Generic Programming for Dependent Types

Constructing Strictly Positive Families

Peter Morris Thorsten Altenkirch

University of Nottingham, UK

{pwm,txa}@cs.nott.ac.uk

Abstract

We begin by revisiting the idea of using a universe of types to write generic programs in a dependently typed setting by constructing a universe for Strictly Positive Types (SPTs). Here we extend this construction to cover dependent types, i.e. Strictly Positive Families (SPFs), thereby fixing a gap left open in previous work. Using the approach presented here we are able to represent all of Epigram's datatypes within Epigram including the universe of datatypes itself.

Keywords Data type generic programming, dependently typed programming, Epigram

1. Introduction

In a dependently typed language like Epigram [8, 7, 3] generic programming is normal programing. This is achieved by defining a universe [6, 10] 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 [9] 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.

In related work, [5] 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 [4].

1.1 Programming in Epigram

Epigram is an dependently typed functional language with an interactive environment for developing programs with the aid of the type checker.

All Epigram programs (currently) are total to ensure that type checking is decidable. We ensure this by only allowing structural recursion. Programs are presented as decision trees, representing the structure of the analysis of the problem being solved. Each node is presented with the information available as a pattern on the left hand side, right hand sides can either be:

$\Rightarrow t$ the function *returns* t, an expression of the appropriate type, constructed over the pattern variables on the left;

$\Leftarrow e$ the function's analysis is refined *by* e, an *eliminator* expression, or 'gadget', characterizing some scheme of case analysis or recursion, giving rise to a bunch of sub nodes with more informative left-hand sides;

$|| w$ the sub nodes' left-hand sides are to be extended *with* the value of w, some intermediate computation, in an extra column: this may then be analysed in addition to the function's original arguments.

In this paper we will need only two 'by' gadgets, rec which constructs the structural recursive calls available to the programmer, and case which applies the appropriate derived case analysis principle and introduces a set of more informative patterns in the sub-nodes. We will use the convention that we suppress the use of case when its presence is inferable from the presence of constructors in the patterns. We will always be explicit about which input we are being structurally recursive on.

Epigram's data types are presented by declaring their formation rules and constructors in natural deduction style as are the types of functions. In these rules arguments whose types are inferable can be omitted for brevity.

As a warm up to Epigram's syntax we'll start with a simple datatype, the natural numbers and define addition:

$$\underline{\text{data}}\ \frac{}{\mathsf{Nat} : \star}\ \underline{\text{where}}\ \frac{}{0 : \mathsf{Nat}}\ \frac{n : \mathsf{Nat}}{1{+}\ n : \star}$$

$$\underline{\text{let}}\ \frac{m, n : \mathsf{Nat}}{\mathbf{plus}\ m\ n : \mathsf{Nat}}$$

$$\begin{array}{l}\mathbf{plus}\ m\ n \Leftarrow \underline{\text{rec}}\ m\\ \quad \mathbf{plus}\quad 0 \quad n \Rightarrow n\\ \quad \mathbf{plus}\ (1{+}\ m)\ n \Rightarrow 1{+}\ (\mathbf{plus}\ m\ n)\end{array}$$

We can then define types which are dependent on the natural numbers, the finite types and vectors (lists of a given length), we can then define safe projection from using the finite types to ensure there are only as many indexes as elements in the array.

$$\underline{\text{data}}\ \frac{n : \mathsf{Nat}}{\mathsf{Fin}\ n : \star}\ \underline{\text{where}}\ \frac{}{0 : \mathsf{Fin}\ (1{+}\ n)}\quad \frac{i : \mathsf{Fin}\ n}{1{+}\ i : \mathsf{Fin}\ (1{+}\ n)}$$

$$\underline{\text{data}}\ \frac{A : \star \quad n : \mathsf{Nat}}{\mathsf{Vec}\ n\ A : \star}\ \underline{\text{where}}\ \frac{}{\varepsilon : \mathsf{Vec}\ 0\ A}\quad \frac{a : A \quad as : \mathsf{Vec}\ n\ A}{a{:}as : \mathsf{Vec}\ (1{+}\ n)\ A}$$

$$\underline{\text{let}}\ \frac{as : \mathsf{Vec}\ n\ A \quad i : \mathsf{Fin}\ n}{\mathbf{proj}\ as\ i : A}$$

$$\begin{array}{l}\mathbf{proj}\ as\ i \Leftarrow \underline{\text{rec}}\ i\\ \quad \mathbf{proj}\ (a{:}as)\ \ 0 \Rightarrow a\\ \quad \mathbf{proj}\ (a{:}as)\ (1{+}\ i) \Rightarrow \mathbf{proj}\ as\ i\end{array}$$

Notice that the nil case doesn't appear since Fin 0 is uninhabited.

We will be returning to each of the above datatypes to encode them into our universe, along with another nice example of Epigram, the scoped lambda terms:

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

2. Strictly Positive Types

Strictly positive types can be constructed using polynomial expressions $(0, 1, +, \times)$, initial algebras (μ) and exponentiation by a constant $K \to -$. Examples are natural numbers $\mathrm{Nat} = \mu X.1 + X$, lists $\mathrm{List}\ A = \mu X.1 + A \times X$, rose trees $\mathrm{RT}\ A = \mu Y.\mathrm{List}\ (A \times Y) = \mu Y.\mu X.1 + A \times Y \times X$ and ordinal notations $\mathrm{Ord} = \mu X.1 + X + \mathrm{Nat} \to X$. The first three examples, which don't use exponentiation are regular tree types which are a proper subset of strictly positive types.

Strictly positive types with n free type variables are represented by the Epigram type SPT n, see figure 1. We use de Bruijn variables, where 'Z' stands for the last variable with index 0 and previous variables can be accessed using weakening 'wk'. Note that the fix-point constructor 'μ' reduces the number of free variables by 1 because the last variable has been bound. We introduce the type of nested type substitutions, or telescopes, of length n, Tel n. Given a type T : SPT n and a matching telescope $\vec{T}$: Tel we define the type of elements El $\vec{T}$ T as an inductive family.

We can now encode the examples mentioned above (Nat, List, RT, Ord):

$$\begin{array}{l}\underline{\text{let}}\ \text{'}\mathbf{Nat}\text{'} : \mathsf{SPT}\ 0\\ \quad \text{'}\mathbf{Nat}\text{'} \Rightarrow \text{'}\mu\text{'}\ (\text{'1'}\ \text{'+'}\ \text{'Z'})\end{array}$$

$$\begin{array}{l}\underline{\text{let}}\ \text{'}\mathbf{List}\text{'} : \mathsf{SPT}\ 1\\ \quad \text{'}\mathbf{List}\text{'} \Rightarrow \text{'}\mu\text{'}\ (\text{'1'}\ \text{'+'}\ ((\text{'wk'}\ \text{'Z'})\ \text{'}\times\text{'}\ \text{'Z'})\end{array}$$

$$\begin{array}{l}\underline{\text{let}}\ \text{'}\mathbf{RT}\text{'} : \mathsf{SPT}\ 1\\ \quad \text{'}\mathbf{RT}\text{'} \Rightarrow \text{'}\mu\text{'}\ ((\text{'wk'}\ \text{'Z'})\ \text{'+'}\ (\text{'}\mu\text{'}\ ((\text{'wk'}\ \text{'Z'})\ \text{'}\times\text{'}\ \text{'Z'})))\end{array}$$

$$\begin{array}{l}\underline{\text{let}}\ \text{'}\mathbf{Ord}\text{'} : \mathsf{SPT}\ 0\\ \quad \text{'}\mathbf{Ord}\text{'} \Rightarrow \text{'}\mu\text{'}\ (\text{'1'}\ \text{'+'}\ (\text{'Z'}\ \text{'+'}\ (\mathsf{Nat}\ \text{'}\to\text{'}\ \text{'Z'}))\end{array}$$

In the system we can build 'constructors' which construct elements of the interpretation of a code, for example with '**Nat**':

$$\begin{array}{l}\underline{\text{let}}\ \text{'}\mathbf{0}\text{'} : \mathsf{El}\ \varepsilon\ \text{'}\mathbf{Nat}\text{'}\\ \quad \text{'}\mathbf{0}\text{'} \Rightarrow \mathsf{in}\ (\mathsf{inl}\ \mathsf{void})\\ \underline{\text{let}}\ \text{'}\mathbf{1+}\text{'} : \mathsf{El}\ \varepsilon\ \text{'}\mathbf{Nat}\text{'} \to \mathsf{El}\ \varepsilon\ \text{'}\mathbf{Nat}\text{'}\\ \quad \text{'}\mathbf{1+}\text{'} \Rightarrow (\lambda n \Rightarrow \mathsf{in}\ (\mathsf{inr}\ (\mathsf{top}\ n)))\end{array}$$

2.1 Generic Map

As our first example of a generic program we shall present generic map. We shall define this by first considering morphisms between telescopes. Our base case is going to be the identity morphism between a telescope and itself, and we can obviously extend a morphism with a function. To ensure our map is structurally recursive (and total) we also need to be able to go under local bindings without extending the, morphism ϕ by the non structural recursive call **gMap** ϕ. So we add a third constructor to morphisms mMap which we will use to denote an extension under a binder.

$$\underline{\text{data}}\ \frac{\vec{S}, \vec{T} : \mathsf{Tel}\ n}{\mathsf{Morph}\ \vec{S}\ \vec{T} : \star}\ \underline{\text{where}}$$

$$\frac{}{\mathsf{mId} : \mathsf{Morph}\ \vec{S}\ \vec{S}}$$

$$\frac{\phi : \mathsf{Morph}\ \vec{S}\ \vec{T} \quad f : \mathsf{El}\ \vec{S}\ S \to \mathsf{El}\ \vec{T}\ T}{\mathsf{mFun}\ \phi\ f : \mathsf{Morph}\ (\vec{S}{:}S)\ (\vec{T}{:}T)}$$

$$\frac{\phi : \mathsf{Morph}\ \vec{S}\ \vec{T}}{\mathsf{mMap}\ \phi : \mathsf{Morph}\ (\vec{S}{:}T)\ (\vec{T}{:}T)}$$

Most of the definition for **gMap** is structural, we extend uniformly under 'μ' and when we see a mMap extension at the top variable we reconstruct the (now obviously structural) call we'd expect.

$$\underline{\text{let}}\ \frac{\phi : \mathsf{Morph}\ \vec{S}\ \vec{T} \quad x : \mathsf{El}\ \vec{S}\ T}{\mathbf{gMap}\ \phi\ x : \mathsf{El}\ \vec{T}\ T}$$

$$\begin{array}{llll}\mathbf{gMap} & \phi & x & \Leftarrow \underline{\text{rec}}\ x\\ \mathbf{gMap} & \mathsf{mId} & (\mathsf{top}\ x) & \Rightarrow \mathsf{top}\ x\\ \mathbf{gMap} & (\mathsf{mFun}\ \phi\ f) & (\mathsf{top}\ x) & \Rightarrow \mathsf{top}\ (f\ x)\\ \mathbf{gMap} & (\mathsf{mMap}\ \phi) & (\mathsf{top}\ x) & \Rightarrow \mathsf{top}\ (\mathbf{gMap}\ \phi\ x)\\ \mathbf{gMap} & \mathsf{mId} & (\mathsf{pop}\ x) & \Rightarrow \mathsf{pop}\ x\\ \mathbf{gMap} & (\mathsf{mFun}\ \phi\ f) & (\mathsf{pop}\ x) & \Rightarrow \mathsf{pop}\ (\mathbf{gMap}\ \phi\ x)\\ \mathbf{gMap} & (\mathsf{mMap}\ \phi) & (\mathsf{pop}\ x) & \Rightarrow \mathsf{pop}\ (\mathbf{gMap}\ \phi\ x)\\ \mathbf{gMap} & \phi & \mathsf{void} & \Rightarrow \mathsf{void}\\ \mathbf{gMap} & \phi & (\mathsf{inl}\ x) & \Rightarrow \mathsf{inl}\ (\mathbf{gMap}\ \phi\ x)\\ \mathbf{gMap} & \phi & (\mathsf{inr}\ x) & \Rightarrow \mathsf{inr}\ (\mathbf{gMap}\ \phi\ x)\\ \mathbf{gMap} & \phi & (\mathsf{pair}\ x\ y) & \\ & & \Rightarrow \mathsf{pair}\ (\mathbf{gMap}\ \phi\ x)\ (\mathbf{gMap}\ \phi\ y) & \\ \mathbf{gMap} & \phi & (\mathsf{fun}\ f) & \\ & & \Rightarrow \mathsf{fun}\ (\lambda k \Rightarrow \mathbf{gMap}\ \phi\ (f\ k)) & \\ \mathbf{gMap} & \phi & (\mathsf{in}\ x) & \\ & & \Rightarrow \mathsf{in}\ (\mathbf{gMap}\ (\mathsf{mMap}\ \phi)\ x) & \end{array}$$

In our work on the regular tree types [9], we present a number of other algorithms in this style for the regular tree types, including a decidable equality. Types in this universe do not have such an equality, there is no such function for the ordinals for instance. It is clear that the larger the universe of types the fewer generic operations we may define. In a system of generic programming it is conceivable that we would need a number of successively large universes to cope with this trade off.

$$\underline{\text{data}}\ \frac{n : \mathsf{Nat}}{\mathsf{SPT}\ n : \star}\ \underline{\text{where}}$$

$$\frac{}{\text{`Z'} : \mathsf{SPT}\ (1{+}\ n)} \quad \frac{T : \mathsf{SPT}\ n}{\text{`wk'}\ T : \mathsf{SPT}\ (1{+}\ n)} \quad \frac{}{\text{`0'} : \mathsf{SPT}\ n} \quad \frac{}{\text{`1'} : \mathsf{SPT}\ n}$$

$$\frac{S, T : \mathsf{SPT}\ n}{S\ \text{`}{+}\text{'}\ T : \mathsf{SPT}\ n} \quad \frac{S, T : \mathsf{SPT}\ n}{S\ \text{`}{\times}\text{'}\ T : \mathsf{SPT}\ n} \quad \frac{K : \star \quad T : \mathsf{SPT}\ n}{K\ \text{`}{\to}\text{'}\ T : \mathsf{SPT}\ n} \quad \frac{F : \mathsf{SPT}\ (1{+}\ n)}{\text{`}\mu\text{'}\ F : \mathsf{SPT}\ n}$$

$$\underline{\text{data}}\ \frac{n : \mathsf{Nat}}{\mathsf{Tel}\ n : \star} \quad \frac{}{\varepsilon : \mathsf{Tel}\ 0} \quad \frac{\vec{T} : \mathsf{Tel}\ n \quad T : \mathsf{SPT}\ n}{\vec{T} \text{:-} T : \mathsf{Tel}\ (1{+}\ n)}$$

$$\underline{\text{data}}\ \frac{\vec{T} : \mathsf{Tel}\ n \quad T : \mathsf{SPT}\ n}{\mathsf{El}\ \vec{T}\ T : \star}\ \underline{\text{where}}$$

$$\frac{e : \mathsf{El}\ \vec{T}\ T}{\mathsf{top}\ e : \mathsf{El}\ (\vec{T} \text{:-} T)\ \text{`Z'}} \quad \frac{e : \mathsf{El}\ \vec{T}\ T}{\mathsf{pop}\ e : \mathsf{El}\ (\vec{T} \text{:-} S)\ (\text{`wk'}\ T)} \quad \frac{}{\mathsf{void} : \mathsf{El}\ \vec{T}\ \text{`1'}} \quad \frac{f : K \to \mathsf{El}\ \vec{T}\ T}{\mathsf{fun}\ f : \mathsf{El}\ \vec{T}\ (K\ \text{`}{\to}\text{'}\ T)}$$

$$\frac{s : \mathsf{El}\ \vec{T}\ S}{\mathsf{inl}\ s : \mathsf{El}\ \vec{T}\ (S\ \text{`}{+}\text{'}\ T)} \quad \frac{t : \mathsf{El}\ \vec{T}\ T}{\mathsf{inr}\ t : \mathsf{El}\ \vec{T}\ (S\ \text{`}{+}\text{'}\ T)} \quad \frac{s : \mathsf{El}\ \vec{T}\ S \quad t : \mathsf{El}\ \vec{T}\ T}{\mathsf{pair}\ s\ t : \mathsf{El}\ \vec{T}\ (S\ \text{`}{\times}\text{'}\ T)} \quad \frac{e : \mathsf{El}\ (\vec{T} \text{:-} \text{`}\mu\text{'}\ F)\ F}{\mathsf{in}\ : \mathsf{El}\ \vec{T}\ (\text{`}\mu\text{'}\ F)}$$

Figure 1. The SPT Universe

3. Strictly Positive Families

There is an obvious imbalance in the above construction, we use the dependent types of Epigram to model a class of simple types for generic programming. If this technique is ever to prove useful then it has to be able to include as much of the power of the host language's type system as possible.

We have been using strictly positive families (presented in Epigram's natural deduction style) throughout this paper, but to illustrate the universe below we will return to the examples from the introduction, the finite types Fin, vectors Vec and scoped lambda terms Lam.

To tell this story we consider the positive ways to build a family of types in $O \to \star$, for the moment we'll call these codes Fam. We call O the output index type. In all of the above examples $O \equiv \mathsf{Nat}$, though it can be *any* type. The simplest such families are constant and ignore the indexing information, we have two such types, '0' and '1' which are have zero or one elements (respectively) at all indices. We introduce the codes for this universe, and the interpretation $[\![-]\!] : \mathsf{Fam}\ O \to O \to \star$ simultaneously.

$$\frac{}{\text{`0'}, \text{`1'} : \mathsf{Fam}\ O} \qquad \frac{}{\mathsf{void} : [\![\text{`1'}]\!]\, o}$$

Clearly another possibility is to to substitute for the given index, that is given a $O \to \star$ and a function in $O' \to O$ we can create a new family in $O' \to \star$. Categorically this relates to a pullback, and we will refer to it as re-indexing of a family.

$$\frac{f : O' \to O \quad T : \mathsf{Fam}\ O}{\text{`}\Delta\text{'}_O f\ T : \mathsf{Fam}\ O'} \qquad \frac{v : [\![T]\!]\,(f\ o)}{\delta\ v : [\![\text{`}\Delta\text{'}_O f\ T]\!]\, o}$$

It is interesting now to consider what we can do to construct values if we have a function on output types that goes the other way. Given a $\mathsf{Fam}\ O$ and a function $O \to O'$ we must construct values at a given index $o' : O'$, to do this we consider only the values $o : O$ for which $f\ o = o'$ but do we consider all such values, or just pick one? The first option gives us dependent sum, the second dependent product:

$$\frac{f : O \to O' \quad T : \mathsf{Fam}\ O}{\text{`}\Sigma\text{'}_O f\ T, \text{`}\Pi\text{'}_O f\ T : \mathsf{Fam}\ O'}$$

$$\frac{v : \exists o : O \Rightarrow (f\ o = o') \times [\![T]\!]\, o}{\sigma_v\ : [\![\text{`}\Sigma\text{'}_O f\ T]\!]\, o'}$$

$$\frac{\vec{v} : \forall o : O \Rightarrow (f\ o = o') \to [\![T]\!]\, o}{\pi \vec{v} : [\![\text{`}\Pi\text{'}_O f\ T]\!]\, o'}$$

Categorically, 'Σ' and 'Π' are respectively the left and right adjoints of 'Δ'.

Although we called these sum and product there is only one type involved, in fact all of our constructs are linear in this way. 'Σ' and 'Π' work only on the index level so we must be able to pick between types based on the index we are given. There are a number of possibilities her but it seems sensible to restrict ourselves to finite collections of types so, using Fin:

$$\frac{f : \forall t : \mathsf{Fin}\ n \Rightarrow \mathsf{Fam}\ O}{\text{`}\mathsf{Tag}\text{'}f : \mathsf{Fam}\ (O \times \mathsf{Fin}\ n)} \qquad \frac{v : [\![f\ t]\!]\, o}{\mathsf{tag}_t v : [\![\text{`}\mathsf{Tag}\text{'}f]\!]\,(o; t)}$$

Finally we add type variables and the fixed point constructor. Our input types will now be dependent so we have to index our codes not only by the input index type but the set of input index types, Fam above becomes:

$$\underline{\text{data}}\ \frac{\vec{I} : \mathsf{Vec}\ \star\ n \quad O : \star}{\mathsf{SPF}\ \vec{I}\ O : \star}$$

For the interpretation we follow the SPT construction and introduce telescopes as closing substitutions, this time with indexing information:

$$\underline{\text{data}}\ \frac{\vec{I} : \mathsf{Vec}\ \star\ n}{\mathsf{dTel}\ \vec{I} : \star}\ \underline{\text{where}}$$

$$\frac{}{\varepsilon : \mathsf{dTel}\ \varepsilon} \quad \frac{\vec{T} : \mathsf{dTel}\ \vec{I} \quad T : \mathsf{SPF}\ \vec{I}\ I}{\vec{T} \text{:-} T : \mathsf{dTel}\ (\vec{I} \text{:-} I)}$$

and our interpretation now has this type:

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

the introduction rules for the codes given above can be uniformly modified to thread the context unchanged. The variable rules are as follows:

$$\frac{}{\text{`Z'} : \mathsf{SPF}\ (\vec{I} :\!: O)\ O} \qquad \frac{v : [\![T]\!]\ \vec{T}\ o}{\mathsf{top}\ v : [\![\text{`Z'}]\!]\ (\vec{T} :\!: T)\ o}$$

$$\frac{T : \mathsf{SPT}\ \vec{I}\ O}{\text{`wk'}\ T : \mathsf{SPF}\ (\vec{I} :\!: I)\ O} \qquad \frac{v : [\![T]\!]\ \vec{T}\ o}{\mathsf{pop}\ T : [\![\text{`wk'}\ \ T]\!]\ (\vec{T} :\!: S)\ o}$$

Finally the fixed point construct:

$$\frac{T : \mathsf{SPF}\ (\vec{I} :\!: O)\ O}{\text{`}\mu\text{'}\ T : \mathsf{SPF}\ \vec{I}\ O} \qquad \frac{v : [\![T]\!]\ (\vec{T} :\!: (\text{`}\mu\text{'}\ T))\ o}{\mathsf{in}\ v : [\![\text{`}\mu\text{'}\ T]\!]\ \vec{T}\ o}$$

The whole construction is given in figure 2.

We can now encode our examples into this universe, as an abbreviation we will first define disjoint union, Cartesian product and non-dependent arrow in a form similar to the SPT construction:

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

$$A\ \text{`+'}\ B \ \Rightarrow\ \text{`}\Sigma\text{'}\mathbf{fst}\left(\text{`Tag'}\left(\begin{array}{l} 0 \Rightarrow A \\ 1 \Rightarrow B \end{array}\right)\right)$$

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

$$\mathbf{inl}_{\vec{T}\ o}\ a \ \Rightarrow\ \sigma_{(o;0)}\ (\mathsf{tag}\ a)$$

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

$$\mathbf{inr}_{\vec{T}\ o}\ b \ \Rightarrow\ \sigma_{(o;1)}\ (\mathsf{tag}\ b)$$

$$A\ \text{`}\times\text{'}\ B \ \Rightarrow\ \text{`}\Pi\text{'}\mathbf{fst}\left(\text{`Tag'}\left(\begin{array}{l} 0 \Rightarrow A \\ 1 \Rightarrow B \end{array}\right)\right)$$

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

$$\mathbf{pair}\ a\ b \ \Rightarrow\ \pi\left(\begin{array}{l} (o;0)\ \mathsf{refl} \Rightarrow \mathsf{tag}\ a \\ (o;1)\ \mathsf{refl} \Rightarrow \mathsf{tag}\ b \end{array}\right)$$

$$\underline{\text{let}}\ \frac{K : \star \quad T : \mathsf{SPF}\ \vec{I}\ O}{K\ \text{`}\rightarrow\text{'}\ T : \mathsf{SPF}\ \vec{I}\ O}$$

$$K\ \text{`}\rightarrow\text{'}\ T \ \Rightarrow\ \text{`}\Pi\text{'}_{(O \times K)}\mathbf{fst}\ (\text{`}\Delta\text{'}\mathbf{fst}\ T)$$

$$\underline{\text{let}}\ \frac{\vec{v} : K \rightarrow [\![T]\!]\ \vec{T}\ T}{\mathbf{fun}\ \vec{v} : [\![K\ \text{`}\rightarrow\text{'}\ T]\!]\ \vec{T}\ o}$$

$$\mathbf{fun}\ \vec{v} \ \Rightarrow\ \pi\ (\delta\ \vec{v})$$

Notice how the 'Tag' interacts with the 'Σ' and 'Π'. These auxiliary constructions allow us to construct codes in a sums of products style as before, except with added indexing information, for instance:

$$\underline{\text{let}}\ \text{`}\mathbf{Fin}\text{'} : \mathsf{SPF}\ []\ \mathsf{Nat}$$
$$\text{`}\mathbf{Fin}\text{'} \ \Rightarrow\ \text{`}\mu\text{'}\ ((\text{`}\Sigma\text{'}1+\ \text{`1'})\ \text{`+'}\ (\text{`}\Sigma\text{'}1+\ \text{`Z'}))$$

Remember that both constructors for Fin target non zero numbers, the 'Σ'1+ construct ensures this, it forces us to choose an index n' that satisfies $n = 1+\ n'$ for the output index n since the equality type is uninhabited if $n = 0$ then so is the interpretation of this code.

In the 'Vec' example we assume that the element type is fully applied, and so is indexed only by the unit type. When we refer to its type variable, we use delta to force the index to $()$. In the nil case we use 'Σ' to ensure that the index is 0.

$$\underline{\text{let}}\ \text{`}\mathbf{Vec}\text{'} : \mathsf{SPF}\ [1]\ \mathsf{Nat}$$
$$\text{`}\mathbf{Vec}\text{'} \ \Rightarrow$$
$$\text{`}\mu\text{'}\left(\begin{array}{l} (\text{`}\Sigma\text{'}\ (\mathbf{const}\ 0)\ \text{`1'})\ \text{`+'} \\ \quad (\text{`}\Sigma\text{'}\ 1+\ (\text{`}\Delta\text{'}(\mathbf{const}\ ())\ (\text{`wk'}\ \text{`Z'}))\ \text{`}\times\text{'}\ \text{`Z'}) \end{array}\right)$$

As before we can create 'constructors' for these types to reflect the conventional ones. For instance for 'list':

$$\underline{\text{let}}\ \text{`}\varepsilon\text{'} : [\![\text{`}\mathbf{Vec}\text{'}]\!][A]\ 0$$
$$\text{`}\varepsilon\text{'} \ \Rightarrow\ \mathsf{in}\ (\mathbf{inl}\ (\sigma\_\ \mathsf{void}))$$

$$\underline{\text{let}}\ \frac{a : [\![A]\!][]\ ()\quad as : [\![\text{`}\mathbf{Vec}\text{'}]\!][A]\ n}{(a\ \text{`}:\!:\text{'}as) : [\![\text{`}\mathbf{Vec}\text{'}]\!][A]\ (1+\ n)}$$
$$(a\ \text{`}:\!:\text{'}as) \ \Rightarrow\ \mathsf{in}\ (\mathbf{inr}\ (\mathbf{pair}\ (\delta\ a)\ as))$$

As promised, we can also define the well scoped lambda terms:

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

3.1 Embedding SPT into SPF

It is clear that the SPT universe represents a subset of the the types in SPF indeed we have overloaded the names ('Z', '+', ..., 'μ') to suggest the relation. Will will now be more explicit about this.

We first give a function that transforms SPT codes into SPF codes, since SPFs have to be indexed by some type, we have no choice but to index the embedded codes by the unit type:

$$\underline{\text{let}}\ \frac{n : \mathsf{Nat} \quad a : A}{\mathbf{vec}\ a\ n : \mathsf{Vec}\ A\ n}$$

$$\begin{array}{l} \mathbf{vec}\ a\ n \Leftarrow \underline{\text{rec}}\ n \\ \quad \mathbf{vec}\ a \quad 0 \quad \Rightarrow \varepsilon \\ \quad \mathbf{vec}\ a\ (1+\ n) \Rightarrow a\text{`}:\!:\text{'}(\mathbf{vec}\ a\ n) \end{array}$$

$$\underline{\text{let}}\ \frac{T : \mathsf{SPT}\ n}{\mathbf{embspt}\ T : \mathsf{SPF}\ (\mathbf{vec}\ \mathbf{1}\ n)\ \mathbf{1}}$$

$$\begin{array}{l} \mathbf{embspt}\ T \Leftarrow \underline{\text{rec}}\ T \\ \quad \mathbf{embspt} \quad \text{`Z'} \quad \Rightarrow \text{`Z'} \\ \quad \mathbf{embspt} \quad (\text{`wk'}\ T) \Rightarrow \text{`wk'}\ (\mathbf{embspt}\ T) \\ \quad \mathbf{embspt} \quad \text{`0'} \quad \Rightarrow \text{`0'} \\ \quad \mathbf{embspt} \quad \text{`1'} \quad \Rightarrow \text{`1'} \\ \quad \mathbf{embspt} \quad (S\ \text{`+'}\ T) \Rightarrow (\mathbf{embspt}\ S)\ \text{`+'}\ (\mathbf{embspt}\ T) \\ \quad \mathbf{embspt} \quad (S\ \text{`}\times\text{'}\ T) \Rightarrow (\mathbf{embspt}\ S)\ \text{`}\times\text{'}\ (\mathbf{embspt}\ T) \\ \quad \mathbf{embspt} \quad (K\ \text{`}\rightarrow\text{'}\ T) \Rightarrow K\ \text{`}\rightarrow\text{'}\ (\mathbf{embspt}\ T) \\ \quad \mathbf{embspt} \quad (\text{`}\mu\text{'}\ F) \ \Rightarrow \text{`}\mu\text{'}\ (\mathbf{embspt}\ F) \end{array}$$

We map this embedding across the telescope:

$$\underline{\text{let}}\ \frac{\vec{T} : \mathsf{Tel}\ n}{\mathbf{embtel}\ \vec{T} : \mathsf{dTel}\ (\mathbf{vec}\ \mathbf{1}\ n)}$$

$$\begin{array}{l} \mathbf{embtel}_n\ \vec{T} \Leftarrow \underline{\text{rec}}\ n \\ \quad \mathbf{embtel}_0 \quad \varepsilon \quad \Rightarrow \varepsilon \\ \quad \mathbf{embtel}_{(1+\ n)}\ (\vec{T} :\!: T) \Rightarrow (\mathbf{embtel}\ \vec{T})\text{`}:\!:\text{'}(\mathbf{embspt}\ T) \end{array}$$

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

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

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

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

$$\underline{\text{data}}\ \frac{T : \mathsf{SPF}\ \vec{I}\ O \quad \vec{T} : \mathsf{dTel}\ \vec{I} \quad o : O}{[\![T]\!]\vec{T}\ o : \star}\ \underline{\text{where}}\ \frac{v : [\![T]\!]\vec{T}\ o}{\mathsf{top}\ v : [\![\text{`Z'}]\!](\vec{T} :: T)\ o} \quad \frac{v : [\![T]\!]\vec{T}\ o}{\mathsf{pop}\ v : [\![\text{`wk'}\ T]\!](\vec{T} :: S)\ o}$$

$$\frac{v : [\![f\ t]\!]\vec{T}\ o}{\mathsf{tag}_t\ v : [\![\text{`Tag'}f]\!]\vec{T}\ (o; t)} \quad \frac{}{\mathsf{void} : [\![\text{`1'}]\!]\vec{T}\ o} \quad \frac{v : [\![T]\!](\vec{T} :: (\text{`}\mu\text{'}\ T))\ o}{\mathsf{in}\ v : [\![\text{`}\mu\text{'}\ T]\!]\vec{T}\ o}$$

$$\frac{v : [\![T]\!]\vec{T}\ o}{\sigma_o\ v : [\![\text{`}\Sigma\text{'}f\ T]\!]\vec{T}\ (f\ o)} \quad \frac{v : [\![T]\!]\vec{T}\ (f\ o)}{\delta\ v : [\![\text{`}\Delta\text{'}f\ T]\!]\vec{T}\ o} \quad \frac{\vec{v} : \forall o : O \quad p : (f\ o) = o' \Rightarrow [\![T]\!]\vec{T}\ o}{\pi\vec{v} : [\![\text{`}\Pi\text{'}f\ T]\!]\vec{T}\ o'}$$

Figure 2. The SPF Universe

Finally we embed elements in the simpler universe as elements of the embedded code interpreted by the embedding of the telescope:

$$\underline{\text{let}}\ \frac{x : \mathsf{El}\ \vec{T}\ T}{\mathbf{embel}\ T : [\![\mathbf{embspt}\ T]\!](\mathbf{embtel}\ \vec{T})\ ()}$$

$$\begin{array}{lll}
\mathbf{embel}\ T \Leftarrow \underline{\text{rec}}\ T & & \\
\quad \mathbf{embel} & (\mathsf{top}\ t) & \Rightarrow \mathsf{top}\ (\mathbf{embel}\ t) \\
\quad \mathbf{embel} & (\mathsf{pop}\ t) & \Rightarrow \mathsf{top}\ (\mathbf{embel}\ t) \\
\quad \mathbf{embel} & \mathsf{void} & \Rightarrow \mathsf{void} \\
\quad \mathbf{embel} & (\mathsf{inl}\ t) & \Rightarrow \mathbf{inl}\ (\mathbf{embel}\ t) \\
\quad \mathbf{embel} & (\mathsf{inr}\ t) & \Rightarrow \mathbf{inr}\ (\mathbf{embel}\ t) \\
\quad \mathbf{embel} & (\mathsf{pair}\ s\ t) & \Rightarrow \mathbf{pair}\ (\mathbf{embel}\ s)\ (\mathbf{embel}\ t) \\
\quad \mathbf{embel} & (\mathsf{fun}\ f) & \Rightarrow \mathbf{fun}\ (\lambda k \Rightarrow \mathbf{embel}\ (f\ k)) \\
\quad \mathbf{embel} & (\mathsf{in}\ t) & \Rightarrow \mathsf{in}\ (\mathbf{embel}\ t)
\end{array}$$

4. Generic Map

To construct the generic map operation is a simple extension of the SPT map, indeed the morphisms are almost identical we only generalise to say that the functions must operate at any output index as we cannot before-hand anticipate which will turn up.

$$\underline{\text{data}}\ \frac{\vec{S}, \vec{T} : \mathsf{dTel}\ \vec{I}}{\mathsf{Morph}\ \vec{S}\ \vec{T} : \star}\ \underline{\text{where}}$$

$$\frac{}{\mathsf{mId} : \mathsf{Morph}\ \vec{S}\ \vec{S}}$$

$$\frac{\phi : \mathsf{Morph}\ \vec{S}\ \vec{T} \quad f : \forall_{i:I} \Rightarrow [\![S]\!]\vec{S}\ i \to [\![T]\!]\vec{T}\ i}{\mathsf{mFun}\ \phi\ f : \mathsf{Morph}\ (\vec{S} :: S)\ (\vec{T} :: T)}$$

$$\frac{\phi : \mathsf{Morph}\ \vec{S}\ \vec{T}}{\mathsf{mMap}\ \phi : \mathsf{Morph}\ (\vec{S} :: T)\ (\vec{T} :: T)}$$

The function itself is structural except at the variables where we do exactly as before:

$$\underline{\text{let}}\ \frac{\phi : \mathsf{Morph}\ \vec{S}\ \vec{T} \quad x : [\![T]\!]\vec{S}\ o}{\mathbf{gMap}\ \phi\ x : [\![T]\!]\vec{T}\ o}$$

$$\begin{array}{llll}
\mathbf{gMap} & \phi & x & \Leftarrow \underline{\text{rec}}\ x \\
\quad \mathbf{gMap} & \mathsf{mId} & (\mathsf{top}\ x) & \Rightarrow \mathsf{top}\ x \\
\quad \mathbf{gMap} & (\mathsf{mFun}\ \phi\ f) & (\mathsf{top}\ x) & \Rightarrow \mathsf{top}\ (f\ x) \\
\quad \mathbf{gMap} & (\mathsf{mMap}\ \phi) & (\mathsf{top}\ x) & \Rightarrow \mathsf{top}\ (\mathbf{gMap}\ \phi\ x) \\
\quad \mathbf{gMap} & \mathsf{mId} & (\mathsf{pop}\ x) & \Rightarrow \mathsf{pop}\ x \\
\quad \mathbf{gMap} & (\mathsf{mFun}\ \phi\ f) & (\mathsf{pop}\ x) & \Rightarrow \mathsf{pop}\ (\mathbf{gMap}\ \phi\ x) \\
\quad \mathbf{gMap} & (\mathsf{mMap}\ \phi) & (\mathsf{pop}\ x) & \Rightarrow \mathsf{pop}\ (\mathbf{gMap}\ \phi\ x) \\
\quad \mathbf{gMap} & \phi & (\mathsf{tag}\ x) & \Rightarrow \mathsf{tag}\ (\mathbf{gMap}\ \phi\ x) \\
\quad \mathbf{gMap} & \phi & \mathsf{void} & \Rightarrow \mathsf{void} \\
\quad \mathbf{gMap} & \phi & (\mathsf{in}\ x) & \\
 & & & \Rightarrow \mathsf{in}\ (\mathbf{gMap}\ (\mathsf{mMap}\ \phi)\ x) \\
\quad \mathbf{gMap} & \phi & (\sigma_o\ x) & \Rightarrow \sigma_o\ (\mathbf{gMap}\ \phi\ x) \\
\quad \mathbf{gMap} & \phi & (\delta\ x) & \Rightarrow \delta\ (\mathbf{gMap}\ \phi\ x) \\
\quad \mathbf{gMap} & \phi & (\pi\ \vec{x}) & \\
 & & & \Rightarrow \pi\ (\lambda o\ p \Rightarrow \mathbf{gMap}\ \phi\ (\vec{x}\ o\ p))
\end{array}$$

As we mentioned earlier the class of strictly positive types does not support all of the generic programs we might like to write in this way. The solution with SPTs is to loose the '$\to$' constructor and we arrive at the regular tee types from our previous work. We can play a similar trick here, although the source of the infinities that cause the trouble, 'Π', is rather more integral to the construction (giving us our '$\times$', for example) so, instead of removing it, we restrict it's power to finite domains. We introduce the regular families (RF), whose construction is identical to the above except we replace the 'Π' rules with:

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

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

This restriction is one that is easy to bear, only the construction of '$\to$' and '**Ord**' need the infinite pi in the examples given above and there is an obvious embedding of the finite into the infinite:

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

In the RF universe it is possible to define a generic equality which is structurally recursive on the data.

$$\underline{\text{let}}\ \frac{bs : \mathsf{Fin}\ n \to \mathsf{Bool}}{\mathbf{all}_n\ bs : \mathsf{Bool}}$$

$$\begin{array}{l}
\mathbf{all}_n\ bs \Leftarrow \underline{\text{rec}}\ n \\
\quad \mathbf{all}_0 \quad bs \Rightarrow \mathsf{true} \\
\quad \mathbf{all}_{(1+\ n)}\ bs \Rightarrow (bs\ 0) \wedge (\mathbf{all}\ (bs\ .\ 1+))
\end{array}$$

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

$$\begin{array}{lll}
\mathbf{gEq}\ a\ b \Leftarrow \underline{\text{rec}}\ a & & \\
\quad \mathbf{gEq}\ (\mathsf{top}\ a)\ (\mathsf{top}\ b) & & \Rightarrow \mathbf{gEq}\ a\ b \\
\quad \mathbf{gEq}\ (\mathsf{pop}\ a)\ (\mathsf{pop}\ b) & & \Rightarrow \mathbf{gEq}\ a\ b \\
\quad \mathbf{gEq}\ (\mathsf{tag}_{ta}\ a)\ (\mathsf{tag}_{tb}\ b) & \| \ ta{=}{=}tb & \\
 & \quad \mathsf{yes}\ (\mathsf{refl}\ t) & \Rightarrow \mathbf{gEq}\ a\ b \\
 & \quad \mathsf{no}\ _ & \Rightarrow \mathsf{false} \\
\quad \mathbf{gEq}\ \ \mathsf{void}\ \ \mathsf{void} & & \Rightarrow \mathsf{true} \\
\quad \mathbf{gEq}\ (\sigma_{oa}\ a)\ (\sigma_{ob}\ b) & & \Rightarrow \mathbf{gEq}\ a\ b \\
\quad \mathbf{gEq}\ (\delta\ a)\ (\delta\ b) & & \Rightarrow \mathbf{gEq}\ a\ b \\
\quad \mathbf{gEq}\ (\pi^{<\omega}\ \vec{a})\ (\pi^{<\omega}\ \vec{b}) & & \\
 & \Rightarrow \wedge(\forall i : \mathsf{Fin}\ n \Rightarrow \mathbf{gEq}\ \vec{a}\ \vec{b}) & \\
\quad \mathbf{gEq}\ \ (\mathsf{in}\ a)\ \ (\mathsf{in}\ b) & & \Rightarrow \mathbf{gEq}\ a\ b
\end{array}$$

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 $0 : \mathsf{Fin}\ n = 0 : \mathsf{Fin}\ (1+\ n)$). In practice it would be better to restrict ourselves only to comparing things for equality at the same index.

5. Conclusions and Further Work

We have here closed the gap left open in our previous work [9] and show how to construct a universe which captures strictly positive families, i.e. all of Epigram's datatypes. This includes the universe itself, since SPF, dTel, $[\![-]\!]$ are strictly positive families and can be laboriously encoded as instances of SPF — for reasons of space, and sanity, we refrain from spelling out the details here. One may wonder whether such a circular construction is necessarily paradoxically, as in Girard's paradox which rules out a type of all types in a consistent theory. However, closer inspection shows that this is not the case but that the universe construction raises the level of the predicative hierarchy of types $\star_n : \star_{n+1}$, which is ignored in the current implementation of Epigram.

We have only touched the potential of the universe presented here. E.g. instead of generic simply typed programs like **gMap** or **gEq** we should consider their dependently typed counterparts, which integrate generic proofs and programs. In the case of generic map this involves defining the modality $\Box$ which, in logic (and here, if we appeal to the Curry-Howard isomorphism), lifts a predicate $P : A \to \star$ over a collection $F : A \Rightarrow \star$ which holds iff P holds for every A position in an $x : F\ A$. For instance $\Box\mathsf{List}$:

$$\underline{\text{data}}\ \frac{P : A \to \star \quad as : \mathsf{List}\ A}{\Box\mathsf{List}\ P\ as : \star}\ \underline{\text{where}}$$

$$\frac{}{\varepsilon : \Box\mathsf{List}\ P\ A\ \varepsilon} \qquad \frac{p : P\ a \quad ps : \Box\mathsf{List}\ P\ as}{p{:}ps : \Box\mathsf{List}\ P\ (a{:}as)}$$

Given a dependent function $f : \forall x : A \Rightarrow B\ a$, **mapList** f then has the type $\forall as : \mathsf{List}\ A \Rightarrow \Box\mathsf{List} B\ as$. It turns out that we can write a function $\Box$ that calculates a code for this modality for any SPF, meaning that dependent map is also a generic program. In the case of generic equality between RFs we can not only give a function into Bool but give a function that returns the evidence of equality or inequality between the arguments much as we've done for the regular tree types.

In joint, so far unpublished work with Ghani, Hancock and McBride [2] we have shown how to extend the notion of container types [1] to cover strictly positive families, arriving at a semantic interpretation of our universe which gives an alternative access to generic programming with and for dependent types.

We plan to implement a reflection principle based on the universe construction presented here in the forthcoming new release of Epigram, thus giving the Epigram programmer full access to all datatypes definable in Epigram.

References

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

[2] T. Altenkirch, N. Ghani, P. Hancock, C. McBride, and P. Morris. Indexed containers. Manuscript, available online, February 2006.

[3] T. Altenkirch, C. McBride, and J. McKinna. Why dependent types matter. Manuscript, available online, April 2005.

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

[5] P. Dybjer and A. Setzer. Indexed induction-recursion. In R. Kahle, P. Schroeder-Heister, and R. F. Stärk, editors, *Proof Theory in Computer Science*, volume 2183 of *Lecture Notes in Computer Science*, pages 93–113. Springer, 2001.

[6] P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis·Napoli, 1984.

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

[8] C. McBride and J. McKinna. The view from the left. *Journal of Functional Programming*, 14(1), 2004.

[9] P. Morris, T. Altenkirch, and C. McBride. Exploring the regular tree types. In C. P.-M. Jean-Christophe Filliatre and B. Werner, editors, *Types for Proofs and Programs (TYPES 2004)*, Lecture Notes in Computer Science, 2006.

[10] B. Nordström, K. Petersson, and J. Smith. *Programming in Martin-Löf's Type Theory: an introduction*. Oxford University Press, 1990.