n → ℕ → Set
P′ zero _ _ = ⊥
P′ (suc m) s n with m ≟ n
P′ (suc .n) s n | yes refl = P s
P′ (suc m) s n | no ¬p = ⊥
```

Our candidate for the final coalgebra of $[\![\mathsf{S} \lhd \mathsf{P}]\!]$ is, then, the limit of the chain WM S P, along with the truncation of a tree of depth suc n to one of depth n. This truncation is achieved by the repeated application of the morphism part of the container functor to the unique morphism into the terminal object. Or, more concretely:

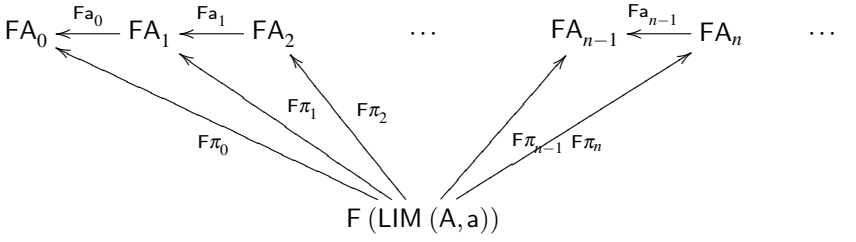
```
trunc : ∀ {S P} → (n : ℕ) → WM S P (suc n) → WM S P n
trunc zero (sup (s, f)) = wm⊤
trunc (suc n) (sup (s, f)) = sup (s, trunc n ∘ f)
```

Now, we can build the chain of finite iterations of a container functor whose limit will form the final coalgebra of the container functor.

```
M-chain : (S : Set) (P : S → Set) → Chain
M-chain S P = WM S P , trunc
```

Proposition 7.2
All container functors are ω-continuous. That is, they preserve ω-limits.

Proof
We want to build the isomorphism $\mathsf{F}\,(\mathsf{LIM}\,(\mathsf{A},\mathsf{a})) \cong \mathsf{LIM}\,((\mathsf{F}\circ\mathsf{A}),\mathsf{F}\circ\mathsf{a})$ in the case that F is a container functor. However, the function from left to right is uniquely given by the universal property of LIM for all functors $\mathsf{F} : \mathsf{Set} \to \mathsf{Set}$. To show this, we build the cone for the chain $((\mathsf{F}\circ\mathsf{A}),\mathsf{F}\circ\mathsf{a})$:



The small triangles in the diagram above obviously commute, so there exists a unique morphism from $\mathsf{F}\,(\mathsf{LIM}\,(\mathsf{A},\mathsf{a}))$ into $\mathsf{LIM}\,((\mathsf{F}\circ\mathsf{A}),\mathsf{F}\circ\mathsf{a})$. All that remains then, is to construct an inverse to this unique morphism, in the case that $\mathsf{F} \equiv [\![\mathsf{S} \lhd \mathsf{P}]\!]$, that is we must build a function:

```
ω-cont : LIM ((λ n → (s : S) × (P s → A n))
             , λ n → λ {(s, f) → (s, a n ∘ f)})
  → (s : S) × (P s → (LIM (A, a)))
```

Note that the shape picked at every point along the chain that we are given must be the same, in order to make the diagrams commute. This is the key insight into constructing this function:

```
ω-cont (f, p) =
  (π₀ (f zero), λ x →
    (λ n → π₁ (f n) (subst P (f₀≡ n) x))
    , λ n → begin
        a n (π₁ (f (suc n)) (subst P (f₀≡ (suc n)) x))
      ≅⟨ exteq⁻¹ (cong (P ∘ π₀) (p n)) (λ≅ _ → refl)
           (cong π₁ (p n))
           (begin
              subst P (f₀≡ (suc n)) x
            ≅⟨ subst-removable P (f₀≡ (suc n)) x ⟩
              x
            ≅⟨ sym (subst-removable P (f₀≡ n) x) ⟩
              subst P (f₀≡ n) x ■) ⟩
        π₁ (f n) (subst P (f₀≡ n) x) ■)
  where f₀≡ : (n : ℕ) → (π₀ (f 0)) ≡ (π₀ (f n))
        f₀≡ zero = refl
        f₀≡ (suc n) = trans (f₀≡ n) (sym (cong π₀ (p n)))
```

□

Now, since we have established that M-chain is isomorphic to the chain of iterations of container functors, and that all container functors are ω-continuous, we know that the terminal co-algebra of a container functor must be the limit of its M-chain:

```
M : (S : Set) (P : S → Set) → Set
M S P = LIM (M-chain S P)
```

In this section, we have established that we can derive WI-types from W (and by duality we argue MI types from M) and also M types from W, by these results we can reduce all the constructions in this paper to the setting of extensional type theory with W-types, or equivalently, *any* Martin-Löf category. That is to say, in the move from containers to indexed containers, we require no extra structure in our underlying type theory.

8 Strictly positive families

We have developed indexed containers as representations of those indexed functors which, intuitively, support shapes and positions metaphor. These shapes and positions are just as with standard containers apart from the fact they are indexed. We now introduce a grammar for strict positivity suitable for generating inductive families, and show that all such functors can be encoded as indexed container functors. This grammar defines what we call the strictly positive families. Strictly positive families are in turn defined from indexed strictly positive types as follows:

```
mutual
  SPF : (I J : Set) → Set₁
  SPF I J = J → ISPT I

  data ISPT (I : Set) : Set₁ where
    η⊤ :          (i : I)                    → ISPT I
    Δ⊤ : ∀ {J K} (f : J → K) (F : SPF I K)   → SPF I J
    Σ⊤ : ∀ {J K} (f : J → K) (F : SPF I J)   → SPF I K
    Π⊤ : ∀ {J K} (f : J → K) (F : SPF I J)   → SPF I K
    μ⊤ : ∀ {J}   (F : SPF (I ⊎ J) J)         → SPF I J
    ν⊤ : ∀ {J}   (F : SPF (I ⊎ J) J)         → SPF I J
```

We show how to interpret strictly positive families as indexed containers and hence indexed functors.

```
mutual
  ⟦_⟧⊤⋆ : ∀ {I J} → SPF I J → ICont⋆ I J
  ⟦ F ⟧⊤⋆ = λ j → ⟦ F j ⟧⊤

  ⟦_⟧⊤ : ∀ {I} → ISPT I → ICont I
  ⟦ η⊤ i ⟧⊤     = ηᶜ i
  ⟦ Δ⊤ f F j ⟧⊤  = Δᶜ f ⟦ F ⟧⊤⋆ j
  ⟦ Σ⊤ f F k ⟧⊤  = Σᶜ f ⟦ F ⟧⊤⋆ k
  ⟦ Π⊤ f F k ⟧⊤  = Πᶜ f ⟦ F ⟧⊤⋆ k
  ⟦ μ⊤ F j ⟧⊤   = μᶜ ⟦ F ⟧⊤⋆ j
  ⟦ ν⊤ F j ⟧⊤   = νᶜ ⟦ F ⟧⊤⋆ j
```

Just as indexed containers support a relative monad structure, so do strictly positive families:

```
mutual
  ISPT : ∀ {I J} → (I → J) → ISPT I → ISPT J
  ISPT γ t = t >>=⊤ (η⊤ ∘ γ)

  SPF : ∀ {I J K} → (I → J) → SPF I K → SPF J K
  SPF γ t k = ISPT γ (t k)

  _>>=⊤_ : ∀ {I J} → ISPT I → SPF J I → ISPT J
  η⊤ i     >>=⊤ F = F i
  Δ⊤ f G j  >>=⊤ F = Δ⊤ f (λ k → G k >>=⊤ F) j
  Σ⊤ f G k  >>=⊤ F = Σ⊤ f (λ j → G j >>=⊤ F) k
  Π⊤ f G k  >>=⊤ F = Π⊤ f (λ j → G j >>=⊤ F) k
  μ⊤ G j    >>=⊤ F = μ⊤ (λ k → G k >>=⊤ [ (SPF inl F) , (η⊤ ∘ inr) ]) j
  ν⊤ G j    >>=⊤ F = ν⊤ (λ k → G k >>=⊤ [ (SPF inl F) , (η⊤ ∘ inr) ]) j
```

As defined above this doesn't pass Agda's termination check, due to deriving the ISPT from the monad instance. If we define the map of the functor directly the whole thing obviously terminates, at the expense of having to show that the two definitions of the map for ISPT agree.

Proposition 8.1
$(\mathsf{ISPT}, \eta^{\mathsf{T}}, \_ \gg\!=^{\mathsf{T}} \_)$ is a *relative monad* on the lifting functor $\uparrow : \mathsf{Set} \to \mathsf{Set}_1$. Moreover, this structure is preserved under the translation to containers $[\![\_]\!]^{\mathsf{T}}$.

Proof
To prove the structure is a relative monad, we observe that the following equalities hold:
For $\mathsf{F} : \mathsf{ISPT}\ \mathsf{K}, \mathsf{G} : \mathsf{SPF}\ \mathsf{J}\ \mathsf{K}, \mathsf{H} : \mathsf{ISPT}\ \mathsf{I}\ \mathsf{J}$:

$$\mathsf{H}\,\mathsf{j} \equiv (\eta^{\mathsf{T}}\,\mathsf{j}) \gg\!=^{\mathsf{T}} \mathsf{H} \tag{4}$$

$$\mathsf{F} \equiv \mathsf{F} \gg\!=^{\mathsf{T}} \eta^{\mathsf{F}} \tag{5}$$

$$(\mathsf{F} \gg\!=^{\mathsf{T}} \mathsf{G}) \gg\!=^{\mathsf{T}} \mathsf{F} \equiv \mathsf{F} \gg\!=^{\mathsf{T}} (\lambda\ \mathsf{k} \to (\mathsf{G}\,\mathsf{k}) \gg\!=^{\mathsf{T}} \mathsf{H}) \tag{6}$$

The first is by definition, and the others follow by induction on F. To show that the structure is preserved by $[\![\_]\!]^{\mathsf{T}}$ it is sufficient to show that for all $\mathsf{F} : \mathsf{ISPT}\ \mathsf{J}$ and $\mathsf{G} : \mathsf{SPF}\ \mathsf{I}\ \mathsf{J}$ there exist mutually inverse container morphisms bindpres and $\mathsf{bindpres}^{-1}$:

$$\begin{aligned}
&\mathsf{bindpres} &&: ([\![\ \mathsf{F} \gg\!=^{\mathsf{T}} \mathsf{G}\ ]\!]^{\mathsf{T}}) &&\Rightarrow^{\mathsf{c}} ([\![\ \mathsf{F}\ ]\!]^{\mathsf{T}} \gg\!=^{\mathsf{c}} [\![\ \mathsf{G}\ ]\!]^{\mathsf{T}\star})\\
&\mathsf{bindpres}^{-1} &&: ([\![\ \mathsf{F}\ ]\!]^{\mathsf{T}} \gg\!=^{\mathsf{c}} [\![\ \mathsf{G}\ ]\!]^{\mathsf{T}\star}) &&\Rightarrow^{\mathsf{c}} ([\![\ \mathsf{F} \gg\!=^{\mathsf{T}} \mathsf{G}\ ]\!]^{\mathsf{T}})
\end{aligned}$$

□

We finish by showing how strictly positive families represent some of the key indexed datatypes we saw in the beginning of the paper. We start by showing that, as with indexed containers and indexed functors, strictly positive families support disjoint unions and cartesian products.

$$\begin{aligned}
&\bot^{\mathsf{T}} : \forall\,\{\mathsf{I}\} \to \mathsf{ISPT}\ \mathsf{I}\\
&\bot^{\mathsf{T}} = \Sigma^{\mathsf{T}}\,\{\mathsf{J} = \bot\}\,\{\mathsf{K} = \top\}\,_\,(\lambda\ ())\ _\\
&_ \uplus^{\mathsf{T}} _ : \forall\,\{\mathsf{I}\} \to (\mathsf{F}\ \mathsf{G} : \mathsf{ISPT}\ \mathsf{I}) \to \mathsf{ISPT}\ \mathsf{I}\\
&\mathsf{F} \uplus^{\mathsf{T}} \mathsf{G} = \Sigma^{\mathsf{T}}\,\{\mathsf{K} = \top\}\,_\,(\lambda\ \mathsf{b} \to \mathsf{if\ b\ then\ F\ else\ G})\ _\\
&\top^{\mathsf{T}} : \forall\,\{\mathsf{I}\} \to \mathsf{ISPT}\ \mathsf{I}\\
&\top^{\mathsf{T}} = \Pi^{\mathsf{T}}\,\{\mathsf{J} = \bot\}\,\{\mathsf{K} = \top\}\,_\,(\lambda\ ())\ _\\
&_ \times^{\mathsf{T}} _ : \forall\,\{\mathsf{I}\} \to (\mathsf{F}\ \mathsf{G} : \mathsf{ISPT}\ \mathsf{I}) \to \mathsf{ISPT}\ \mathsf{I}\\
&\mathsf{F} \times^{\mathsf{T}} \mathsf{G} = \Pi^{\mathsf{T}}\,\{\mathsf{K} = \top\}\,_\,(\lambda\ \mathsf{b} \to \mathsf{if\ b\ then\ F\ else\ G})\ _
\end{aligned}$$

We can now define finite sets, vectors and lambda terms as strictly positive families.

$$\begin{aligned}
&\mathsf{T}_{\mathsf{Fin}} : \mathsf{SPF}\ \bot\ \mathbb{N}\\
&\mathsf{T}_{\mathsf{Fin}} = \mu^{\mathsf{T}}\,(\Sigma^{\mathsf{T}}\,\mathsf{suc}\,(\top^{\mathsf{T}} \uplus^{\mathsf{T}} (\eta^{\mathsf{T}} \circ \mathsf{inr})))\\
&\mathsf{T}_{\mathsf{Vec}} : \mathsf{SPF}\ \top\ \mathbb{N}\\
&\mathsf{T}_{\mathsf{Vec}} = \mu^{\mathsf{T}}\ \ (\ \ \Sigma^{\mathsf{T}}\,\{\mathsf{J} = \top\}\,(\lambda\ _ \to \mathsf{zero})\,(\lambda\ _ \to \top^{\mathsf{T}})\\
&\qquad\qquad \uplus^{\mathsf{T}}\ \Sigma^{\mathsf{T}}\,\mathsf{suc}\,(\lambda\ \mathsf{n} \to \eta^{\mathsf{T}}\,(\mathsf{inl}\ _) \times^{\mathsf{T}} \eta^{\mathsf{T}}\,(\mathsf{inr\ n})))\\
&\mathsf{T}_{\mathsf{ScLam}} : \mathsf{SPF}\ \bot\ \mathbb{N}\\
&\mathsf{T}_{\mathsf{ScLam}} = \mu^{\mathsf{T}}\,(\ \ \mathsf{SPF}\,(\lambda\ ())\,\mathsf{T}_{\mathsf{Fin}}\\
&\qquad\qquad \uplus^{\mathsf{T}}\ (((\eta^{\mathsf{T}} \circ \mathsf{inr}) \times^{\mathsf{T}} (\eta^{\mathsf{T}} \circ \mathsf{inr}))\\
&\qquad\qquad \uplus^{\mathsf{T}}\ \Delta^{\mathsf{T}}\,\mathsf{suc}\,(\eta^{\mathsf{T}} \circ \mathsf{inr})))
\end{aligned}$$

Note that we have to weaken the reference to $\mathsf{T}_{\mathsf{Fin}}$ in the definition of $\mathsf{T}_{\mathsf{ScLam}}$, since under the μ^{T} we can refer to the recursive TSCLam trees, but $\mathsf{T}_{\mathsf{Fin}}$ itself can refer to no

variables. We can also define the mutual types Ne and Nf. Here, a copy of the normal forms is defined *inside* the definition of the neutral terms, and vice versa:

$$
\begin{array}{l}
\mathsf{T}_{\mathsf{NeLam}} : \mathsf{SPF}\ \bot\ \mathbb{N} \\
\mathsf{T}_{\mathsf{NeLam}} = \mu^{\mathsf{T}}\ (\mathsf{SPF}\ (\lambda\ ())\ \mathsf{T}_{\mathsf{Fin}} \\
\quad \uplus^{\mathsf{T}} \qquad ((\eta^{\mathsf{T}} \circ \mathsf{inr})\ \times^{\mathsf{T}}\ \mathsf{T}_{\mathsf{NeNf}})) \\
\quad \textbf{where}\ \mathsf{T}_{\mathsf{NeNf}} : \mathsf{SPF}\ (\bot \uplus \mathbb{N})\ \mathbb{N} \\
\qquad\quad \mathsf{T}_{\mathsf{NeNf}} = \mu^{\mathsf{T}}\ (\quad (\Delta^{\mathsf{T}}\ \mathsf{suc}\ (\eta^{\mathsf{T}} \circ \mathsf{inr})) \\
\qquad\qquad\qquad \uplus^{\mathsf{T}}\ (\eta^{\mathsf{T}} \circ (\mathsf{inl} \circ \mathsf{inr}))) \\
\mathsf{T}_{\mathsf{NfLam}} : \mathsf{SPF}\ \bot\ \mathbb{N} \\
\mathsf{T}_{\mathsf{NfLam}} = \mu^{\mathsf{T}}\ (\quad \Delta^{\mathsf{T}}\ \mathsf{suc}\ (\eta^{\mathsf{T}} \circ \mathsf{inr}) \\
\qquad\quad \uplus^{\mathsf{T}}\ \mathsf{T}_{\mathsf{NfNe}}) \\
\quad \textbf{where}\ \mathsf{T}_{\mathsf{NfNe}} : \mathsf{SPF}\ (\bot \uplus \mathbb{N})\ \mathbb{N} \\
\qquad\quad \mathsf{T}_{\mathsf{NfNe}} = \mu^{\mathsf{T}}\ (\quad \mathsf{SPF}\ (\lambda\ ())\ \mathsf{T}_{\mathsf{Fin}} \\
\qquad\qquad\qquad \uplus^{\mathsf{T}}\ ((\eta^{\mathsf{T}} \circ \mathsf{inr})\ \times^{\mathsf{T}}\ (\eta^{\mathsf{T}} \circ (\mathsf{inl} \circ \mathsf{inr}))))
\end{array}
$$

From these definitions, we can derive the actual datatypes with constructors and eliminators by unfolding all definitions. Example, in the case of $\mathsf{T}_{\mathsf{Fin}}$ we derive the container

$$
\begin{array}{l}
\mathsf{TFinC} : \mathsf{ICont}^{\star}\ \bot\ \mathbb{N} \\
\mathsf{TFinC} = [\![\ \mathsf{T}_{\mathsf{Fin}}\ ]\!]^{\mathsf{T}\star}
\end{array}
$$

the next step is to construct the associated indexed functor:

$$
\begin{array}{l}
\mathsf{TFinF} : \mathsf{IFunc}^{\star}\ \bot\ \mathbb{N} \\
\mathsf{TFinF} = [\![\ \mathsf{TFinC}\ ]\!]^{\star}
\end{array}
$$

and finally the actual datatype

$$
\begin{array}{l}
\mathsf{Fin} : \mathbb{N} \to \mathsf{Set} \\
\mathsf{Fin}\ \mathsf{n} = (\mathsf{TFinF}\ \mathsf{n})\ (\lambda\ ())
\end{array}
$$

We leave the laborious derivation of the constructors and the eliminator to the reader.

9 Conclusions

We have shown how inductive and co-inductive families, a central feature in dependently typed programming, can be constructed from the standard infrastructure present in type theory, i.e. W-types together with Π, Σ and equality types. Indeed, we are able to reduce the syntactically rich notion of families to a small collection of categorically inspired combinators. This is an alternative to the syntactic schemes to define inductive families present in the *Calculus of Inductive Constructions* (CIC), or in the Agda and Epigram systems. Indeed, indexed containers can also be viewed as normal forms of Dybjer–Setzer codes (Dybjer & Setzer 2006) for non-recursive indexed inductive definitions. We are able to encode inductively defined families in a small core language which means that we rely only on a small trusted code base. The reduction to W-types requires an extensional propositional equality. Our current approach using an axiom ext is sufficient for proofs but isn't computationally adequate. A more satisfying approach would built on *Observational Type Theory* (OTT) (Altenkirch *et al.* 2007).

The present paper is an annotated Agda script, i.e. all the proofs are checked by the Agda system. We have tried hard to integrate the formal development with the narrative. In some cases, we have suppressed certain details present in the source of the paper to keep the material readable.

A more serious challenge are mutual inductively (or coinductively) defined families where one type depends on another (Nordvall Forsberg & Setzer 2010; Nordvall Forsberg 2013). A typical example is the syntax of type theory itself which, to simplify, can be encoded by mutually defining contexts containing terms, types in a given context and terms in a given type:

$$\begin{aligned}&\mathsf{Con} : \mathsf{Set}\\&\mathsf{Ty} : \mathsf{Con} \to \mathsf{Set}\\&\mathsf{Tm} : (\Gamma : \mathsf{Con}) \to \mathsf{Ty}\,\Gamma \to \mathsf{Set}\end{aligned}$$

In recent work (Altenkirch *et al.* 2011), present a categorical semantics for this kind of definitions based on dialgebras. However, a presentation of strictly positive definitions in the spirit of containers is not yet available.

Acknowledgment

We would like to thank the anonymous referees for useful comments and suggestions and Frederik Forsberg for help with proofreading.

References

Abbott, M., Altenkirch, T. & Ghani, N. (2003) Categories of containers. In Proceedings of Foundations of Software Science and Computation Structures. Springer, LNCS 2620, pp. 23–38.

Abbott, M., Altenkirch, T. & Ghani, N. (2005) Containers - constructing strictly positive types. *Theor. Comput. Sci.* **342**, 3–27.

Adámek, J. & Koubek, V. (1995) On the greatest fixed point of a set functor. *Theor. Comput. Sci.* **150**(1), 57–75.

Altenkirch, T. & Chapman, J. (2009) Big-step normalisation. *J. Funct. Program.* **19**(3–4), 311–333.

Altenkirch, T., Chapman, J., & Uustalu, T. (2010) Monads need not be endofunctors. *Foundations of Software Science and Computational Structures*. Springer, LNCS 6014, pp. 297–311.

Altenkirch, T., Ghani, N., McBride, C. & Morris, P. (2015) Agda sources for indexed containers. Available at: http://cs.not.ac.uk/~ txa/ic-code/.

Altenkirch, T., Levy, P. & Staton, S. (2010c) Higher order containers. Computability in Europe. *Programs, Proofs, Processes.* Springer, LNCS 6158, pp. 11–20.

Altenkirch, T., McBride, C. & Swierstra, W. (2007) Observational equality, now! In Programming Languages meet Program Verification (PLPV2007) New York: ACM, pp. 57–68.

Altenkirch, T., Nordvall Forsberg, F., Morris, P. and Setzer, A. (2011) A categorical semantics for inductive-inductive definitions. In CALCO 2011: 4th International Conference on Algebra and Coalgebra in Computer Science. Springer, LNCS 6859, pp 70–84.

Altenkirch, T. & Reus, B. (1999) Monadic presentations of lambda terms using generalized inductive types. In *Computer Science Logic*. Springer, LNCS 1683, pp. 453–468.

Asperti, A. & Longo, G. (1991) *Categories, Types, and Structures*. MIT Press.

Bird, R. & de Moor, O. (1997) Algebra of programming. Prentice Hall.

Chapman, J., Dagand, P., McBride, C. & Morris, P. (2010) The gentle art of levitation. In *ACM Sigplan Notices*, vol. 45. ACM, pp. 3–14.

Cheney, J. & Hinze, R. (2003) First-class phantom types. Cornell University, Techn. Report.

Danielsson, N. A. & Altenkirch, T. (2010) Subtyping, declaratively; an exercise in mixed induction and coinduction. In Proceedings of the 9th International Conference on Mathematics of Program Construction (MPC 10). Springer, LNCS 6120, pp 100–118.

Dybjer, P. (1997) Representing inductively defined sets by wellorderings in Martin-Löf's type theory. *Theor. Comput. Sci.* **176**(1), 329–335.

Dybjer, P. & Setzer, A. (2001) Indexed induction-recursion. In *Proof Theory in Computer Science*, Springer, LNCS 2183, pp. 93–113.

Dybjer, P. & Setzer, A. (2006) Indexed induction–recursion. *J. Log. Algebr. Program.* **66**(1), 1–49.

Fiore, M., Gambino, N., Hyland, M. & Winskel, G. (2008) The cartesian closed bicategory of generalised species of structures. *J. London Math. Soc.* **77**(1), 203.

Gambino, N. & Hyland, M. (2004a) Wellfounded trees and dependent polynomial functors. In *Types for Proofs and Programs (TYPES 2003)*, Berardi, S., Coppo, M. & Damiani, F. (eds), Springer, LNCS 3085.

Gambino, N. & Hyland, M. (2004b) Wellfounded trees and dependent polynomial functors. In *Types for Proofs and Programs*. Springer, LNCS 3085, pp. 210–225.

Girard, J.-Y. (1988) Normal functors, power series and lambda-calculus. *Ann. Pure Appl. Log.* **37**(2), 129–177.

Hancock, P. & Hyvernat, P. (2006) Programming interfaces and basic topology. *Ann. Pure Appl. Log.* Vol. 137 (1–3), pp. 189–239.

Hancock, P., McBride, C., Ghani, N., Malatesta, L. & Altenkirch, T. (2013) Small induction recursion. In Typed Lambda Calculi and Applications, 11th International Conference, TLCA 2013, pp. 156–172.

Hofmann, M. (1996) Conservativity of equality reflection over intensional type theory. In *Types for Proofs and Programs*. Springer, LNCS 1158, pp. 153–164.

Huet, G. (1997) The zipper. *J. Funct. Program.* **7**(5), 549–554.

Joyal, A. (1987) Foncteurs analytiques et espèces de structures. In *Combinatoire énumerative*, Labelle, G. & Leroux, P. (eds), Springer, LNM 1234, pp. 126–159.

Kock, J. (2009) Notes on polynomial functors. Manuscript, available online.

Lindström, I. (1989) A construction of non-well-founded sets within martin-löf's type theory. *J. Symb. Log.* **54**(1), 57–64.

Mac Lane, S. (1998) *Categories for the Working Mathematician*, vol. 5. Springer.

McBride, C. (2001) The derivative of a regular type is its type of one-hole contexts. Available online.

McBride, C. (2010) Ornamental algebras, algebraic ornaments. Available online.

Morris, P. & Altenkirch, T. (2009) Indexed containers. In 24th IEEE Symposium in Logic in Computer Science (LICS 2009). IEEE, pp 277–285.

Morris, P., Altenkirch, T. & Ghani, N. (2007a) Constructing strictly positive families. In The Australasian Theory Symposium (CATS2007). ACS, pp 111–121.

Morris, P., Altenkirch, T. & Ghani, N. (2007b) A universe of strictly positive families. *International journal of foundations of computer science* **20**(1), 83–107, (2009).

Nordvall Forsberg, F. (2013) *Inductive-inductive definitions*. PhD thesis, Swansea University.

Nordvall Forsberg, F. and Setzer, A. (2010) Inductive-inductive definitions. In Proceedings of the 24th International Conference/19th Annual Conference on Computer Science Logic, pp. 454–468. Citeseer.

Petersson, K. & Synek, D. (1989) A set constructor for inductive sets in Martin-Löf's type theory. In Proceedings of the 1989 Conference on Category Theory and Computer Science, Manchester, UK: Springer Verlag, Lecture Notes in Computer Science, vol. 389.

Sozeau, M. (2007) Subset coercions in Coq. In *TYPES 2006,* Lecture Notes in Computer Science, vol. 4502. p. 237.

The Agda developers. (2015) The agda wiki. on the web.

The Agda Team. (2015) The agda wiki. Available at: `http://wiki.portal.chalmers.se/agda/agda.php`.

The Coq Development Team. (2008) *The Coq Proof Assistant Reference Manual – Version 8.1*.

Turner, D. A. (1985) Elementary strong functional programming. In 1st International Symposium on Functional Programming Languages in Education. Springer, LNCS 1022, pp. 1–13.