Constructing Strictly Positive Families

Peter Morris and Thorsten Altenkirch
School of Computer Science and Information Technology
University of Nottingham
{*pwm,txa*}*@cs.nott.ac.uk*

Abstract

We present an inductive definition of a universe containing codes for strictly positive families (SPFs) such as vectors or simply typed lambda terms. This construction extends the usual definition of inductive strictly positive types as given in previous joint work with McBride. We relate this to Indexed Containers, which were recently proposed in joint work with Ghani, Hancock and McBride. We demonstrate by example how dependent types can be encoded in this universe and give examples for generic programs.

Keywords: datatypes, containers, universes, generic programming, dependent types, Epigram

1. INTRODUCTION

In a dependently typed language like Epigram [9, 8, 5] generic programming is normal programing. This is achieved by defining a universe [**?**, **?**] consisting of a type of names $\mathsf{U} : \star$ and a family of elements $\mathsf{El} : \mathsf{U} \to \star$ indexed by type names. We have exploited this opportunity in [10] by defining the universe of regular tree types and developing generic programs and proofs for this universe. However, there is an obvious asymmetry in our previous definitions where we exploit the power of dependent types to encode a universe of non-dependent types. In the present paper we show how to correct this imbalance and construct universes of dependent types. We define the universe of strictly positive families, which contains codes for all datatypes definable within Epigram, including the definition of this universe itself. We also consider a smaller universe of regular families, which is the dependent counterpart of the universe of regular tree types. This smaller universe which excludes infinitely branching trees is interesting because it allows more programs including a generic program to decide equality.

While the universe constructions present a family of types by an inductive definition, e.g. by giving the constructions under which the universe is closed we can alternatively give a semantic characterisation of a class of types by containers [1]. A simple 1-ary container is given by a type $\mathsf{S} : \star$ of shapes and a family of positions $\mathsf{P} : \mathsf{S} \to \star$. To every 1-ary container we can assign an endofunctor on the category of types. In loc.cit. we show that all strictly positive types can be interpreted as containers, hence containers can be viewed as a semantic normal form of strictly positive types. This gives an alternative access to generic programming: a generic program is simply one which works on all containers. We have exploited this approach to study derivatives of datatypes [2].

In yet unpublished work [3] we have generalized containers to indexed containers which capture dependent types. Given types $I, O : \star$ an indexed container $\mathsf{IC}\ I\ O$ represents an endofunctor form the slice category over I (whose objects are families $I \to \star$) to the slice category over O (whose objects are families $O \to \star$). In the present paper we relate the two approaches showing that any strictly positive type gives rise to a semantically equivalent indexed container.

In related work, [7] presents an alternative approach to defining a universe of indexed strictly positive definitions. This has been the base for generic programming within the AGDA system [6].

2. STRICTLY POSITIVE FAMILIES, REGULAR FAMILIES

We start by defining the type of strictly positive families $\mathsf{SPF}\ \vec{I}\ O$ representing functors from indexed by a sequence of input types $\vec{I}$ and an output type O. We use a finite sequence of input indices instead of just one type — this simplifies the encoding of families but adds nothing to the expressive power of the construction.

$$\underline{\text{data}}\ \frac{\vec{I} : \mathsf{Vec}\ \star\ n \quad O : \star}{\mathsf{SPF}\ \vec{I}\ O : \star}\ \underline{\text{where}}\ \frac{}{\text{`Z'} : \mathsf{SPF}\ (\vec{I} \mathbin{:} O)\ O} \quad \frac{T : \mathsf{SPF}\ \vec{I}\ O}{\text{`wk'}\ T : \mathsf{SPF}\ (\vec{I} \mathbin{:} I)\ O}$$

$$\frac{f : \forall t : \mathsf{Fin}\ n \Rightarrow \mathsf{SPF}\ \vec{I}\ O}{\text{`Tag'}\ f : \mathsf{SPF}\ \vec{I}\ (O \times \mathsf{Fin}\ n)} \quad \frac{}{\text{`0'}, \text{`1'} : \mathsf{SPF}\ \vec{I}\ O} \quad \frac{T : \mathsf{SPF}\ (\vec{I} \mathbin{:} O)\ O}{\text{`}\mu\text{'}\ T : \mathsf{SPF}\ \vec{I}\ O}$$

$$\frac{f : O \to O' \quad T : \mathsf{SPF}\ \vec{I}\ O}{\text{`}\Sigma\text{'}O\ f\ T : \mathsf{SPF}\ \vec{I}\ O'} \quad \frac{f : O' \to O \quad T : \mathsf{SPF}\ \vec{I}\ O}{\text{`}\Delta\text{'}O\ f\ T : \mathsf{SPF}\ \vec{I}\ O'}$$

$$\frac{f : O \to O' \quad T : \mathsf{SPF}\ \vec{I}\ O}{\text{`}\Pi^{\infty}\text{'}O\ f\ T : \mathsf{SPF}\ \vec{I}\ O'}$$

Let's explain the purpose of the different constructors:

'Z','wk' Access to the input types via deBruijn indices.

'Tag' Choice between n alternatives. Note that this is the only non-linear constructor, without we couldn't define non-linear types like trees.

'0', '1' The empty and the unit type.

'μ' Inductively defined families.

'Σ', 'Δ', 'Π^{∞}' 'Δ' is substitution on the output type (categorically modeled by a pullback) while 'Σ' represents its left adjoint and 'Π^{∞}' its right adjoint.

As in previous work, we now define an inductive family which, for any $T : \mathsf{SPF}\ \vec{I}\ O$ defines the type of its elements $[\![T]\!]^{\mathsf{T}}$. As before a crucial ingredient in for this construction is the type of closed type substitutions or telescopes Tel:

$$\underline{\text{data}}\ \frac{\vec{I} : \mathsf{Vec}\ \star\ n}{\mathsf{Tel}\ \vec{I} : \star}\ \underline{\text{where}}\ \frac{}{\varepsilon : \mathsf{Tel}\ \varepsilon} \quad \frac{\vec{T} : \mathsf{Tel}\ \vec{I} \quad T : \mathsf{SPF}\ \vec{I}\ I}{(\vec{T} \mathbin{:} T) : \mathsf{Tel}\ (\vec{I} \mathbin{:} I)}$$

Once we have defined telescopes we can define the type of elements inductively by giving the value constructors associated to each type constructor. As it has to be expected, there is none for the empty type '0'.

$$\underline{\text{data}}\ \frac{T : \mathsf{SPF}\ \vec{I}\ O \quad \vec{T} : \mathsf{Tel}\ \vec{I} \quad o : O}{[\![T]\!]^{\mathsf{T}}\ \vec{T}\ o : \star}\ \underline{\text{where}}$$

$$\frac{v : [\![T]\!]^{\mathsf{T}}\ \vec{T}\ \vec{X}\ o}{\mathsf{top}\ v : [\![\text{`Z'}]\!]^{\mathsf{T}}(\vec{T} \mathbin{:} T)\ o} \quad \frac{v : [\![T]\!]^{\mathsf{T}}\ \vec{T}\ \vec{X}\ o}{\mathsf{pop}\ v : [\![\text{`wk'}\ T]\!]^{\mathsf{T}}(\vec{T} \mathbin{:} S)\ o} \quad \frac{v : [\![f\ t]\!]^{\mathsf{T}}\ \vec{T}\ o}{\mathsf{tag}_t\ v : [\![\text{`Tag'}f]\!]^{\mathsf{T}}\ \vec{T}\ (o; t)}$$

$$\frac{}{\mathsf{void} : [\![\text{`1'}]\!]^{\mathsf{T}}\ \vec{T}\ o} \quad \frac{v : [\![T]\!]^{\mathsf{T}}\ \vec{T}\ o}{\sigma_o\ v : [\![\text{`}\Sigma\text{'}f\ T]\!]^{\mathsf{T}}\ \vec{T}\ (f\ o)} \quad \frac{v : [\![T]\!]^{\mathsf{T}}\ \vec{T}\ (f\ o)}{\delta\ v : [\![\text{`}\Delta\text{'}f\ T]\!]^{\mathsf{T}}\ \vec{T}\ o} \quad \frac{v : [\![T]\!]^{\mathsf{T}}(\vec{T} \mathbin{:} (\text{`}\mu\text{'}\ T))\ o}{\mathsf{in}\ v : [\![\text{`}\mu\text{'}\ T]\!]^{\mathsf{T}}\ \vec{T}\ o}$$

$$\frac{\vec{v} : \forall o : O \quad p : (f\ o) = o' \Rightarrow [\![T]\!]^{\mathsf{T}}\ \vec{T}\ o}{\pi^{\infty}\ \vec{v} : [\![\text{`}\Pi^{\infty}\text{'}f\ T]\!]^{\mathsf{T}}\ \vec{T}\ o'}$$

If we allow arbitrary functions for 'Π^∞' we obtain strictly positive types which only support very few generic operations, not including generic equality. An alternative in line with our previous work on regular tree types is the type of regular families $\mathsf{RF}\ \vec{I}\ O$ which is obtained by replacing 'Π^∞' by

$$\frac{n : \mathsf{Nat} \quad T : \mathsf{RF}\ \vec{I}\ (O \times \mathsf{Fin}\ n)}{\text{`}\Pi^{<\omega}\text{'}\, n\ T : \mathsf{RF}\ \vec{I}\ O}$$

whose elements can be constructed by

$$\frac{\vec{v} : \forall i : \mathsf{Fin}\ n \Rightarrow [\![T]\!]^{\mathsf{T}}\ \vec{T}\ (o; i)}{\pi^{<\omega} \vec{v} : [\![\text{`}\Pi^{<\omega}\text{'}\, n\ T]\!]^{\mathsf{T}}\ \vec{T}\ o}$$

The RF universe reflects what we shall call the Regular Families, which correspond to ω-continuous functors. There is an obvious embedding of the finite '$\Pi^{<\omega}$' in to the possibly infinite 'Π^∞' given by:

$$\begin{array}{lcl} \text{`}\Pi^{<\omega}\text{'}\, n\ T : \mathsf{RF}\ \vec{I}\ O & \mapsto & \text{`}\Pi^{\infty}\text{'}_{(O \times \mathsf{Fin}\ n)}\ \mathbf{fst}\ T : \mathsf{SPF}\ \vec{I}\ O \\ \pi^{<\omega} \vec{v} : [\![\text{`}\Pi^{<\omega}\text{'}\, n\ T ]\!]^{\mathsf{T}}\ \vec{T}\ o & \mapsto & \pi^{\infty} (\lambda (o; i); \mathsf{refl} \Rightarrow \vec{v}\ i) : [\![ \text{`}\Pi^{\infty}\text{'}_{(O \times \mathsf{Fin}\ n)}\ \mathbf{fst}\ T]\!]^{\mathsf{T}}\ \vec{T}\ o \end{array}$$

3. EXAMPLES OF SPFS

To give examples of datatypes in this universe, it is very useful to first define some auxiliary combinators for Cartesian product and disjoint union. We do this for RF universe since the constructions preserve finiteness:

$$\underline{\mathsf{let}}\ \frac{A, B : \mathsf{RF}\ \vec{I}\ O}{A \text{`}+\text{'}\ B, A \text{`}\times\text{'}\ B : \mathsf{RF}\ \vec{I}\ O}$$

$$A \text{`}+\text{'}\ B \ \Rightarrow\ \text{`}\Sigma\text{'}\mathbf{fst} \left(\text{`Tag'} \left(\begin{array}{lcl} \mathsf{fz} & \Rightarrow & A \\ \mathsf{fs\ fz} & \Rightarrow & B \end{array} \right) \right)$$

$$\underline{\mathsf{let}}\ \frac{a : [\![A]\!]^{\mathsf{T}}\ \vec{T}\ o}{\mathbf{inl}\ a : [\![A \text{`}+\text{'}\ B ]\!]^{\mathsf{T}}\ \vec{T}\ o} \qquad \underline{\mathsf{let}}\ \frac{b : [\![ B ]\!]^{\mathsf{T}}\ \vec{T}\ o}{\mathbf{inr}\ b : [\![ A \text{`}+\text{'}\ B]\!]^{\mathsf{T}}\ \vec{T}\ o}$$

$$\text{`}\mathbf{inl}\text{'}_{\vec{T}\ o}\, a \ \Rightarrow\ \sigma_{(o; \mathsf{fz})}\ (\mathsf{tag}\ a) \qquad \text{`}\mathbf{inr}\text{'}_{\vec{T}\ o}\, b \ \Rightarrow\ \sigma_{(o; \mathsf{fs\ fz})}\ (\mathsf{tag}\ b)$$

$$A \text{`}\times\text{'}\ B \ \Rightarrow\ \text{`}\Pi^{<\omega}\text{'}2 \left(\text{`Tag'} \left(\begin{array}{lcl} \mathsf{fz} & \Rightarrow & A \\ \mathsf{fs\ fz} & \Rightarrow & B \end{array} \right) \right)$$

$$\underline{\mathsf{let}}\ \frac{a : [\![A]\!]^{\mathsf{T}}\ \vec{T}\ o \quad b : [\![B]\!]^{\mathsf{T}}\ \vec{T}\ o}{\mathbf{pair}\ a\ b : [\![A \text{`}\times\text{'}\ B]\!]^{\mathsf{T}}\ \vec{T}\ o}$$

$$\mathbf{pair}\ a\ b \ \Rightarrow\ \pi^{<\omega} \left(\begin{array}{lcl} \mathsf{fz} & \Rightarrow & \mathsf{tag}\ a \\ \mathsf{fs\ fz} & \Rightarrow & \mathsf{tag}\ b \end{array} \right)$$

We can now as a first example define the type of vectors which in Epigram is given by:

$$\underline{\mathsf{data}}\ \frac{A : \star \quad n : \mathsf{Nat}}{\mathsf{Vec}\ A\ n : \star}\ \underline{\mathsf{where}}\ \frac{}{\varepsilon : \mathsf{Vec}\ A\ \mathsf{zero}}\ \frac{a : A \quad as : \mathsf{Vec}\ A\ n}{(a \mathbin{:} as) : \mathsf{Vec}\ A\ (1 + n)}$$

The type can becomes an instance of RF [One] Nat, here [One] encodes fact that there is one input type A which is trivially a family indexed by One and the resulting type is indexed by Nat:

$$\underline{\text{let}}\ \ \text{`Vec'} : \mathsf{RF}\ [\mathsf{One}]\ \mathsf{Nat} \quad \text{`Vec'} \Rightarrow \text{`}\mu\text{'}((\text{`}\Sigma\text{'}\ (\mathbf{const}\ 0)\ \text{`1'})\ \text{`}+\text{'}\ (\text{`}\Sigma\text{'}\ 1{+}\ (\text{`}\Delta\text{'}(\mathbf{const}\ ())\ (\text{`wk'}\ \text{`Z'})\ \text{`}\times\text{'}\ \text{`Z'}))$$

We can encode the constructors for vectors given above using the generic constructors:

$$\underline{\text{let}}\ \ \text{`}\varepsilon\text{'} : [\![\text{`Vec'}]\!]^{\mathsf{T}}[\text{`}A\text{'}]\ 0$$
$$\text{`}\varepsilon\text{'} \ \Rightarrow\ \mathsf{in}\ (\text{`inl'}\ (\sigma\_\ \mathsf{void}))$$

$$\underline{\text{let}}\ \ \frac{a : [\![\text{`}A\text{'}]\!]^{\mathsf{T}}[]\ ()\quad as : [\![\text{`Vec'}]\!]^{\mathsf{T}}[\text{`}A\text{'}]\ n}{(a\text{`::'}as) : [\![\text{`Vec'}]\!]^{\mathsf{T}}[\text{`}A\text{'}]\ (1{+}\ n)}$$
$$(a\text{`::'}as) \ \Rightarrow\ \mathsf{in}\ (\text{`inr'}\ (\text{`pair'}\ (\delta\ a)\ as))$$

We use the definitions above to present the type in a 'sums of products' style, with added indexing information. In the $\text{`}\varepsilon\text{'}$ case $\text{`}\Sigma\text{'}(\mathbf{const}\ 0)$ forces the output type to be zero on the top level, no matter what the index for the `1' is; in the $(\text{`::'})$ case $\text{`}\Sigma\text{'}(1{+})$ makes the output $1{+}\ n$ if the value underneath the constructor has index n.

As another example, we can encode lambda terms with n free variables in this way:

$$\underline{\text{data}}\ \ \frac{n : \mathsf{Nat}}{\mathsf{Lam}\ n : \star}\ \ \underline{\text{where}}\ \ \frac{i : \mathsf{Fin}\ n}{\mathsf{var}\ i : \mathsf{Lam}\ n}\quad \frac{f, a : \mathsf{Lam}\ n}{\mathsf{app}\ f\ a : \mathsf{Lam}\ n}\quad \frac{b : \mathsf{Lam}\ (1{+}\ n)}{\mathsf{abs}\ b : \mathsf{Lam}\ n}$$

$$\underline{\text{let}}\ \ \text{`Lam'} : \mathsf{RF}\ [] \ \mathsf{Nat}$$
$$\text{`Lam'} \Rightarrow \text{`}\mu\text{'}\ ((\text{`wk'}\ \text{`Fin'})\ \text{`}+\text{'}\ ((\text{`Z'}\ \text{`}\times\text{'}\ \text{`Z'})\ \text{`}+\text{'}\ (\text{`}\Delta\text{'}1{+}\ \text{`Z'})))$$

The above definitions satisfy syntactic conditions for strict positivity, as implemented in systems such as COQ or Epigram. A more delicate case are types where the strictly positive occurrence appears inside another inductively define type, such as n-branching trees:

$$\underline{\text{data}}\ \ \frac{A : \star \quad n : \mathsf{Nat}}{\mathsf{NBrTree}\ A\ n : \star}\ \ \underline{\text{where}}\ \ \frac{a : A}{\mathsf{leaf}\ a : \mathsf{NBrTree}\ A\ n}\quad \frac{\vec{t} : \mathsf{Vec}\ (\mathsf{NBrTree}\ A\ n)\ n}{\mathsf{node}\ \vec{t} : \mathsf{NBrTree}\ A\ n}$$

The translation of this definition is not completely straightforward, we need an extra piece of kit, *composition* of families. We define SPF composition by explaining what to insert at the variable positions of the 'outer' container, each of these 'inner' containers use the same type context, we generalise slightly by allowing the output index of the outer container to filter down to the inner containers:

$$\underline{\text{let}}\ \frac{\vec{I_A} : \mathsf{Vec}\ \star\ n \quad C : \mathsf{F}\ \vec{I_A}\ O \quad \vec{D} : \forall i : \mathsf{Fin}\ n \Rightarrow \mathsf{F}\ \vec{I_B}\ (\vec{I_A}!!i \times O)}{C{\circ}\vec{D} : \mathsf{F}\ \vec{I_B}\ O}$$

$$\begin{array}{rcl}
C{\circ}\vec{D} & \Leftarrow & \underline{\text{rec}}\ C \\
\quad \text{`Z'}{\circ}\vec{D} & \Rightarrow & \text{`}\Delta\text{'}(\lambda x \Rightarrow (x;x))\ (\vec{D}\ \mathsf{fz}) \\
\quad (\text{`wk'}\ T){\circ}\vec{D} & \Rightarrow & \text{`wk'}\ (T{\circ}\vec{D} \cdot \mathsf{fs}) \\
\quad \text{`0'}{\circ}\vec{D} & \Rightarrow & \text{`0'} \\
\quad \text{`1'}{\circ}\vec{D} & \Rightarrow & \text{`1'} \\
\quad (\text{`Tag'}f){\circ}\vec{D} & \Rightarrow & \text{`Tag'}(\lambda i \Rightarrow (f\ i){\circ}(\mathbf{map}\ (\text{`}\Delta\text{'}\mathbf{fst})\ \vec{D})) \\
\quad (\text{`}\Sigma\text{'}f\ T){\circ}\vec{D} & \Rightarrow & \text{`}\Sigma\text{'}f\ (T{\circ}(\mathbf{map}\ (\text{`}\Delta\text{'}(\mathbf{id};f))\ \vec{D})) \\
\quad (\text{`}\Delta\text{'}f\ T){\circ}\vec{D} & \Rightarrow & \text{`}\Delta\text{'}f\ (T{\circ}(\mathbf{map}\ (\text{`}\Sigma\text{'}(\mathbf{id};f))\ \vec{D})) \\
\quad (\text{`}\Pi^{<\omega}\text{'}n\ T){\circ}\vec{D} & \Rightarrow & \text{`}\Pi^{<\omega}\text{'}n\ (T{\circ}(\mathbf{map}\ (\text{`}\Delta\text{'}(\mathbf{id};\mathbf{fst}))\ \vec{D})) \\
\quad (\text{`}\Pi^{\infty}\text{'}f\ T){\circ}\vec{D} & \Rightarrow & \text{`}\Pi^{\infty}\text{'}f\ (T{\circ}(\mathbf{map}\ (\text{`}\Delta\text{'}(\mathbf{id};f))\ \vec{D})) \\
\quad (\text{`}\mu\text{'}\ F){\circ}\vec{D} & \Rightarrow & \text{`}\mu\text{'}\left(F{\circ}\begin{pmatrix} \mathsf{fz} & \Rightarrow & \text{`}\Sigma\text{'}(\lambda x \Rightarrow (x;x))\ \text{`Z'} \\ \mathsf{fs}\ i & \Rightarrow & \vec{D}\ i \end{pmatrix}\right)
\end{array}$$

We can now define **'NBrTree'** by right composing it with **'Vec'**:

$$\begin{array}{l}
\underline{\text{let}}\ \ \textbf{'NBrTree'} : \mathsf{RF}\ [\mathsf{One}]\ \mathsf{Nat} \\
\quad \textbf{'NBrTree'} \Rightarrow \text{`}\mu\text{'}\ (\text{`}\Delta\text{'}(\mathbf{const}\ ())\ (\text{`wk'}\ \text{`Z'})\ \text{`}+\text{'}\ \textbf{'Vec'}{\circ}(\mathsf{fz} \Rightarrow \text{`}\Delta\text{'}\mathbf{snd}\ \text{`Z'})
\end{array}$$

4. SPFS ARE CONTAINERS

The claim that our grammar of Strictly Positive (or Regular) Families relates directly to the notion of Container may need some justification.

The output and input sorts relate directly to our own notion of input and output, but for shapes and positions things may not be so clear, we claim that the translation below is what we require:

$$\underline{\text{data}}\ \frac{\vec{I} : \mathsf{Vec}\ \star\ n \quad O : \star}{\mathsf{IC}\ \vec{I}\ O : \star}\ \underline{\text{where}}\ \frac{S : O \to \star \quad P : \forall o : O;\ s : S\ o;\ i : \mathsf{Fin}\ n \Rightarrow \vec{I}!!i \to \star}{S \triangleleft P : \mathsf{IC}\ \vec{I}\ O}$$

$$\underline{\text{let}}\ \frac{T : \mathsf{SPF}\ \vec{I}\ O}{\mathbf{ICont}\ T : \mathsf{IC}\ \vec{I}\ O}$$

$$\begin{array}{rcl}
\mathbf{ICont}\ T & \Leftarrow & \underline{\text{rec}}\ T \\
\quad \mathbf{ICont}\ \text{`Z'} & \Rightarrow & (\lambda o \Rightarrow \mathsf{One}) \triangleleft \left(\lambda \begin{array}{cccc} out & s & \mathsf{fz} & in \Rightarrow (out = in) \\ out & s & (\mathsf{fs}\ i) & in \Rightarrow \mathsf{Zero} \end{array}\right) \\
\quad \mathbf{ICont}\ (\text{`wk'}\ T) & \Rightarrow & S \triangleleft \left(\lambda \begin{array}{cccc} out & s & \mathsf{fz} & in \Rightarrow \mathsf{Zero} \\ out & s & (\mathsf{fs}\ i) & in \Rightarrow P\ out\ s\ i\ in \end{array}\right) \\
\multicolumn{3}{r}{\underline{\text{where}}\ \ (S \triangleleft P) = \mathbf{ICont}\ T}
\end{array}$$

At variables we check to see if we are looking for the right input and that the indexes match up in which case we have a singleton position set.

$$\begin{array}{rcl}
\mathbf{ICont}\ \text{`0'} & \Rightarrow & (\lambda_ \Rightarrow \mathsf{Zero}) \triangleleft (\lambda_ \Rightarrow !) \\
\mathbf{ICont}\ \text{`1'} & \Rightarrow & (\lambda_ \Rightarrow \mathsf{One}) \triangleleft (\lambda_\ _\ _\ _ \Rightarrow \mathsf{Zero})
\end{array}$$

At the base types we have the obvious shapes, the position sets are empty as there is no room for data in these types.

$$\begin{array}{l}\mathbf{ICont}\ (\text{`Tag'}\ f)\ \Rightarrow\ (\exists i\ :\ \mathsf{Fin}\ n \Rightarrow S\ i) \triangleleft (\lambda out\ (i; s)\ j\ in \Rightarrow P\ i\ out\ s\ j\ in) \\ \qquad \underline{\text{where}}\ \ (\lambda i \Rightarrow (S\ i \triangleleft P\ i)) = \lambda i \Rightarrow \mathbf{ICont}\ (f\ i)\end{array}$$

At a tagged sum we calculate the container relating to each of the options, the shapes then become a pair of a tag and a shape for the corresponding container; similarly for Positions, we only consider the appropriate set for the tag we are given.

$$\begin{array}{l}\mathbf{ICont}\ (\text{`}\Sigma\text{'}f\ T)\ \Rightarrow\ (\lambda out' \Rightarrow \exists out\ :\ O \Rightarrow (f\ out = out')) \rightarrow S\ out) \\ \qquad \triangleleft (\lambda (f\ out)\ (out;\ \mathsf{refl};\ s)\ i\ in \Rightarrow P\ out\ s\ i\ in) \\ \qquad \underline{\text{where}}\ \ (S \triangleleft P) = \mathbf{ICont}\ T\end{array}$$

In the sigma case we add the new output index to the type of shapes, along with a proof that it relates to the given index according to f. Positions are then indexed by the new output index, extracted from the shape.

$$\begin{array}{l}\mathbf{ICont}\ (\text{`}\Delta\text{'}f\ T)\ \Rightarrow\ (\lambda out' \Rightarrow S\ (f\ out')) \triangleleft (\lambda out'\ s\ i\ in \Rightarrow P\ (f\ out')\ s\ i\ in) \\ \qquad \underline{\text{where}}\ \ (S \triangleleft P) = \mathbf{ICont}\ T\end{array}$$

In the re-indexing case we simply apply the function f to the given index.

$$\begin{array}{l}\mathbf{ICont}\ (\text{`}\Pi^{\infty}\text{'}f\ T)\ \Rightarrow\ (\lambda out' \Rightarrow \forall out\ :\ O \Rightarrow (f\ out = out')) \rightarrow S\ out) \\ \qquad \triangleleft (\lambda out'\ f\ i\ in \Rightarrow \exists out\ :\ O;\ p\ :\ (f\ out = out') \Rightarrow P\ out\ (f\ out\ p)\ i\ in) \\ \qquad \underline{\text{where}}\ \ (S \triangleleft P) = \mathbf{ICont}\ T\end{array}$$

At a 'Π^{∞}' we have to give a shape for the sub container at every possible new output index, the positions then have to carry at which particular index they exist.

$$\begin{array}{l}\mathbf{ICont}\ (\text{`}\mu\text{'}\ F)\ \Rightarrow\ (\lambda out \Rightarrow \mathsf{W}(\lambda s\ :\ S\ out \Rightarrow P\ out\ s\ \mathsf{fz}\ out) \triangleleft \mathsf{Paths}) \\ \qquad \underline{\text{where}}\ \ (S \triangleleft P) = \mathbf{ICont}\ F \\ \qquad\qquad \underline{\text{data}}\ \ \dfrac{out\ :\ O \quad s\ :\ \mathsf{W}(\lambda w\ :\ S\ out \Rightarrow P\ out\ s\ \mathsf{fz}\ out) \quad i\ :\ \mathsf{Fin}\ n \quad in\ :\ \vec{I}!!i}{\mathsf{Paths}\ out\ w\ i\ in\ :\ \star} \\ \qquad\qquad\qquad \underline{\text{where}}\ \ \dfrac{p\ :\ P\ out\ s\ (\mathsf{fs}\ i)\ in}{\mathsf{here}\ p\ :\ \mathsf{Paths}\ out\ (\mathsf{sup}\ s\ f)\ i\ in} \\ \qquad\qquad\qquad\qquad \dfrac{q\ :\ P\ out\ s\ \mathsf{fz}\ out \quad r\ :\ \mathsf{Paths}\ out\ (f\ q)\ i\ in}{\mathsf{there}\ q\ r\ :\ \mathsf{Paths}\ out\ (\mathsf{sup}\ s\ f)\ i\ in}\end{array}$$

The shapes of the fixed point case are given a trees that branch over the fz positions in the IC formed from F. We then define inductively the paths through these trees to fs positions, if we choose a recursive position we descend further into the tree.

The extension of an indexed container $(S \triangleleft P)\ :\ \mathsf{IC}\ \vec{I}\ \ O$ in this form is a functor $(\forall i\ :\ \mathsf{Fin}\ n \Rightarrow (\vec{I}!!i) \rightarrow \star) \rightarrow (O \rightarrow \star)$ and is given by:

$$[\![S \triangleleft P]\!]^{\mathbf{E}}\ \vec{X}\ o \Rightarrow \exists s\ :\ S\ o \Rightarrow \forall i\ :\ \mathsf{Fin}\ n \Rightarrow P\ o\ s\ i\ (\vec{I}!!i) \rightarrow \vec{X}!!i$$

There is a natural isomorphism between the telescope semantics of a family T and the extension of its translation to a container guided by the codes $\mathsf{SPF}\ \vec{I}\ O$ that is we can define two families of mutually inverse functions:

$$\underline{\text{let}}\ \frac{T : \mathsf{SPF}\ \vec{I}\ O \quad v : [\![T]\!]^{\mathsf{T}}\ \vec{T}\ o}{\phi\ T\ v : [\![\mathbf{ICont}\ T]\!]^{\mathbf{E}}(\mathbf{ICTel}\ \vec{T})\ o} \quad \ldots$$

$$\underline{\text{let}}\ \frac{T : \mathsf{SPF}\ \vec{I}\ O \quad v : [\![\mathbf{ICont}\ T]\!]^{\mathbf{E}}(\mathbf{ICTel}\ \vec{T})\ o}{\phi^{-1}\ T\ v : [\![T]\!]^{\mathsf{T}}\ \vec{T}\ o} \quad \ldots$$

Where $\mathbf{ICTel}$ iterates $\mathbf{ICont}$ over the telescope.

Since we can embed the regular families into the strictly positive we know we can play this same game with the RF types, indeed if we change the infinite 'Π^{∞}' case above for:

$$\begin{aligned}\mathbf{ICont}\ (\text{`}\Pi^{<\omega}\text{'}n\ T)\ \Rightarrow\ (&\lambda out\ :\ O \Rightarrow \forall j\ :\ \mathsf{Fin}\ n \Rightarrow S\ (out;\ j)\\ &\lhd(\lambda out\ f\ i\ in \Rightarrow \exists j\ :\ \mathsf{Fin}\ n \Rightarrow P\ (out;\ j)\ (f\ j)\ i\ in)\\ &\underline{\text{where}}\ \ (S \lhd P) = \mathbf{ICont}\ T\end{aligned}$$

we can then construct a similar isomorphism for RFs.

5. GENERIC PROGRAMS

5.1. Equality of RFs

Given a any regular type we can define a generic equality which is structural on the elements of its telescope semantics:

$$\underline{\text{let}}\ \frac{T : \mathsf{RF}\ \vec{I}\ O \quad a : [\![T]\!]^{\mathsf{T}}\ \vec{T}\ ob \quad b : [\![T]\!]^{\mathsf{T}}\ \vec{T}\ ob}{\mathbf{gEq}\ T\ a\ b : \mathsf{Bool}}$$

$$\begin{array}{lllll}
\mathbf{gEq}\ T\ a\ b \Leftarrow \underline{\text{rec}}\ a & & & & \\
\quad \mathbf{gEq}\ \text{`Z'}_{(\vec{T} : T)} & (\mathsf{top}\ a) & (\mathsf{top}\ b) & & \Rightarrow \mathbf{gEq}\ T\ a\ b \\
\quad \mathbf{gEq}\ (\text{`wk'}\ T) & (\mathsf{pop}\ a) & (\mathsf{pop}\ b) & & \Rightarrow \mathbf{gEq}\ T\ a\ b \\
\quad \mathbf{gEq}\ (\text{`Tag'}\ T) & (\mathsf{tag}_{ta}\ a) & (\mathsf{tag}_{tb}\ b) & \| \ ta{==}tb & \\
 & & & \mathsf{yes}\ (\mathsf{refl}\ t) & \Rightarrow \mathbf{gEq}\ (T\ t)\ a\ b \\
 & & & \mathsf{no}\ _ & \Rightarrow \mathsf{false} \\
\quad \mathbf{gEq} & \text{`1'} & \mathsf{void} & \mathsf{void} & \Rightarrow \mathsf{true} \\
\quad \mathbf{gEq} & (\text{`}\Sigma\text{'}\ f\ T) & (\sigma_{oa}\ a) & (\sigma_{ob}\ b) & \Rightarrow \mathbf{gEq}\ T\ a\ b \\
\quad \mathbf{gEq} & (\text{`}\Delta\text{'}\ f\ T) & (\delta\ a) & (\delta\ b) & \Rightarrow \mathbf{gEq}\ T\ a\ b \\
\quad \mathbf{gEq} & (\text{`}\Pi^{<\omega}\text{'}\ n\ T) & (\pi^{<\omega}\vec{a}) & (\pi^{<\omega}\vec{b}) & \Rightarrow \wedge(\forall i\ :\ \mathsf{Fin}\ n \Rightarrow \mathbf{gEq}\ T\ \vec{a}\ \vec{b}) \\
\quad \mathbf{gEq} & (\text{`}\mu\text{'}\ F) & (\mathsf{in}\ a) & (\mathsf{in}\ b) & \Rightarrow \mathbf{gEq}\ F\ a\ b
\end{array}$$

Notice that we can decide the equality of values in a purely syntactic manner, in fact this test equates values at different output indexes as long as the syntax is the same (so for instance $\mathsf{fz} : \mathsf{Fin}\ n = \mathsf{fz} : \mathsf{Fin}\ (1{+}\ n)$). In practice it would be better to restrict ourselves only to comparing things for equality at the same index.

As in our work on the Regular Tree Types, it is possible to show that this equality is decidable, that is we can return evidence for the equality or inequality. This is something that is especially useful in dependently typed programming.

5.2. Modalities, map and find

In our final example we give definitions for the modalities $\Box$ and $\Diamond$. Informally the modality $\Box$ is, for a given family $F : \star \to \star$ and predicate $P : A \to \star$ a new type $\Box F\ P : F\ A \to \star$ that says that the predicate P 'holds' (is inhabited) for each $a : A$ in an $F\ A$.

$$\underline{\text{data}}\ \frac{P\ :\ A \to \star \quad as\ :\ \mathsf{List}\ A}{\Box\mathsf{List}\ P\ as\ :\ \star}\ \underline{\text{where}}\ \frac{}{\varepsilon\ :\ \Box\mathsf{List}\ P\ A\ \varepsilon}\quad \frac{p\ :\ P\ a \quad ps\ :\ \Box\mathsf{List}\ P\ as}{p :: ps\ :\ \Box\mathsf{List}\ P\ (a :: as)}$$

The dual of $\Box$, the modality $\Diamond$ gives a type which describes the predicate P holding *somewhere* in the structure, so again for lists:

$$\underline{\text{data}}\ \frac{P\ :\ A \to \star \quad as\ :\ \mathsf{List}\ A}{\Diamond\mathsf{List}\ P\ as\ :\ \star}\ \underline{\text{where}}\ \frac{p\ :\ P\ a}{\mathsf{here}\ p\ :\ \Diamond\mathsf{List}\ P\ A\ (a :: as)}\quad \frac{ps\ :\ \Diamond\mathsf{List}\ P\ as}{\mathsf{there}\ ps\ :\ \Diamond\mathsf{List}\ P\ (a :: as)}$$

It seems that the idea of both $\Box$ and $\Diamond$ fit nicely with our abstraction of datatypes as containers and, indeed, we can give *generically* the types $\Box F\ P$ and $\Diamond F\ P$ for any $F\ :\ \mathsf{SPF}\ \vec{I}\ O$. Moreover, we can define a notion of generic dependent map using the generic $\Box$ and find that $\mathbf{map} has adual for$ $\Diamond$, which we call $\mathbf{find}$.

Firstly we define $\Box$:

$$\underline{\text{let}}\ \frac{F\ :\ \mathsf{RF}\ (\vec{I} :: \mathsf{One})\ O \quad P\ :\ [\![A]\!]^{\mathsf{T}}\ \vec{T}\ () \to \mathsf{RF}\ \vec{I}\ \mathsf{One} \quad v\ :\ [\![F]\!]^{\mathsf{T}}(\vec{T} :: A)\ o}{\Box F\ P\ v\ :\ \mathsf{RF}\ \vec{I}\ \mathsf{One}}$$

$$\begin{array}{lllll}
\Box T\ P\ a & \Leftarrow & \underline{\text{rec}}\ a & & \\
\quad \Box\text{`Z'} & P\ (\mathsf{top}\ a) & \Rightarrow & P\ a & \\
\quad \Box(\text{`wk'}\ T) & P\ (\mathsf{pop}\ v) & \Rightarrow & \text{`1'} & \\
\quad \Box(\text{`Tag'}\ T) & P\ (\mathsf{tag}_t\ v) & \Rightarrow & \Box(T\ t)\ P\ v & \\
\quad \Box\text{`1'} & P\ \ \mathsf{void} & \Rightarrow & \text{`1'} & \\
\quad \Box(\text{`}\Sigma\text{'}f\ \ T) & P\ (\sigma_o\ \ v) & \Rightarrow & \Box T\ P\ v & \\
\quad \Box(\text{`}\Delta\text{'}f\ \ T) & P\ \ (\delta\ \ v) & \Rightarrow & \Box T\ P\ v & \\
\quad \Box(\text{`}\Pi^{<\omega}\text{'}\ n\ \ T) & P\ (\mathsf{tag}\ \vec{v}) & \Rightarrow & \text{`}\Pi^{<\omega}\text{'}n\ (\mathsf{Tag}\ (\forall i \Rightarrow \Box T\ P\ (\vec{v}\ i))) & \\
\quad \Box(\text{`}\mu\text{'}\ \ F) & P\ \ (\mathsf{in}\ v) & \Rightarrow & \text{`}\mu\text{'}\ \ (\Box_1 F\ P\ v) &
\end{array}$$

You'll notice the $\text{`}\mu\text{'}$ moves the target under another $(::)$ constructor so we cannot appeal to a simple recursive call. In fact we'd have to define a more general $\Box_i$ where i is the index of the target type in the context. We then have that:

$$\begin{array}{lll}
\Box_0\text{`Z'} & P\ (\mathsf{top}\ a) \Rightarrow & P\ a \\
\Box_0(\text{`wk'}\ T) & P\ (\mathsf{pop}\ v) \Rightarrow & \text{`1'} \\
\Box_{(1+\ n)}\text{`Z'} & P\ (\mathsf{top}\ v) \Rightarrow & \text{`Z'} \\
\Box_{(1+\ n)}(\text{`wk'}\ T) & P\ (\mathsf{pop}\ v) \Rightarrow & \text{`wk'}\ \ (\Box_n T\ P\ v) \\
 & \vdots \quad\quad \vdots & \\
\Box_n(\text{`}\mu\text{'}\ \ F) & P\ \ (\mathsf{in}\ v) \Rightarrow & \text{`}\mu\text{'}\ \ (\Box_{(1+\ n)} F\ P\ v)
\end{array}$$

and $\Box = \Box_0$.

The definition of $\Diamond$ follows much the same pattern, but we replace the following rules:

$$\begin{array}{lll}
 & \vdots \quad\quad \vdots & \\
\Diamond_0(\text{`wk'}\ T) & P\ (\mathsf{pop}\ v) \Rightarrow & \text{`0'} \\
 & \vdots \quad\quad \vdots & \\
\Diamond_n\text{`1'} & P\ \ \mathsf{void} \Rightarrow & \text{`0'} \\
 & \vdots \quad\quad \vdots & \\
\Diamond_n(\text{`}\Pi^{<\omega}\text{'}\ n\ \ T) & P\ (\mathsf{tag}\ \vec{v}) \Rightarrow & \text{`}\Sigma\text{'}\mathbf{fst}\ (\mathsf{Tag}\ (\forall i \Rightarrow \Diamond_n T\ P\ (\vec{v}\ i)))
\end{array}$$

That is we no longer succeed by not finding a variable of the right type and when confronted by a set of possibilities, we need only pick one.

What about 'map' and 'find'? Informally, given $f : (\forall a : A \Rightarrow B\ a)$ we can produce a value of $\Box F\ B$ for any $F\ A$; we use the f as evidence that B is inhabited for any A. In the dual **find** case we are given an element of $\Diamond F\ B$ for some A from which we can produce a witness that $B\ a$ is inhabited for some value $a : A$.

$$\underline{\text{let}}\ \frac{f : (\forall a : [\![A]\!]^{\mathsf{T}}\ \vec{T}\ () \Rightarrow [\![B\ a]\!]^{\mathsf{T}}\ \vec{T}\ ())}{\mathbf{map}\ f : (\forall v : [\![F]\!]^{\mathsf{T}}(\vec{T} :: T)\ o \Rightarrow \Box F\ B\ v)}\ \ldots$$

$$\underline{\text{let}}\ \frac{d : (\exists v : [\![F]\!]^{\mathsf{T}}(\vec{T} :: T)\ o \Rightarrow \Diamond F\ B\ v)}{\mathbf{find}\ d : (\exists a : [\![A]\!]^{\mathsf{T}}\ \vec{T}\ () \Rightarrow [\![B\ a]\!]^{\mathsf{T}}\ \vec{T}\ ())}\ \ldots$$

These definitions follow exactly the same recursive pattern as the definitions of $\Box$ and $\Diamond$ themselves.

6. FUTURE WORK AND CONCLUSIONS

We have tied the knot by presenting a universe construction which is powerful enough to encode all inductive types needed in Epigram, including the construction itself. While encoding types by hand is a rather cumbersome process, we can translate the high level Epigram syntax mechanically into the SPFs. We plan to integrate the universe directly into Epigram giving the programmer direct access to the internal representations of types for generic programming as part of the system. This approach also has the benefit that it allows a more flexible and extensible positivity test as we have demonstrated in the example of n-branching trees. Exploiting Observational Type Theory [4] we are also planning to include coinductive definitions in the universe.

It turns out that the Epigram 'gadgets' that build the structural recursion, and case analysis principals ($\Leftarrow$ rec and $\Leftarrow$ case) for Epigram datatypes are generic programs in this universe. Expressing them in the language may well help us on the road to building Epigram in Epigram.

The move from strictly positive to regular families is but one example for a hierarchy of universes important for generic programming. The trade-off is clear — the further up we move the more generality we gain, the further down we go the more generic functions are definable. It is the subject of future work to see how we can give the programmer the opportunity to move along this axis freely, seeking the optimal compromise for a certain collection of generic functions.

REFERENCES

[1] Michael Abott, Thorsten Altenkirch, and Neil Ghani. Containers - constructing strictly positive types. *Theoretical Computer Science*, 342:3–27, September 2005. Applied Semantics: Selected Topics.

[2] Michael Abott, Thorsten Altenkirch, Neil Ghani, and Conor McBride. ∂ for data. *Fundamentae Informatica*, 65(1,2):1 – 28, March 2005. Special Issue on Typed Lambda Calculi and Applications 2003.

[3] Thorsten Altenkirch, Neil Ghani, Peter Hancock, Conor McBride, and Peter Morris. Indexed containers. Manuscript, available online, February 2006.

[4] Thorsten Altenkirch and Conor McBride. Towards observational type theory. Manuscript, available online, February 2006.

[5] Thorsten Altenkirch, Conor McBride, and James McKinna. Why dependent types matter. Manuscript, available online, April 2005.

[6] Marcin Benke, Peter Dybjer, and Patrik Jansson. Universes for generic programs and proofs in dependent type theory. *Nordic Journal of Computing*, 10:265–269, 2003.

[7] Peter Dybjer and Anton Setzer. Indexed induction-recursion. In Reinhard Kahle, Peter Schroeder-Heister, and Robert F. Stärk, editors, *Proof Theory in Computer Science*, volume 2183 of *Lecture Notes in Computer Science*, pages 93–113. Springer, 2001.

[8] Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis·Napoli, 1984.

[9] Conor McBride. Epigram, 2004. `http://www.e-pig.org/`.

[10] Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1), 2004.

[11] Peter Morris, Thorsten Altenkirch, and Conor McBride. Exploring the regular tree types. In Christine Paulin-Mohring Jean-Christophe Filliatre and Benjamin Werner, editors, *Types for Proofs and Programs (TYPES 2004)*, Lecture Notes in Computer Science, 2006.

[12] Bengt Nordström, Kent Petersson, and Jan Smith. *Programming in Martin-Löf's type theory: an introduction*. Oxford University Press, 1990.