Indexed Containers

Thorsten Altenkirch Peter Morris
University of Nottingham
{txa,pwm}@cs.nott.ac.uk

Abstract

We show that the syntactically rich notion of inductive families can be reduced to a core type theory with a fixed number of type constructors exploiting the novel notion of indexed containers. Indexed containers generalize simple containers, capturing strictly positive families instead of just strictly positive types, without having to extend the core type theory. Other applications of indexed containers include datatype-generic programming and reasoning about polymorphic functions. The construction presented here has been formlized using the Agda system.

1 Introduction

Inductive datatypes are a central feature of modern Type Theory (e.g. COQ [23]) or functional programming (e.g. Haskell[1]) . A simple example is the type (or set) of de Bruijn λ-terms $\mathrm{Lam} \in \mathbf{Set}$, which can be specified as the inductive type generated by the following constructors, which using a 2-dimensional syntax inspired by the Epigram [17] syntax, can be written as follows:

$$\frac{i \in \mathbb{N}}{\mathrm{var}\, i \in \mathrm{Lam}} \qquad \frac{t, u \in \mathrm{Lam}}{\mathrm{app}\, t\, u \in \mathrm{Lam}} \qquad \frac{t \in \mathrm{Lam}}{\mathrm{lam}\, t \in \mathrm{Lam}}$$

Of course, we don't have to assume the natural numbers as primitive, but instead give an inductive definition a la Peano:

$$\frac{}{0 \in \mathbb{N}} \qquad \frac{n \in \mathbb{N}}{\mathrm{succ}\, n \in \mathbb{N}}$$

An elegant way to formalize and reason about inductive types is to model them as the initial algebra of an endofunctor, e.g. in the case of natural numbers $F_{\mathbb{N}}\, X = 1 + X$ and in the case of λ-terms $F_{\mathrm{Lam}}\, X = \mathbb{N} + X \times X + X$. 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* [7].

While the theory of inductive types is well developed, we often want to have a finer, more expressive, notion of types, for example to ensure the absence of run time errors — access to arrays out of range or access to an undefined variable in the previous example of λ-terms. To model this we move to the notion of an inductive family in Type Theory, where we can for example define the set of lambda terms $\mathrm{Lam}\, n$ with at most n free variables. Indeed we can view Lam as a function $\mathbb{N} \to \mathbf{Set}$ which assigns to every natural number n the type of λ terms with n free variables. Functions whose codomain is $\mathbf{Set}$ we call families and to define this family we first introduce the family of finite sets $\mathrm{Fin} \in \mathbb{N} \to \mathbf{Set}$ which assigns to n a type with exactly n elements:

$$\frac{n \in \mathbb{N}}{\mathrm{fzero} \in \mathrm{Fin}\,(n+1)} \qquad \frac{i \in \mathrm{Fin}\, n}{\mathrm{fsucc}\, i \in \mathrm{Fin}\,(n+1)}$$

which we use to encode the de Bruijn variables in the representation of λ-terms ($\mathrm{Lam} \in \mathbb{N} \to \mathbf{Set}$):

$$\frac{i \in \mathrm{Fin}\, n}{\mathrm{var}\, i \in \mathrm{Lam}\, n} \qquad \frac{t, u \in \mathrm{Lam}\, n}{\mathrm{app}\, t\, u \in \mathrm{Lam}\, n} \qquad \frac{t \in \mathrm{Lam}\,(n+1)}{\mathrm{lam}\, t \in \mathrm{Lam}\, n}$$

Importantly, the constructor lam reduces the number of *free* variables by one — by binding one. Inductive families may be mutually defined, an example being the β normal and neutral terms ($\mathrm{Nf}, \mathrm{Ne} \in \mathbb{N} \to \mathbf{Set}$)

$$\frac{i \in \mathrm{Fin}\, n}{\mathrm{var}\, i \in \mathrm{Ne}\, n} \qquad \frac{t \in \mathrm{Ne}\, n \quad u \in \mathrm{Nf}\, n}{\mathrm{app}\, t\, u \in \mathrm{Ne}\, n}$$

$$\frac{t \in \mathrm{Lam}\,(n+1)}{\mathrm{lam}\, t \in \mathrm{Lam}\, n} \qquad \frac{t \in \mathrm{Ne}\, n}{t \in \mathrm{Nf}\, n}$$

Inductive families like this are the backbone of recent dependently typed programming as present in Epigram or Agda [22]. Coq supports the definition of inductive families but made programming with them rather hard — the situation has been now improved due to the new `Program` tactic

[1] Here we shall view Haskell as an approximation of strong functional programming as proposed by Turner [24] and ignore non-termination.

[21]. Using Generalized Algebraic Datatypes (GADTs) [8] Fin and Lam can 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. We could have gone further and expressed this by predicating the constructors with a type level predicate `Nat`. Note that `Lam` is actually just a nested datatype [6] while `Fin` exploits the full power of GADTs because the range of the constructors is constrained.

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 λ-terms this indexing type was $\mathbb{N}$. The objects of the category of families indexed over a type $I \in \mathbf{Set}$ are I-indexed families, i.e. functions of type $I \to \mathbf{Set}$, and a morphism between I-indexed families $A, B \in I \to \mathbf{Set}$ is given by a family of maps $f \in \Pi i \in I.A\,i \to B\,i$. Indeed, this category is easily seen to be isomorphic to the slice category $\mathbf{Set}/I$ but the chosen representation is more convenient type-theoretically. Using Σ-types and equality types from Type Theory, we can define the following endofunctors F_{Fin} and F_{Lam} on the category of families over $\mathbb{N}$ whose initial algebras are Fin and Lam, respectively:

$$\begin{array}{rcl} F_{\mathrm{Fin}}, F_{\mathrm{Lam}} & \in & (\mathbb{N} \to \mathbf{Set}) \to \mathbb{N} \to \mathbf{Set} \\ F_{\mathrm{Fin}}\,A\,n & = & \Sigma m \in \mathbb{N}.(m+1=n) \times (1 + A\,m) \\ F_{\mathrm{Lam}}\,A\,n & = & \mathrm{Fin}\,n + A\,n \times A\,n + A\,(n+1) \end{array}$$

We are using equality types to express the focussed character of the constructors for Fin. This corresponds to the use of GADTs in the Haskell encoding.

This approach extends uniformly to more complicated examples such as the family of typed λ-terms, using lists of types $[\mathrm{Ty}]$ to represent typing contexts:

$$\begin{array}{rcl} \mathrm{Ty} & \in & \mathbf{Set} \\ \mathrm{Var}, \mathrm{Lam} & \in & [\mathrm{Ty}] \to \mathrm{Ty} \to \mathbf{Set} \end{array}$$

given by the following constructors

$$\frac{}{\mathrm{ty} \in \mathrm{Ty}} \qquad \frac{\sigma, \tau \in \mathrm{Ty}}{\mathrm{arr}\,\sigma\,\tau \in \mathrm{Ty}}$$

$$\frac{}{\mathrm{vzero} \in \mathrm{Var}\,(\sigma : \Gamma)\,\sigma} \qquad \frac{x \in \mathrm{Var}\,\Gamma\,\sigma}{\mathrm{vsucc}\,x \in \mathrm{Var}\,(\tau : \Gamma)\,\sigma}$$

$$\frac{x \in \mathrm{Var}\,\Gamma\,\sigma}{\mathrm{var}\,x \in \mathrm{Lam}\,\Gamma\,\sigma} \qquad \frac{t \in \mathrm{Lam}\,\Gamma\,(\mathrm{arr}\,\sigma\,\tau) \quad u \in \mathrm{Lam}\,\Gamma\,\sigma}{\mathrm{app}\,t\,u \in \mathrm{Lam}\,\Gamma\,\tau}$$

$$\frac{t \in \mathrm{Lam}\,(\sigma : \Gamma)\,\tau}{\mathrm{lam}\,t \in \mathrm{Lam}\,\Gamma\,(\mathrm{arr}\,\sigma\,\tau)}$$

Types like this can be used to implement a tag-free, terminating evaluator [4]. To obtain the corresponding functors is a laborious but straightforward exercise.

Containers

The initial algebra semantics is useful to provide a generic analysis of inductive types exploiting generic concepts such as constructors and map-function. However, doesn't say whether inductive types actually exist and it falls short to provide a systematic construction of generic operations such as equality or the zipper [13, 16].

In previous work, [2, 1], we have proposed the notion of a container type: A (unary) container is given by a set of shapes $S \in \mathbf{Set}$ and a family of positions $P \in S \to \mathbf{Set}$ assigning to each shape, the set of positions where data can be stored in a data structure of that shape. We write $S \lhd P$ for a container, for example the type of lists is given by $\mathbb{N} \lhd \mathrm{Fin}$ indicating that the shape of a list is a natural number (its length) and lists of length n have $\mathrm{Fin}\,n$ positions where data is stored. Every container gives rise to a functor, and containers are closed under forming products, coproducts, constant exponentiation and taking initial algebras and terminal coalgebras which model lazy datatypes. Hence, the theory of containers also provides a convenient way to express that a category has all strictly positive datatypes. We have introduced the notion of a container morphism and showed that they uniquely capture polymorphic functions/natural transformations between the functors generated by the containers. We have related containers to strictly positive datatypes and showed that every strictly positive datatype gives rise to a container. The categorical infrastructure required for this interpretation is quite modest, it works in any locally cartesian closed category with finite types (i.e. an initial object and a disjoint boolean object) and W-types.

While containers provide an elegant foundation for generic programming with inductive types, the present pa-

per develops the theory of *indexed containers* which allows us to provide a similar foundation for programming with inductive families like Fin and Lam. Our results give us the best of both worlds in that the theory of ordinary containers carries over to the more expressive indexed setting, e.g. we can again establish a notion of container morphism such that the interpretation functor is full and faithful. Maybe most surprisingly all this additional expressivity comes for free, as far as the type-theoretic infrastructure is concerned — W-types are still enough.

Main results

We introduce the notion of indexed container which generalizes containers allowing us to represent inductive families. This is a further step from *dependent polynomial functors* [10] representing endofunctors on a slice category. Indexed containers as introduced in the present paper allow to represent functors between different slices and capture also mutual and nested inductive definitions.

While [10] show that dependent polynomial functors always have initial algebras, we show that indexed containers are closed under parametrized initial algebras. Hence we can apply the fixpoint construction several times. The flexibility of indexed containers allows us to also establish closure under the adjoints of reindexing which leads directly to a grammar for strictly positive families, which itself is an instance of a strictly positive family section 7) — see also our previous work [18, 19].

Our presentation here uses type theoretic notation while our own previous work on containers and [10] used categorical notions. While this can be more bureaucratic in places, it leads directly to an implementation in a dependently typed programming language. This is witnessed by our formalisation of the main results of this paper in Agda (see appendix A).

Related Work

We have already discussed the relation to our own work on containers and strictly positive families and to dependent polynomial functors.

Containers are related to normal functors [11] which themselves are a special case of analytic functors [14] — those allow only finite sets of positions. Fiore's work on generalized species [9] considers those concept in a more generic setting — the precise relation to indexed containers remain to be explored.

Perhaps the earliest publication related to indexed containers occurs in Petersson and Synek's paper [20] 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.

Inspired in part by Petersson and Synek's constructor, Hancock, Hyvernat and Setzer [12] 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.

Acknowledgments

The present paper grew out of previous joint work with Neil Ghani, Peter Hancock and Conor McBride. Neil Ghani provided very elegant proofs of propositions 1 and 2. We profited greatly from comments and feedback from our colleagues in the functional programming laboratory in Nottingham.

2 Type theoretic preliminaries

We work in an extensional Type Theory [15] with the following ingredients:

$\mathbf{Set}_i$ A cumulative hierarchy of universes $\mathbf{Set}_i \in \mathbf{Set}_{i+1}$ for $i \in \mathbb{N}$, cumulativity means that $A : \mathbf{Set}_i$ implies $A : \mathbf{Set}_{i+1}$. We will omit the indices in our development pretending that $\mathbf{Set} : \mathbf{Set}$ but making sure that all our definitions can be stratified.

$0, 1$ An empty type $0 \in \mathbf{Set}_i$ and a unit type $1 \in \mathbf{Set}$. Categorically, those correspond to initial and terminal objects. For any set A we write $?_A \in 0 \to A$ for the unique map of this type. We write $() \in 1$ for the unique inhabitant of 1 and $!_A \in A \to 1$ with $!_A\, a = ()$ for the unique map into 1.

2 A type of Booleans $2 \in \mathbf{Set}_i$, which is disjoint, i.e. we have that $(\mathrm{true} = \mathrm{false}) \to 0$ is inhabited.

Σ- and Π-types Given $A \in \mathbf{Set}_i$ and $B \in \mathbf{Set}_i$ given that $x \in A$ then $\Sigma x \in A.B, \Pi x \in A, B \in \mathbf{Set}_i$. Elements of Σ-types are pairs, if $a \in A$ and $b \in B[x := a]$ then $(a, b) \in \Sigma x \in A.B$, while elements of Π-types are functions: given $b \in B$ assuming $x \in A$ then $\lambda x.b \in \Pi x \in A.B$[2]

Equality types Given $a, b \in A \in \mathbf{Set}_i$ we write $a = b \in \mathbf{Set}_i$ for the equality type. The constructor for equality is reflexivity $\mathrm{refl}\, a \in a = a$ if $a \in A$.

W-types As for Σ and Π Given $A \in \mathbf{Set}_i$ and $B \in \mathbf{Set}_i$ given that $x \in A$ then $\mathrm{W}\, x \in A.B \in \mathbf{Set}_i$. The elements of a W-type are well-founded trees which are constructed using sup: if $w \in \Sigma x \in A.B \to \mathrm{W}\, x \in A.B$ then $\mathrm{sup}\, w \in \mathrm{W} x \in A.B$. [3]

[2] We use untyped λ-abstraction and make sure that the type can be inferred from the context.

[3] Usually the type of sup is curried and sup has two arguments instead. This version is more convenient for our purposes.

We omit a detailed treatment of eliminators and use functional programming notation, like if-then-else, dependently typed pattern matching and structural recursion as present in Agda and Epigram. All our definitions can be translated into using the standard eliminators at the price of readability. To avoid clutter we adopt the usual type-theoretic device of allowing hidden arguments, if they are inferable from the use. We indicate hidden arguments by subscripting the type, i.e. writing $\Pi_{x\in A}B$ and $\Sigma_{x\in A}B$ instead $\Pi x \in A.B$ and $\Sigma x \in A.B$.

If we ignore the predicative hierarchy, the categorical infrastructure corresponds to Locally Cartesian Closed category, with initial objects, a disjoint boolean object and initial algebras for functors of the form

$$F_{\mathrm{W}x\in A.B}X = \Sigma x \in A.B \to X$$

While finite products arise as non-dependent Σ-types, finite coproducts can be represented as

$$A + B = \Sigma b \in 2.\mathrm{if}\, b\, \mathrm{then}\, A\, \mathrm{else}\, B$$

Given $f \in A \to C, g \in B \to C$ we define

$$\begin{aligned}&\mathrm{case}\, f\, g \in A + B \to C\\ &\mathrm{case}\, f\, g\, (b, x) = \mathrm{if}\, (f\, x)\, (g\, x)\, b\end{aligned}$$

Replacing Σ by Π this gives rise to an alternative way to define $\times$.

3 Indexed functors

Given $I \in \mathbf{Set}$ we consider the category of families over I. It's objects are families of types $A \in I \to \mathbf{Set}$ and given $A, B \in I \to \mathbf{Set}$ a morphism f is a family of functions $f \in \Pi_{i\in I} A\, i \to B\, i$, identity and composition are obvious. We denote this category as $\mathrm{Fam}\, I$. Indeed $\mathrm{Fam}\, I$ is isomorphic to the slice category $\mathbf{Set}/I$ whose objects are morphisms $J \to I$ for some $J \in \mathbf{Set}$ and whose morphisms are commuting triangles.

An indexed functor over $I \in \mathbf{Set}$ is a functor from $\mathrm{Fam}\, I$ to $\mathbf{Set}$, to make this precise an indexed functor F is given by

$$\begin{aligned}F_{obj} &\in (I \to \mathbf{Set}) \to \mathbf{Set}\\ F_{\mathrm{m}or} &\in \Pi_{A,B\in I\to\mathbf{Set}}(\Pi_{i\in I} A\, i \to B\, i) \to F\, A \to F\, B\end{aligned}$$

subject to the conditions that $F_{\mathrm{m}or}$ preserves identity and composition. As usual we omit the annotations when the meaning is clear from the context and overload F to mean F_{obj} and $F_{\mathrm{m}or}$.

Given two indexed functors F, G a family of maps

$$\alpha \in \Pi A \in I \to \mathbf{Set}.F\, A \to G\, A$$

is a natural transformation, if the naturality condition holds. We write $F \Rightarrow G$ for the set of natural transformations and $\mathrm{Func}\, I$ for the category of indexed functors and natural transformations.

Func comes with a monad-like structure, we have

$$\begin{aligned}&\eta^{\mathrm{F}} \in I \to \mathrm{Func}\, I\\ &\eta^{\mathrm{F}}\, i\, A = A\, i\end{aligned}$$

and given $H \in I \to \mathrm{Func}\, J$ and $F \in \mathrm{Func}\, I$ we construct

$$\begin{aligned}&H \gg\!=^{\mathrm{F}} F \in \mathrm{Func}\, J\\ &(H \gg\!=^{\mathrm{F}} F)\, A = F\, (\lambda i.H\, i\, A)\end{aligned}$$

it is easy to see that the usual equations for a monad hold. It is not actually a monad since $\mathrm{Func} : \mathbf{Set}_i \to \mathbf{Set}_{i+1}$ is not closed at any level. Func is. however, a monoid in the category of endofunctors from $\mathbf{Set}_i$ to $\mathbf{Set}_{i+1}$.

The opposite of the Kleisli category associated to Func has as objects sets $I, J \in \mathbf{Set}$ and morphisms given by an J-indexed family of I-indexed functors. We overload Func and write $\mathrm{Func}\, I\, J = J \to \mathrm{Func}\, I$. We use the opposite of the Kleisli category so that $F \in \mathrm{Func}\, I\, J$ is a functor from $\mathrm{Fam}\, I$ to $\mathrm{Fam}\, J$. Indeed given $A \in \mathrm{Fam}\, I$ we write:

$$\begin{aligned}&F@A \in \mathrm{Fam}\, J\\ &(F@A)\, j = F\, j\, A\end{aligned}$$

This construction also extends to natural transformations, i.e. the set of natural transformation between $F, G \in \mathrm{Func}\, I\, O$ is given by $\Pi o \in O.F\, o \Rightarrow G\, o$.

There are several operations which we can use to construct indexed functors. Clearly the Kleisli structure gives identities and composition in Func. Assuming $f \in O \to O'$, reindexing is defined by composition:

$$\begin{aligned}&\Delta^{\mathrm{F}}\, f \in \mathrm{Func}\, I\, O' \to \mathrm{Func}\, I\, O\\ &\Delta^{\mathrm{F}}\, f\, F = F \circ f\end{aligned}$$

Notice the contravariance in the above definition which is to be expected of reindexing. An example of the use of reindexing occurs in the definition of the indexed functor whose fixed point is Lam where we need to turn an indexed functor F into an indexed functor F' defined by $F'\, n\, X = F\, (n + 1)\, X$. In this situation, F' is just $\Delta^{\mathrm{F}}\, (+1)\, F$.

Unsurprisingly, $\Delta^{\mathrm{F}}\, f$ has both left and right adjoints, those are given by

$$\begin{aligned}&\Sigma^{\mathrm{F}}\, f, \Pi^{\mathrm{F}}\, f \in \mathrm{Func}\, I\, O \to \mathrm{Func}\, I\, O'\\ &\Sigma^{\mathrm{F}}\, f\, F = \lambda X, o.\Sigma o' \in O'.f\, o = o' \times F\, X\, o'\\ &\Pi^{\mathrm{F}}\, f\, F = \lambda X, o.\Pi o' \in O'.(f\, o = o') \to F\, X\, o'\end{aligned}$$

Σ^{F} is used in the construction of F_{Fin}:

$$F_{\mathrm{Fin}} = F\, \mathrm{succ}\, (\lambda A\, m.1 + A\, m)$$

Finite coproducts and products of indexed functors arise as special cases of Σ^{F} and Π^{F}. Given $F, G \in \mathrm{Func}\, I$ we construct

$$\begin{aligned}
&H_0 \in \mathrm{Func}\, I\, 0\\
&H_0 = ?_{\mathrm{Func}\, I}\\
&H_2 \in \mathrm{Func}\, I\, 2\\
&H = \lambda b.\mathrm{if}\, b\, \mathrm{then}\, F\, \mathrm{else}\, G
\end{aligned}$$

Using the unique forgetful functions $!_A \in A \to 1$ we obtain

$$\begin{aligned}
&\mathrm{K}_0^{\mathrm{F}}, F +^{\mathrm{F}} G, \mathrm{K}_1^{\mathrm{F}}, F \times^{\mathrm{F}} G \in \mathrm{Func}\, I\, 1 \simeq \mathrm{Func}\, I\\
&\mathrm{K}_0^{\mathrm{F}} = \Sigma^{\mathrm{F}}\, !_0\, H_0\\
&F +^{\mathrm{F}} G = \Sigma^{\mathrm{F}}\, !_2\, H_2\\
&\mathrm{K}_1^{\mathrm{F}} = \Pi^{\mathrm{F}}\, !_0\, H_0\\
&F \times^{\mathrm{F}} G = \Pi^{\mathrm{F}}\, !_2\, H_2
\end{aligned}$$

More concretely $\mathrm{K}_0^{\mathrm{F}}, \mathrm{K}_1^{\mathrm{F}}$ are simply the constant functors returning 0 (resp. 1), and $F +^{\mathrm{F}} G$ and $F \times^{\mathrm{F}} G$ are the point-wise coproducts (resp. coproducts) of F and G. The point of the reduction to Σ^{F} and Π^{F} is to be able to avoid having to apply the same construction several times later.

In the examples we only used finite products, the more general case of Π^{F} is relevant when we define infinitely branching trees.

4 Initial algebras of indexed functors

Given an indexed endofuncor $F \in \mathrm{Func}\, I\, I$ we can apply the usual notion of initial algebra, To spell it out: an F-algebra (A, a) is a a family $A : \mathrm{Fam}\, I$ together with a function in $\mathrm{Fam}\, I$: $a : \Pi_{i \in I}(F@A)\, i \to A\, i$. Given F-algebras (A, a) and (B, b) a morphism is a function $f : \Pi_{i \in I} A\, i \to B\, i$ s.t.

$$\begin{array}{ccc}
F@A & \xrightarrow{a} & A\\
{\scriptstyle F@f}\downarrow & & \downarrow{\scriptstyle f}\\
F@B & \xrightarrow{b} & B
\end{array}$$

An initial F-algebra $(\mu^0\, F, \mathrm{in})$ is the initial object in this category, if it exists. Initial algebras of functors don't always exist as we already know from the non-indexed case (which arises as a special case of indexed functors where $I = 1$). For example the functor $F \in \mathbf{Set} \to \mathbf{Set}$ given by $F\, A = (A \to 2) \to 2$ hasn't got an initial algebra in a predicative theory.

We can apply the initial algebra construction only once, because it takes is us from an indexed functor $F \in \mathrm{Func}\, I\, I$ to the category of families $\mu^0\, F \in \mathrm{Fam}\, I$. Hence we can't define nested or mutual inductive families. To overcome this we introduce parametrized initial objects.

Coproducts of indexed functors correspond to indexed functors on the product of slices, i.e. just by looking at the object part:

$$\begin{aligned}
&\mathrm{Func}\, (I + J)\\
&= (I + J \to \mathbf{Set}) \to \mathbf{Set}\\
&\simeq (I \to \mathbf{Set}) \times (J \to \mathbf{Set}) \to \mathbf{Set}
\end{aligned}$$

We can curry the last line:

$$\simeq (I \to \mathbf{Set}) \to (J \to \mathbf{Set}) \to \mathbf{Set}$$

This gives rise to partial application of an indexed functor which we will employ in our definition of parametrized initial algebras.

$$\frac{F \in \mathrm{Func}\, (I + J)\, J \qquad G \in \mathrm{Func}\, I\, J}{F[G] \in \mathrm{Func}\, I\, J}$$

$$F[G]\, j\, A = F\, j\, (\mathrm{case}\, A\, (G@A))$$

This definition is functorial, i.e. given $\alpha \in G \Rightarrow H$ we obtain $F[\alpha] \in F[G] \Rightarrow F[H]$.

Given $F \in \mathrm{Func}\, (I + J)\, J$ a parametrized F-algebra is given by an indexed functor $G \in \mathrm{Func}\, I\, J$ and a natural transformation $\alpha \in F[G] \Rightarrow G$. A morphism between two parametrized F-algebras (G, α) and (H, β) is a natural transformation $\gamma \in G \Rightarrow H$ such that

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

The parametrized initial algebra of F, i.e. the initial object in the category of parametrized F-algebras, we denote by $(\mu^I\, F, \mathrm{in}_F)$, if it exists. The special case of $I = 0$ corresponds to non-parametrized algebras.

5 Indexed containers

Given $I \in \mathbf{Set}$ we define the category $\mathrm{Cont}\, I$ of indexed containers over I, together with an extension functor $[\![-]\!]^{\mathrm{C}} \in \mathrm{Cont}\, I \to \mathrm{Func}\, I$. While an ordinary container is a pair of a set of shapes and a family of positions indexed by shapes, an indexed container also consists of a set of shapes

$$S \in \mathbf{Set}$$

and a family of positions indexed by shapes and the indexing set I:

$$P \in S \to I \to \mathbf{Set}.$$

We denote this container as

$$S \lhd P \in \mathrm{Cont}\, I.$$

Its extension is given as:

$$\begin{aligned}
&[\![S \lhd P]\!]^{\mathrm{C}} \in \mathrm{Func}\, I \\
&[\![S \lhd P]\!]^{\mathrm{C}}{}_{obj}\, A = \Sigma s \in S.\Pi_{i \in I} P\, s\, i \to A\, i \\
&[\![S \lhd P]\!]^{\mathrm{C}}{}_{mor}\, f\, (s, h) = (s, \lambda i \in I.f\, i \circ h\, i)
\end{aligned}$$

Given indexed containers $S \lhd P, T \lhd Q \in \mathrm{Cont}\, I$ a container morphism is given by a function on shapes

$$f \in S \to T$$

and an I-indexed contravariant function on positions

$$r \in \Pi_{s \in S, i \in I} Q\, (f\, s)\, i \to P\, s\, i.$$

We write the morphism as

$$f \lhd r \in \mathrm{Cont}\, I\, (S \lhd P)\, (T \lhd Q)$$

The extension of a container morphism is a natural transformation between the associated functors given by:

$$\begin{aligned}
&[\![f \lhd r]\!]^{\mathrm{C}} \in \Pi X \in I \to \mathbf{Set}.[\![S \lhd P]\!]^{\mathrm{C}}\, X \to [\![T \lhd Q]\!]^{\mathrm{C}}\, X \\
&[\![f \lhd r]\!]^{\mathrm{C}}\, X\, .(s, h) = (f\, s, \lambda q.h\, (r\, q))
\end{aligned}$$

As for ordinary containers (see [2]) we can show

Proposition 1 *The extension functor* $[\![-]\!]^{\mathrm{C}} \in \mathrm{Cont}\, I \to \mathrm{Func}\, I$ *is full and faithful.*

We will now lift the operations we have previously defined on indexed functors to indexed containers. However, as we will see – unlike indexed functors – indexed containers always admit initial algebras. We start with identifying the monadic structure on Cont which is preserved by $[\![-]\!]^{\mathrm{C}}$:

$$\begin{aligned}
&\eta^{\mathrm{C}} \in I \to \mathrm{Cont}\, I \\
&\eta^{\mathrm{C}}\, i = (1, \lambda s, j.i = j)
\end{aligned}$$

Given $H = \lambda i.S\, i \lhd P\, i \in I \to \mathrm{Cont}\, J$ with $S \in I \to \mathbf{Set}$ and $P \in \Pi i \in I.S\, i \to J \to \mathbf{Set}$ and $T \lhd Q \in \mathrm{Cont}\, I$:

$$\begin{aligned}
&H \gg\!=^{\mathrm{C}} T \lhd Q \in \mathrm{Cont}\, J \\
&(H \gg\!=^{\mathrm{C}} T \lhd Q) = U \lhd R
\end{aligned}$$

with

$$\begin{aligned}
&U \in \mathbf{Set} \\
&U = [\![T \lhd Q]\!]^{\mathrm{C}}\, S \\
&R \in U \to I \to \mathbf{Set} \\
&R\, (t, f)\, i = \Sigma q \in Q\, t\, i, j \in J.P\, i\, (F\, i\, q)\, j
\end{aligned}$$

The operations satisfy the monadic laws but suffer the from the same size issue as the monadic structure on indexed functors. Moreover, the monadic structure is preserved by the extension operation.

As in the case of Func this gives rise to the category Cont, whose objects are sets $I, J \in \mathbf{Set}$ and morphisms are $\mathrm{Cont}\, I\, J = J \to \mathrm{Cont}\, I$. We will exploit that the elements of $\mathrm{Cont}\, I\, J$ can be isomorphically represented as a pair:

$$\begin{aligned}
&\vec{S} \in J \to \mathbf{Set} \\
&\vec{P} \in \Pi_{j \in J} \vec{S}\, j \to I \to \mathbf{Set}
\end{aligned}$$

and write $\vec{S} \lhd \vec{P} \in \mathrm{Cont}\, I\, J$.

The definition of reindexing carries over without change from indexed functors, i.e. given $f \in O \to O'$, we define $\Delta^{\mathrm{C}}\, f \in \mathrm{Cont}\, I\, O' \to \mathrm{Cont}\, I\, O$ by $\Delta^{\mathrm{C}}\, f\, F = F \circ f$ Moreover, we can define its adjoints on indexed containers:

$$\begin{aligned}
\Sigma^{\mathrm{C}}\, f, \Pi^{\mathrm{C}}\, f \quad &\in \mathrm{Cont}\, I\, O \to \mathrm{Cont}\, I\, O' \\
\Sigma^{\mathrm{C}}\, f(\vec{S} \lhd \vec{P}) \quad &= \lambda o.\Sigma o' \in O'.f\, o = o' \times \vec{S}\, o' \\
&\lhd \lambda o, (_, _, s).P\, s \\
\Sigma^{\mathrm{C}}\, f(\vec{S} \lhd \vec{P}) \quad &= \lambda o.\Pi_{o' \in O'} f\, o = o' \to \vec{S}\, o' \\
&\lhd \lambda o, g.\Sigma o \in O, e \in f\, o = p, \vec{P}\, (g\, e)
\end{aligned}$$

It is easy to see that $[\![\Sigma^{\mathrm{C}}\, f\, (S \lhd P]\!]^{\mathrm{C}} \simeq \Sigma^{\mathrm{F}}\, f\, [\![S \lhd P]\!]^{\mathrm{C}}$ and $[\![\Pi^{\mathrm{C}}\, f\, (S \lhd P)]\!]^{\mathrm{C}} \simeq \Pi^{\mathrm{F}}\, f\, [\![S \lhd P]\!]^{\mathrm{C}}$ and using proposition 1 we can conclude

Proposition 2 *Reindexing of indexed containers* $\Delta^{\mathrm{C}}\, f$ *has left and right adjoints*

$$\Sigma^{\mathrm{C}}\, f \vdash \Delta^{\mathrm{C}}\, f \vdash \Pi^{\mathrm{C}}\, F$$

and this structure is preserved by $[\![-]\!]^{\mathrm{C}}$.

Using the construction in section 3 this entails that indexed containers are closed under coproducts and products.

6 Initial algebras of containers

An indexed container $\vec{S} \lhd \vec{P} \in \mathrm{Cont}\, I\, I$ denotes an endofunctor on $\mathrm{Fam}\, I$. Unlike for functors in general, initial algebras of indexed containers always exist. While the initial algebra of an ordinary unary container $S \lhd P$ is given by the W-type $\mathrm{W}\, x \in S.P\, s$, the initial algebra of an indexed container is given by an indexed generalisation of W-types. That is given

$$\begin{aligned}
&\vec{S} \in I \to \mathbf{Set} \\
&\vec{P} \in I \to \Pi_{i \in I} \vec{S}\, i \to I \to \mathbf{Set}
\end{aligned}$$

we define

$$\mathrm{W}^I\, \vec{S}\, \vec{P} \in I \to \mathbf{Set}$$

as inductively generated by

$$\frac{i \in I \quad (s,f) \in \Sigma s \in \vec{S}\, i.\Pi s \in S\, i.\Pi_{j \in I} \vec{P}\, s\, j \to \mathrm{W}^I\, \vec{S}\, \vec{P}\, j}{\sup^I (s,f) \in \mathrm{W}^I\, \vec{S}\, \vec{P}\, j}$$

or with other words $\mathrm{W}^I\, \vec{S}\, \vec{P}$ is the initial algebra of $[\![S \lhd P]\!]^{\mathrm{C}}$. The interesting observation is that W^I can be constructed from ordinary W-types. The idea is to define an approximation of W^I as an ordinary W type, namely

$$\mathrm{W}\,(j,s) \in \Sigma j \in J.\vec{S}\, j \Sigma j' \in J.\vec{P}\, s\, j'$$

and then to identify the good trees by *type-checking*. Variants of this construction can be found in [2] and [10].

Proposition 3 $(\mathrm{W}^I\, \vec{S}\, \vec{P}, \sup_I)$*, i.e. the initial algebra of* $[\![\vec{S} \lhd \vec{P}]\!]^{\mathrm{C}}$ *can be constructed from* W*-types using* Π, Σ *and equality types. Equivalently it exists in any Martin-Löf category (Locally Cartesian Closed category with* W*-types).*

We are going to generalize this result to parametrized initial algebras. We first define partial application, given

$$\vec{S} \lhd \vec{P} \in \mathrm{Cont}\,(I+J)\,J$$
$$\vec{T} \lhd \vec{Q} \in \mathrm{Cont}\,I\,J$$

with

$$\vec{S} \in J \to \mathbf{Set}$$
$$\vec{P} \in \Pi_{j \in J} \vec{S}\, j \to I + J \to \mathbf{Set}$$
$$\vec{T} \in J \to \mathbf{Set}$$
$$\vec{Q} \in \Pi_{j \in J} \vec{T}\, j \to I \to \mathbf{Set}$$

we define

$$\vec{S} \lhd \vec{P}[\vec{T} \lhd \vec{Q}] = \vec{U} \lhd \vec{R}$$
$$\in \mathrm{Cont}\,I\,J$$

We observe that we can separate the I- and J-indexed positions of $\vec{P}$ by case analysis, giving rise to

$$\vec{P}^I \in \Pi_{j \in J} \vec{S}\, j \to I \to \mathbf{Set}$$
$$\vec{P}^J \in \Pi_{j \in J} \to \vec{S}\, j \to J \to \mathbf{Set}$$

Clearly $\vec{S} \lhd \vec{P}^J \in \mathrm{Cont}\,J\,J$ is a container, and we can use it's extension to construct the shapes of the resulting container:

$$\vec{U} \in J \to \mathbf{Set}$$
$$\vec{U} = [\![\vec{S} \lhd \vec{P}^J]\!]^{\mathrm{C}} @ \vec{T}$$

We also express the construction of the positions parametrically, using

$$\diamond \vec{Q} \in \Pi_{j \in J} \vec{U}\, j \to I \to \mathbf{Set}$$
$$\diamond \vec{Q}\,(s,f)\,i = \vec{P}^I\, s\, i + \Sigma_{k \in J} \Sigma p \in \vec{P}^J\, s\, k.\vec{Q}\,(f\,p)\,i$$

This represents the possible positions in the combined container which are either top-level positions ($\vec{P}^I$) or positions inside the second container given by a top-level ($\vec{P}^J$) position combined with a position in $\vec{Q}$ of the appropriate shape. Hence we set:

$$\vec{R} \in \Pi_{j \in J} \vec{U}\, j \to I \to \mathbf{Set}$$
$$\vec{R} = \diamond \vec{Q}$$

As expected this construction is functorial: on shapes we exploit the morphism part of $[\![\vec{S} \lhd \vec{P}^J]\!]^{\mathrm{C}}$. To construct the position part of the morphism we exploit that $\diamond$ is an indexed functor itself and it commutes with $[\![\vec{S} \lhd \vec{P}^J]\!]^{\mathrm{C}}$, i.e.

$$\diamond \vec{Q}'\,([\![\vec{S} \lhd \vec{P}^J]\!]^{\mathrm{C}}\, f\, x = \diamond\,(\vec{Q}' \circ f)\, x$$

We are going to reuse those components in the construction of the parametrized initial algebra of $\vec{S} \lhd \vec{P}$ above:

$$\mu^I = \vec{U}_\mu \lhd \vec{R}_\mu$$
$$\in \mathrm{Cont}\,I\,J$$

The shapes are the trees which can be constructed using $[\![\vec{S} \lhd \vec{P}^J]\!]^{\mathrm{C}}$, hence

$$\vec{U}_\mu \in J \to \mathbf{Set}$$
$$\vec{U}_\mu = \mathrm{W}^J\, \vec{S}\, \vec{P}^J$$

The positions are the paths in the resulting trees which can be constructed using structural recursion of the indexed W-type:

$$\vec{R}_\mu \in \Pi_{j \in J} \vec{U}_\mu\, j \to I \to \mathbf{Set}$$
$$\vec{R}_\mu\,(\sup^J (s,f)) = \diamond\, \vec{R}_\mu\,(s,f)$$

We are ready to define the constructor, i.e. the morphism part of the parametrized initial algebra:

$$\mathrm{in} = f \lhd r$$
$$\in \mathrm{Cont}\,(\vec{S} \lhd \vec{P}[\mu^I])\,\mu^I$$

Clearly, the shape component is just the constructor for W_J:

$$f \in \Pi_{j \in J} ([\![\vec{S} \lhd \vec{P}^J]\!]^{\mathrm{C}} @ \vec{U}_\mu)\, j \to \vec{U}_\mu\, j$$
$$f = \sup^J$$

The morphism on positions is just the identity, because we can exploit the recursive definition of $\vec{R}_\mu$:

$$r \in \Pi_{j \in J, i \in I} \Pi (s,f) \in \vec{U}_\mu\, j.\vec{R}_\mu\,(\sup^J (s,f)\, i \to \diamond\, \vec{R}\, u\, i$$
$$r\,(s,f)\,r = r$$

The definition of fold is straightforward, it is basically forced upon us and hence unicity holds. We summarize:

Proposition 4 (μ^I, in) *is the parametrized initial algebra of* $[\![\vec{S} \lhd \vec{P}]\!]^{\mathrm{C}}$.

7 Strictly positive families

Finally, we will define a syntactic representation of strictly positive families, which itself is a strictly positive indexed family. Given $J \in \mathbf{Set}$ we define $\mathrm{SPF} \in I \to \mathbf{Set}$ inductively. As before for Func and Cont we define $\mathrm{SPF}\, I\, J = J \to \mathrm{SPF}\, I$. This notation will subsequently be justified by establishing that SPF is monadic and hence gives rise to a category of strictly positive families.

$$\frac{i \in I}{\eta^{\mathrm{T}}\, i \in \mathrm{SPF}\, I}$$

$$\frac{O' \in \mathbf{Set} \quad f \in O' \to O \quad F \in \mathrm{SPF}\, O'\, I}{\begin{array}{c}\Pi^{\mathrm{T}}\, f\, F \in \mathrm{SPF}\, I\, O \\ \Sigma^{\mathrm{T}}\, f\, F \in \mathrm{SPF}\, O\, J\end{array}}$$

$$\frac{F \in \mathrm{SPF}\, (I + J)\, J}{\mu^{\mathrm{T}}\, F \in \mathrm{SPF}\, I\, J}$$

As before $\mathrm{SPF}\, I \in \mathbf{Set}_{i+1}$ if $I \in \mathbf{Set}_i$ because the constructors $\Sigma^{\mathrm{T}}, \Pi^{\mathrm{T}}$ are indexed by a set which should live on the same level as the index. Ignoring size again, SPF is monadic, η^{T} is a constructor. Given $F \in I \to \mathrm{SPF}\, J = \mathrm{SPF}\, J\, I$ and $G \in \mathrm{SPF}\, I$ we define $F \gg\!=^{\mathrm{T}} G \in \mathrm{SPF}\, J$ recursively over G:

$$\begin{aligned} F \gg\!=^{\mathrm{T}} (\eta^{\mathrm{T}}\, i) &= F\, i \\ F \gg\!=^{\mathrm{T}} (\Pi^{\mathrm{T}}\, f\, G\, j) &= \Pi^{\mathrm{T}}\, f\, (F \circ G))\, j \\ F \gg\!=^{\mathrm{T}} (\Sigma^{\mathrm{T}}\, f\, G\, j) &= \Sigma^{\mathrm{T}}\, f\, (F \circ G))\, j \\ F \gg\!=^{\mathrm{T}} (\mu^{\mathrm{T}}\, G\, j) &= \mu^{\mathrm{T}}\, (G \circ \mathrm{SPF}\, \mathrm{inl})\, j \end{aligned}$$

Here $F \circ G = \lambda o.F \gg\!=^{\mathrm{T}} (G\, o)$ is the derived composition operation and $\mathrm{SPF}\, f\, G = (\eta^{\mathrm{T}} \circ f) \gg\!=^{\mathrm{T}} G$. In this presentation the algorithm isn't structurally recursive but this can be addressed by defining $\mathrm{SPF}\, \mathrm{inl}$ separately along the lines of [3].

It is straightforward to define an interpretation of the syntax as indexed containers, that is given $A \in \mathrm{SPF}\, I$ we obtain $[\![A]\!]^{\mathrm{T}} \in \mathrm{Cont}\, I$. As before this automatically extends to morphisms: given $F \in \mathrm{SPF}\, I\, J$ we obtain $[\![F]\!]^{\mathrm{T}} \in \mathrm{Cont}\, I\, J$.

$$\begin{aligned} [\![\eta^{\mathrm{T}}\, i]\!]^{\mathrm{T}} &= \eta^{\mathrm{C}}\, i \\ [\![\Pi^{\mathrm{T}}\, f\, G\, j]\!]^{\mathrm{T}} &= \Pi^{\mathrm{C}}\, f\, [\![G]\!]^{\mathrm{T}}\, j \\ [\![\Sigma^{\mathrm{T}}\, f\, G\, j]\!]^{\mathrm{T}} &= \Sigma^{\mathrm{C}}\, f\, [\![G]\!]^{\mathrm{T}}\, j \\ [\![\mu^{\mathrm{T}}\, G\, j]\!]^{\mathrm{T}} &= \mu^{\mathrm{C}}\, [\![G]\!]^{\mathrm{T}}\, j \end{aligned}$$

The monadic structure is preserved by the interpretation function.

By combining $[\![-]\!]^{\mathrm{T}}$ and $[\![-]\!]^{\mathrm{C}}$ we obtain an interpretation of indexed types as indexed functors $[\![-]\!]^{\mathrm{F}} \in \mathrm{SPF}\, I \to \mathrm{Func}\, I$. Note that it us necessary to go via indexed containers, since – as we have remarked earlier – functors are not necessarily closed under initial algebras.

As an example we encode Fin and Lam as strictly positive families exploiting the construction of finite coproducts and products in the syntax:

$$\begin{aligned} &\mathrm{Fin}^T, \mathrm{Lam}^T \in \mathrm{SPF}\, 0\, \mathbb{N} \\ &\mathrm{Fin}^T = \mu^{\mathrm{T}}\, (\lambda n.\Sigma^{\mathrm{T}}\, \mathrm{succ}\, \mathbf{K}_0^{\mathrm{T}}\, n +^{\mathrm{T}} \Sigma^{\mathrm{T}}\, \mathrm{succ}\, (\eta^{\mathrm{T}} \circ \mathrm{inr}) \\ &\mathrm{Lam}^T = \mu^{\mathrm{T}}\, (\lambda n.(\mathrm{Fin}^T \circ (\mathrm{SPF}\, \mathrm{inl}))\, n \\ &\qquad +^{\mathrm{T}} \eta^{\mathrm{T}}\, (\mathrm{inr}\, n) \times^{\mathrm{T}} \eta^{\mathrm{T}}\, (\mathrm{inr}\, n) \\ &\qquad +^{\mathrm{T}} \eta^{\mathrm{T}}\, (\mathrm{inr}\, (\mathrm{succ}\, n)) \end{aligned}$$

8 Conclusions and further work

We have shown how 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 inductive families to a small collection of categorically inspired combinators. This is an interesting alternative to the complex syntactic schemes present in the *Calculus of Inductive Constructions* (CIC), or in the Agda and Epigram systems. 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 recent work on *Observational Type Theory* (OTT) [5] shows that this is not an obstacle to implementation.

Central to our construction is the notion of indexed container generalizing both simple containers and dependent polynomial functors. Indexed containers like simple containers are closed under an initial algebra construction and like dependent polynomial functors model inductive families. We were able to exploit recent progress in the implementation of the Agda system to give a certified implementation of our main results closely following the high level structure of our construction. We decided not to base our presentation on the formalisation to be able to ignore irrelevant details of the actual formalisation and provide a better narrative.

We haven't included coinductive families, i.e. terminal coalgebras of indexed functors, in the present paper. It seems clear that we could representing coinductive families by following [2]. There, we showed how to encode M-types, i.e. the coinductive counterpart of W-types, using a limit construction. To dualize the construction presented

here we have to replace the recursive definition of $\vec{R}^{\mu}$ by an inductive definition.

A more serious challenge are mutual inductively (or coinductively) defined families where one type depends on another. A typical example is the syntax of Type Theory itself which, to simplify, can be encoded by mutually defining contexts, types and terms:

$$\mathrm{Con} \in \mathbf{Set}$$
$$\mathrm{Ty} \in \mathrm{Con} \to \mathbf{Set}$$
$$\mathrm{Tm} \in \Pi\Gamma \in \mathrm{Con}.\mathrm{Ty}\,\Gamma \to \mathbf{Set}$$

This sort of definition can be justified using Dybjer and Setzer's inductive-recursive definitions. However, induction-recursion doesn't seem necessary here because there are no negative occurrences of types as in universe constructions. Can we extend indexed containers to capture this kind of families? A promising avenue seems to be to consider the interpretation of containers in the category of families or more generally telescopes or equivalently arrow categories.

References

[1] M. Abbott, T. Altenkirch, and N. Ghani. Categories of containers. In *Proceedings of Foundations of Software Science and Computation Structures*, 2003.

[2] M. Abbott, T. Altenkirch, and N. Ghani. Containers - constructing strictly positive types. *Theoretical Computer Science*, 342:3–27, September 2005. Applied Semantics: Selected Topics.

[3] T. Altenkirch. Logical relations and inductive/coinductive types. In *Computer Science Logic, 12th International Workshop, CSL '98*, LNCS 1584, pages 343–354, 1998.

[4] T. Altenkirch and J. Chapman. Big-Step Normalisation. *Accepted for publication in JFP*, 2008.

[5] T. Altenkirch, C. McBride, and W. Swierstra. Observational equality, now! In *PLPV '07: Proceedings of the 2007 workshop on Programming languages meets program verification*, pages 57–68, New York, NY, USA, 2007. ACM.

[6] T. Altenkirch and B. Reus. Monadic presentations of lambda terms using generalized inductive types. In *Computer Science Logic*, 1999.

[7] R. Bird and O. de Moor. Algebra of programming. *Deductive Program Design*, 1996.

[8] J. Cheney and R. Hinze. First-Class Phantom Types. *The Fun of Programming*, 2003.

[9] M. Fiore, N. Gambino, M. Hyland, and G. Winskel. The cartesian closed bicategory of generalised species of structures. *Journal of the London Mathematical Society*, 77(1):203, 2008.

[10] N. Gambino and M. Hyland. Wellfounded trees and dependent polynomial functors. In S. Berardi, M. Coppo, and F. Damiani, editors, *Types for Proofs and Programs (TYPES 2003)*, Lecture Notes in Computer Science, 2004.

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

[12] P. Hancock and P. Hyvernat. Programming interfaces and basic topology. *Annals of Pure and Applied Logic*, 2006.

[13] G. Huet. The Zipper. *Journal of Functional Programming*, 7(5):549–554, 1997.

[14] A. Joyal. Foncteurs analytiques et espèces de structures. In G. Labelle and P. Leroux, editors, *Combinatoire énumerative*, number 1234 in Lecture Notes in Mathematics, pages 126–159. Springer-Verlag, 1987.

[15] P. Martin-Löf. *Intuitionistic type theory*. Bibliopolis, 1984.

[16] C. McBride. The Derivative of a Regular Type is its Type of One-Hole Contexts. Available at `http://www.dur.ac.uk/c.t.mcbride/diff.ps`, 2001.

[17] C. McBride. Epigram, 2008. `http://www.e-pig.org/`.

[18] P. Morris, T. Altenkirch, and N. Ghani. Constructing strictly positive families. In *The Australasian Theory Symposium (CATS2007)*, January 2007.

[19] P. Morris, T. Altenkirch, and N. Ghani. A universe of strictly positive families. *Theory of Computation*, 2007.

[20] K. Petersson and D. Synek. 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*, volume 389 of *LNCS*. Springer Verlag, 1989.

[21] M. Sozeau. Subset Coercions in Coq. *TYPES 2006, LNCS*, 4502:237, 2007.

[22] The Agda Team. The agda wiki, 2009. `http://wiki.portal.chalmers.se/agda/agda.php`.

[23] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version 8.1*, 2008.

[24] D. Turner. Elementary strong functional programming. *First International Symposium on Functional Programming Languages in Education*, 1022:1–13, 1985.

A Appendix: Formalisation in Agda

We begin by defining Cont:

```
data Cont (I : Set) : Set1 where
  _◁_ : (S : Set) → (P : S → I → Set) → Cont I
sh : {I : Set} → Cont I → Set
sh (S ◁ P) = S
po : {I : Set} → (C : Cont I) → sh C → I → Set
po (S ◁ P) = P
ICont : Set → Set → Set1
ICont I O = O → Cont I
```

It will be useful to view ICont, as O-indexed shapes and positions, rather than being an O-indexed Cont:

```
data ICont* (I O : Set) : Set1 where
  _◁_ : (S : O → Set) → (P : (o : O) → S o → I → Set) → ICont* I O
ic* : {I O : Set} → ICont I O → ICont* I O
ic* C = (λ o → sh (C o)) ◁ λ o → po (C o)
*ic : {I O : Set} → ICont* I O → ICont I O
*ic (S ◁ P) o = S o ◁ P o
```

Similarly we define this view which allows us to see a Cont (I + J) as having 2 sets of positions:

```
data Cont+ (I J : Set) : Set1 where
  _◁_,_ : (S : Set) → (P : S → I → Set) →
    (Q : S → J → Set) → Cont+ I J
sh : {I J : Set} → Cont+ I J → Set
sh (S ◁ P, Q) = S
po^I : {I J : Set} → (C : Cont+ I J) → (sh C) → I → Set
po^I (S ◁ P, Q) = P
po^J : {I J : Set} → (C : Cont+ I J) → (sh C) → J → Set
po^J (S ◁ P, Q) = Q
c+ : {I J : Set} → Cont (I + J) → Cont+ I J
c+ (S ◁ P) = S ◁ (λ s i → P s (inl i)), (λ s j → P s (inr j))
ICont+ : Set → Set → Set → Set1
ICont+ I J O = O → Cont+ I J
data ICont+* (I J O : Set) : Set1 where
  _◁_,_ : (S : O → Set) → (P : (o : O) → S o → I → Set) →
    (Q : (o : O) → S o → J → Set) → ICont+* I J O
ic+* : {I J O : Set} → ICont+ I J O → ICont+* I J O
ic+* C = (\o → sh (C o)) ◁ (λ o → po^I (C o)), λ o → po^J (C o)
```

Morphisms in Cont I:

```
data _⇒_ {I : Set} (C D : Cont I) : Set where
  _◁_ : (f : sh C → sh D) →
        (r : (s : sh C) → (i : I) → po D (f s) i → po C s i) → C ⇒ D
shf : {I : Set} → {C D : Cont I} → C ⇒ D → sh C → sh D
shf (f ◁ r) = f
pof : {I : Set} → {C D : Cont I} → (m : C ⇒ D) → (s : sh C) → (i : I) →
      (po D) (shf m s) i → (po C) s i
pof (f ◁ r) = r
_·_ : {I : Set} {C D E : Cont I} → (f : D ⇒ E) → (g : C ⇒ D) → C ⇒ E
(a ◁ c) · (b ◁ d) = (λ s → a (b s)) ◁ λ s i r → d s i (c (b s) i r)
```

Morphisms in ICont I J:

```
data _⇒*_ {I J : Set} (C D : ICont I J) : Set where
  _◁_ : (f : (j : J) → sh (C j) → sh (D j)) →
        (r : (j : J) (s : sh (C j)) →
             (i : I) → po (D j) (f j s) i → po (C j) s i) →
        C ⇒* D
m* : {I J : Set} → {C D : ICont I J} →
  ((j : J) → C j ⇒ D j) → C ⇒* D
m* {_} {_} {C} {D} m with ic* C | ic* D
m* {_} {_} {C} {D} m | (S ◁ P) | (T ◁ Q) = (λ j → shf (m j)) ◁ λ j → pof (m j)
*m : {I J : Set} → {C D : ICont I J} →
     (C ⇒* D) → ((j : J) → C j ⇒ D j)
*m {_} {_} {C} {D} (f ◁ r) = (λ j → f j ◁ r j)
```

Give rise to functors, we do not prove the laws here, because we do not need them:

```
left_right : {I : Set} → Cont I → (I → Set) → Set
left_right {I} C T = Σ (sh C) (λ s → (i : I) → po C s i → T i)
_$_ : {I : Set} → {C D : Cont I} → (m : C ⇒ D) → {X : I → Set} →
      ⟦ C ⟧ X → ⟦ D ⟧ X
_$_ {_} (a ◁ b) (s, f) = a s, λ i q → f i (b s i q)
```

Before we define the least fixed point we define partial application. We begin by defining the functor U:

```
U : {I J : Set} → Cont (I + J) → (J → Set) → Set
U C T = ⟦ (sh C) ◁ (λ s j → po C s (inr j)) ⟧ T
mapU : {I J : Set} → (C : Cont (I + J)) → {T T' : J → Set} →
       (f : (j : J) → T j → T' j) → U C T → U C T'
mapU C f (s, g) = s, λ j p → f j (g j p)
```

And the indexed functor R:

```
R : {I J : Set} → (C : Cont (I + J)) → {T : J → Set} →
    (Q : (j : J) → T j → I → Set) → U C T → I → Set
R {_} {J} C Q (s, f) i =
    po C s (inl i)
  + Σ J (λ j → Σ (po C s (inr j)) (λ p → Q j (f j p) i))
mapR : {I J : Set} → (C : Cont (I + J)) → {T : J → Set} →
       {Q Q' : (j : J) → T j → I → Set}
       (f : (j : J) (t : T j) (i : I) → Q j t i → Q' j t i) →
       (u : U C T) (i : I) → R C Q u i → R C Q' u i
mapR C f (s, g) i (inl p)          = inl p
mapR C f (s, g) i (inr (j, (p, q))) = inr (j, (p, f j (g j p) i q))
```

Again, we do not need to prove the functor laws for U and R but it would be possible in an extensional style.
We will need this relationship between U and R:

```
φ : {I J : Set} (C : Cont (I + J)) {T T' : J → Set}
    (f : (j : J) → T j → T' j)
    (Q' : (j : J) → T' j → I → Set) (u : U C T) (i : I) →
    R C Q' (mapU C f u) i → R C (λ j' t i' → Q' j' (f j' t) i') u i
φ C f Q' (a, b) i x = x
```

We now define the partial application of a Cont (I + J) to an ICont I J, which gives rise to a functor:

```
_ [_] : {I J : Set} → (C : Cont (I + J)) → (D : ICont I J) → Cont I
C [D]     with ic* D
C [D]     | (T ◁ Q) = U C T ◁ R C Q
map [] : {I J : Set} (C : Cont (I + J)) → {D D' : ICont I J} →
         ((j : J) → D j ⇒ D' j) → C [D] ⇒ C [D']
map [] C {D} {D'} m with m* m
map [] C {D} {D'} m | (f ◁ r) =
  mapU C f ◁ λ u i x → mapR C r u i (φ C f (λ j → po (D' j)) u i x)
```

We can begin the construction of the least fixed point. The shapes inductive containers will be given by W-Types, here the constructor sup takes advantage of the functor U:

```
data W {I J : Set} (C : ICont (I + J) J) (j : J) : Set where
  sup : U (C j) (W C) → W C j
foldW : {I J : Set} (C : ICont (I + J) J) (D : J → Set) →
        ((j : J) → U (C j) D → D j) → (j : J) → W C j → D j
foldW C D m j (sup u) = m j (mapU (C j) (foldW C D m) u)
```

Note that U^{μ} from the paper is simply defined as W.
The postions will then be given by paths through such a tree, again we take advatage of the functor R.

```
R^μ : {I J : Set} → (C : ICont (I + J) J) → (j : J) → W C j → I → Set
R^μ C j (sup u) i = R (C j) (R^μ C) u i
```

Note that this definition is not structurally recursive, it is however terminating on the size of the W-type.
The fixed point of an ICont (I + J) J is an ICont I J:

```
μ : {I J : Set} → (F : ICont (I + J) J) → ICont I J
μ F j = W F j ◁ R^μ F j
```

The algebra contains a morphism from $\mathsf{F}\ [\mu\ \mathsf{F}]$ to $\mu\ \mathsf{F}$, given by:

```
in𝜇 : {I J : Set} (F : ICont (I + J) J) → (j : J) → (F j) [𝜇 F] ⇒ 𝜇 F j
in𝜇 F j with ic+* (\j → c+ (F j))
in𝜇 F j | (S ◁ P^I, P^J) = sup ◁ λ _ _ r → r
```

And the associated fold by:

```
fold𝜇 : {I J : Set} → (F : ICont (I + J) J) → (H : ICont I J) →
        ((j : J) → (F j) [H] ⇒ H j) → (j : J) → 𝜇 F j ⇒ H j
fold𝜇 {I} {J} F H m with m* m
fold𝜇 {I} {J} F H m | (a ◁ b) = *m (foldW F (λ j → sh (H j)) a ◁ d)
  where d : (j : J) (w : W F j) (i : I) →
            po (H j) (foldW F (λ j → sh (H j)) a j w) i → R^𝜇 F j w i
        d j (sup (s, f)) i q = mapR (F j) d (s, f) i (b j _ i q)
```

We show that the fold square commutes by observation on the action of the container functor.

```
fold𝜇□ : {I J : Set} (F : ICont (I + J) J) (H : ICont I J)
         (m : (j : J) → (F j) [H] ⇒ H j) → {X : I → Set}
         (j : J) → (x : ⟦ (F j) [𝜇 F] ⟧ X) →
           (((m j) · (map [] (F j) (fold𝜇 F H m))) $ x)
         ≡ (((fold𝜇 F H m j) · (in𝜇 F j)) $ x)
fold𝜇□ F H m j ((s, f), g) with m j
fold𝜇□ F H m j ((s, f), g) | (a ◁ b) = SigEq refl (≡ ext (λ x y → refl))
```