Constructing Strictly Positive Families

Peter Morris **Thorsten Altenkirch** **Neil Ghani**

School of Computer Science and Information Technology
University of Nottingham, England,
Email:{pwm,txa,nxg}@cs.nott.ac.uk

Abstract

In order to represent, compute and reason with advanced data types one must go beyond the traditional treatment of data types as being inductive types and, instead, consider them as inductive families. Strictly positive types (SPTs) form a grammar for defining inductive types and, consequently, a fundamental question in the the theory of inductive families is what constitutes a corresponding grammar for inductive families.

This paper answers this question in the form of *strictly positive families* or SPFs. We show that these SPFs can be used to represent and compute with a variety of advanced data types, that generic programs can naturally be written over the universe of SPFs and that SPFs have a normal form in terms of indexed containers which are based upon the shapes and positions metaphor. Finally, we validate our computational perspective by implementing SPFs in the programming language Epigram and, further, comment on how SPFs provide a meta-language for Epigram's data types.

Keywords: Advanced Data Types, Dependently Typed Programming, Generic Programming, Epigram

1 Introduction

The search for an expressive calculus of data types in which canonical algorithms can be easily written and proven correct has proved to be an enduring challenge to the theoretical computer science community. Ideally, we want a calculus of data types which allows programs to be written in a natural style and which also has a clear semantic foundation so as to justify principles for reasoning about such programs. Approaches such as polynomial types, strictly positive types and inductive types have all met with much success but they tend not to cover advanced data structures, e.g. types with variable binding such as untyped λ-terms, types with constraints such as square matrices and dependent types such as the type of finite sets.

Our first key observation is that in order to represent, compute and reason with such advanced data types one must go beyond the traditional treatment of data types as being inductive types and, instead, consider them as inductive families. To understand this, consider as an example the natural numbers which is an inductive type $\mathsf{Nat} : \star$. Going further, one may next consider the list type constructor $\mathsf{List} : \star \to \star$. Notice that, crucially, $\mathsf{List}\ A$ is an inductive type and does not depend upon, i.e. can be defined independently of, $\mathsf{List}\ B$ for any $A \neq B$. Thus, List is a *family* of inductive types indexed by the type of small types.

In contrast to the family of inductive types List, consider $\mathsf{Fin} : \mathsf{Nat} \to \star$ which is defined inductively by the constructors

$$\mathsf{fz} : \mathsf{Fin}\ (1{+}\ n) \qquad \mathsf{fs} : \mathsf{Fin}\ n \to \mathsf{Fin}\ (1{+}\ n)$$

Concretely, $\mathsf{Fin}\ n$ represents the finite type with n elements, fz and fs are the zero and successor of these types where fz exists in every non-empty Fin type and fs embeds elements of $\mathsf{Fin}\,n$ into $\mathsf{Fin}\,(1{+}\,n)$. In effect the type $\mathsf{Fin}\,3$ contains elements that look much like 0,1 and 2: $\mathsf{fz}, \mathsf{fs}\,\mathsf{fz}$ and $\mathsf{fs}\,(\mathsf{fs}\,\mathsf{fz})$, later in the paper we will use the type $\mathsf{Fin}\,n$ to index into collections of n items. The key point is that, unlike the case with lists, the type $\mathsf{Fin}\ n$ cannot be defined in isolation and with recourse only to the elements of $\mathsf{Fin}\ n$ that have already been built. Rather, we need elements of the type $\mathsf{Fin}\ n$ to build elements of $\mathsf{Fin}\ (1{+}\ n)$ etc. In effect, the Nat-indexed family $\mathsf{Fin} : \mathsf{Nat} \to \star$ has to be inductively built up simultaneously for every n and is thus an *inductive family* of types rather than a family of inductive types.

Our interests are in total programming and concrete data types so we avoid negative occurrences in definitions and the pathological issues raised by non-strict positivity by concentrating on the strictly positive. Strictly positive types form a grammar for defining inductive types and, consequently, a fundamental question in the theory of inductive families is what is a corresponding grammar for inductive families. This paper answers this question in the form of *strictly positive families* or SPFs. In detail, the contributions of this paper are:

- We show that these SPFs are expressive in that they can be used to represent and compute with a variety of advanced data types. To do this we define a number of SPFs, and programs which manipulate them, in the programming language Epigram.
- We define a data type whose elements are names, or *codes*, of strictly positive families and a decoding function which assigns to each code, the actual elements of the type it represents. This construction is an example of a universe (Martin-Löf 1984, Nordström, Petersson & Smith 1990) and allows us to write generic programs for SPFs by simply writing programs which manipulate this universe.
- 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.

- Containers (Abbott, Altenkirch & Ghani 2005) are an alternative presentation of data types which focuses less on the inductive structure of data types and more on the fact that they store data at positions within the data type. In loc.cit. we show that all strictly positive types can be interpreted as containers which gives an alternative access to generic programming: a generic program is simply one which works on all containers — see for example our treatment of derivatives of data types (Abbott, Altenkirch, Ghani & McBride 2005).

  In as yet unpublished work (Altenkirch, Ghani, Hancock, McBride & Morris 2006) we have generalized containers to indexed containers which capture inductive families. In the present paper we relate the two approaches showing that any SPF gives rise to a semantically equivalent indexed container.

- We give a full implementation of all our constructions in Epigram which is a dependently typed programming language (McBride & McKinna 2004, McBride 2004, Altenkirch, McBride & McKinna 2005). Not only does Epigram provide a language to implement SPFs, but SPFs are also sufficiently expressive to provide a metalanguage for Epigram's data types.

Therefore this paper will appeal to those interested in the theory of data types, generic programming and type theory. In particular, we are interested in the relationship between indexed containers with shapely types.

Related Work: Previously we defined the universe of regular tree types and developed generic programs and proofs for this universe (Morris, Altenkirch & McBride 2006). The goal of this paper is thus to extend this universe-based approach to generic programming to cover the more advanced SPFs.

In related work, Dybjer and Setzer (Dybjer & Setzer 2001) present an alternative approach to defining a universe of indexed families by giving an axiomatization of indexed inductive recursive definitions. This has been the base for generic programming within the AGDA system (Benke, Dybjer & Jansson 2003).

Containers are a natural generalisation of the shapely types of (**?**) while indexed containers have been studied under the name of dependent polynomials in (**?**).

Structure of the Paper: The rest of the paper is structured as follows: Since our use of dependent types, universes and containers are a novel approach to the theory of data types, we begin in section 2 by reviewing the construction in Epigram of the universe of SPTs and the use of this universe in providing a framework for generic programming. In section 3 we discuss the elements of the grammar of SPFs, while in section 4 we give a variety of examples of SPFs and discuss the composition of SPFs. In section 5, we give a number of generic programs for SPFs while section 6 shows that every SPF is an indexed container. In section 7, we conclude with some final remarks. Finally, if possible, we ask readers to print this paper in colour as we have used the Epigram colouring scheme to improve legibility of code. For example, constructors always occur in red, type constructors in blue, defined constants in green etc.

2 SPTs, Epigram and Universes

We begin the paper by recalling strictly positive types, their implementation in Epigram and the representation of the type of strictly positive types as a universe in Epigram. The rest of the paper will apply this treatment to the more advanced strictly positive families.

2.1 Strictly Positive Types

We introduce strictly positive types (SPTs) by way of a generative grammar as follows:

$$\tau = X \mid 0 \mid 1 \mid \tau + \tau \mid \tau \times \tau \mid K \to \tau \mid \mu X.\tau$$

where X ranges over type variables, 0 and 1 represent the empty and unit types, the operators $+$ and $\times$ stand for disjoint union and cartesian product. If K is a constant type (an SPT with no free type variables) then $K \to -$ is exponentiation by that constant. Finally the least fixed point operator (μ) creates recursive types by binding a type variable. Examples of SPTs include the natural numbers, lists, rose trees and ordinal notations:

$$\begin{aligned} \mathrm{Nat} &= \mu X.1 + X \\ \mathrm{List}\, A &= \mu X.1 + A \times X \\ \mathrm{RT}\, A &= \mu Y.A \times \mathrm{List}\, Y \\ &= \mu Y.A \times (\mu X.1 + Y \times X) \\ \mathrm{Ord} &= \mu X.1 + X + \mathrm{Nat} \to X \end{aligned}$$

The first three examples, which don't use exponentiation are regular tree types which are a proper subset of strictly positive types.

SPTs have traditionally been used as part of the semantics of programming languages and, for such applications, the informal grammar given above is adequate. However, in order to reason about and to program with SPTs, we need a formal definition of SPTs. For this reason we define a type SPT n whose elements consist of the names or *codes* of SPTs and a decoding function which computes the elements of an SPT. This construction forms a universe for SPTs so that generic programming with SPTs can then be achieved by writing programs which manipulate this universe. This construction of a universe of SPTs requires a dependently typed programming language and we now give a summary of one such language, namely Epigram.

2.2 Epigram

Epigram is a dependently typed functional language with an interactive environment for developing programs with the aid of the type checker. Epigram's syntax is based on Type Theory, using $\lambda x : A \Rightarrow t$ for λ abstraction, $\forall x : A \Rightarrow t$ for Π-types and $\exists x : A \Rightarrow t$ for Σ-types. All type annotations can be omitted when inferable by the context. $\star$ stands for the type of types which is implicitly stratified, i.e. we have $\star_i : \star_{i+1}$ but omit the indices.

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 consists of a left-hand side of a pattern match defining the problem to be solved plus one of three possible right-hand sides:

$\Rightarrow t$ 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 number 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 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. Here are the natural numbers and addition in Epigram:

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

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

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

We can then define types which are dependent on the natural numbers such as the finite types and vectors (lists of a given length) and a safe projection using the finite types to ensure there are only as many indexes as elements in the array - the nil case doesn't appear since Fin 0 is uninhabited. In this example, the correctness by construction ideal is achieved by means of type checking, but this could only be done because of the extra sophistication of dependent types.

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

$$\frac{}{\mathsf{fz} : \mathsf{Fin}\ (1{+}\, n)}\quad \frac{i : \mathsf{Fin}\ n}{\mathsf{fs}\ i : \mathsf{Fin}\ (1{+}\, n)}$$

$$\underline{\text{data}}\ \frac{A : \star \quad n : \mathsf{Nat}}{\mathsf{Vec}\ n\ A : \star}\ \underline{\text{where}}$$

$$\frac{}{\varepsilon : \mathsf{Vec}\ 0\ A}\quad \frac{a : A \quad as : \mathsf{Vec}\ n\ A}{a{:}as : \mathsf{Vec}\ (1{+}\, n)\ A}$$

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

$$\begin{array}{l}as!!i \Leftarrow \underline{\text{rec}}\ i\\ \quad (a{:}as)!!\mathsf{fz} \Rightarrow a\\ \quad (a{:}as)!!(\mathsf{fs}\ i) \Rightarrow as!!i\end{array}$$

Any universe to capture the strictly positive families would need to include these examples, but not only Nat indexed families must be encodable. One family we will use in our constructions is known as the W-types and can be used to construct any tree like data structure in a higher order fashion:

$$\underline{\text{data}}\ \frac{A : \star \quad B : A \to \star}{\mathsf{W}\ A\ B : \star}\ \underline{\text{where}}$$

$$\frac{a : A \quad f : B\ a \to \mathsf{W}\ A\ B}{\mathsf{sup}\ a\ f : \mathsf{W}\ A\ B}$$

2.3 A Universe for Strictly Positive Types

The traditional, informal, definition of SPTs given above is problematic when we come to compute as it is not a data type in a language. This is a fundamental problem if we want generic programming to be programs which manipulate the type of SPTs.

Codes for SPTs: To rectify this problem, we represent the syntax of SPTs with n free type variables by the Epigram type SPT n, see figure 1. We use de Bruijn notation to represent type variables - thus Vz [1] represents the type variable with index 0 while previous type variables are represented via weakening Vs. The empty type 0 and unit type 1 are represented by '0' and '1' while sums and products of SPTs are represented using the SPT constructors '+' and '×'. Finally, note that the fix-point constructor 'μ' reduces the number of free type variables by 1 because the last variable has been bound. In summary, SPT n represents names or *codes* for SPTs.

Here are the codes for the four examples above.

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

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

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

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

Interpretation of SPTs: Recall that so far we constructed, for each SPT containing (at most) n type variables, a name or code which is an expression of type SPT n. Thus we have a data type that represents the syntax of SPTs. Of course, there is no guarantee that '0' behaves like the empty type or that S '+' T behaves like the sum of S and T.

In order to ensure that the codes for SPTs behave as intended, we give an interpretation El which intuitively assigns, to each code T : SPT n and appropriate n-tuple of inputs, the type of elements of the actual SPT. In order that this construction can be formalised within the universe of SPTs, we require each input to be an SPT. Further, the interpretation of fixed points shows that the $n+1$'th SPT must be able to depend on the previous n-SPTs. Such an input is called a *telescope* and we therefore introduce the type of telescopes of length n which we denote Tel n.

Then, given a type T : SPT n and a matching telescope $\vec{T}$: Tel n we define the type of elements El $\vec{T}$ T. The idea is that, for example, El $\vec{T}$ '1' will really have one element showing that '1' really is the unit type, and that El $\vec{T}$ (S '+' T) really is the sum of El $\vec{T}$ S and El $\vec{T}$ T). The universe of SPTs thus

[1] The quotes here have no semantic significance, but rather remind the reader that this is a code.

consists of the codes given by SPT and the intended meanings of these codes given by El. See Figure 1 for the full definition of this universe.

2.4 Generic Map

As our first example of a generic program, we shall present a generic map operation for all SPTs by using the universe of SPTs. We shall define this by first considering morphisms between telescopes. Had a telescope of length n been an n-tuple of types, a morphism between two telescopes of length n would have been an n-tuple of functions between the associated types. However, since an SPT in a telescope can depend upon the previous SPTs in a telescope, this information must also be taken into account as is shown in the mF constructor for morphisms.

In the mu case we would like to have:

$$\mathbf{gMap}\ \phi\ (\mathsf{in}\ x)\ \Rightarrow\ \mathbf{gMap}\ (\mathsf{mF}\ \phi\ (\mathbf{gMap}\ \phi))\ x$$

However, the nested recursive call is not guaranteed to be structurally recursive since it could be eventually applied to anything - hence this definition would be rejected by Epigram. To solve this problem, we introduce a third constructor for morphisms $\mathsf{mU}\phi$ which stands for extending ϕ by $\mathbf{gMap}\ \phi$ as follows.

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

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

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

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

We now have the following, obviously structural definition for **gMap**:

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

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

In our work on the regular tree types (Morris et al. 2006), ie those SPTs which are finitely branching, we present a number of other algorithms in this style including a decidable equality. Types in the SPT universe do not have such an equality since they permit infinite branching - for example there is no such decidable equality function for the ordinals '**Ord**'. 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 larger universes to cope with this trade off.

We now turn to the central question of this paper. That is, can we find a grammar of SPFs for inductive families similar to the grammar of SPTs for inductive types? Further, can we construct a universe for SPFs which allows us to program generically with SPFs?

3 Strictly Positive Families

Recall that our central motivation for studying inductive families is that inductive types cannot capture advanced data types such as Fin and Vec above. Another nice example of an inductive family is the type of untyped λ-terms in n free variables, which can be defined as follows using de Bruijn indices to refer to variable:

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

$$\frac{i : \mathsf{Fin}\ n}{\mathsf{var}\ i\ :\ \mathsf{Lam}\ n} \qquad \frac{f, a : \mathsf{Lam}\ n}{\mathsf{app}\ f\ a\ :\ \mathsf{Lam}\ n} \qquad \frac{b : \mathsf{Lam}\ (1{+}\ n)}{\mathsf{abs}\ b\ :\ \mathsf{Lam}\ n}$$

How To Construct Families: Recall that SPTs were essentially constructed as fixed points of polynomials but, rather surprisingly, SPFs are actually not constructed from polynomials. This is because the fundamental structure of families lies in the indexes which were not present in the SPT case. Recall that an O-indexed family is a function $F : O \to \star$ — in our examples so far we have looked only at Nat indexed families although, in general, O can be *any* type. If $t : F\ o$, we say that t is indexed by o.

To gain some intuition about how families can be constructed, let us now define a universe for O-indexed families - that is a type $\mathsf{Fam}\,O$ consisting of codes for O-indexed families and an interpretation $[\![-]\!] : \mathsf{Fam}\ O \to O \to \star$ which associates to each code, the actual family. The simplest such families are constant and ignore the indexing information - we have two such families, '0' and '1' which have zero or one elements (respectively) at all indices.

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

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 $O' \to \star$ by composition. Thus we add a constructor for Fam and give its interpretation.

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

This operation on families corresponds categorically to re-indexing of functors we use that same term to describe it here for families. It is interesting now to consider what we can do to construct families 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 just *one* possible value, or *all* possibilities? The first option gives us dependent sum, the second dependent product:

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

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

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

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

$$\frac{}{\mathsf{Vz} : \mathsf{SPT}\ (1{+}\ n)} \qquad \frac{T : \mathsf{SPT}\ n}{\mathsf{Vs}\ T : \mathsf{SPT}\ (1{+}\ n)} \qquad \frac{}{\text{`0'} : \mathsf{SPT}\ n} \qquad \frac{S, T : \mathsf{SPT}\ n}{S\ \text{`+'}\ T : \mathsf{SPT}\ n}$$

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

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

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

$$\frac{e : \mathsf{El}\ \vec{T}\ T}{\mathsf{top}\ e : \mathsf{El}\ (\vec{T} \text{:-} T)\ \mathsf{Vz}} \qquad \frac{e : \mathsf{El}\ \vec{T}\ T}{\mathsf{pop}\ e : \mathsf{El}\ (\vec{T} \text{:-} S)\ (\mathsf{Vs}\ T)} \qquad \frac{}{\mathsf{void} : \mathsf{El}\ \vec{T}\ \text{`1'}} \qquad \frac{f : K \rightarrow \mathsf{El}\ \vec{T}\ T}{\mathsf{fun}\ f : \mathsf{El}\ \vec{T}\ (K\ \text{`}\rightarrow\text{'}\ T)}$$

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

Figure 1: The SPT Universe

In the above we are using Epigram's built in equality type, which for the purposes of this paper can be viewed as being defined, thus:

$$\underline{\text{data}}\ \frac{a : A \quad b : B}{a = b : \star}\ \underline{\text{where}}\ \frac{}{\mathsf{refl} : a = a}$$

Categorically, 'Σ' and 'Π' are respectively the left and right adjoints of 'Δ'. That these operations have universal properties suggests we are basing our constructors on solid mathematical foundations as we said was important in the introduction.

Although we called these two constructors dependent sum and dependent product, all the constructors for families introduced so far are linear. For example 'Σ' f maps a single family T to the single family 'Σ' $f\ T$. Therefore, to define constructors which take as input more than one family, we require a non-linear constructor of families. There are a number of possibilities but we choose the intuitively simple option of constructing finite sums of families as follows:

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

In the above, $(o; t)$ represents the pair consisting of o and t. Finally we add the fixed point constructor and variables standing for families. As with SPTs, a family may contain several variables (accessed by de Bruijn indices) but, now, each of these variables represents families which could be indexed over different types.

To conclude, a natural set of constructors for forming strictly positive families is given by variable families, unit and empty families, 'Σ', 'Π', 'Δ'-families and fixed points of families. We could at this point give an grammar for SPFs based upon these constructors and similar to that for SPTs. However, we prefer to go straight to the construction of a universe for SPFs.

A Universe of SPFs: We now define a universe of SPFs guided by the discussion above. The type of SPFs will be similar to that for SPTs, except each input and output will require an index. Thus we get our type to represent the syntax of SPFs:

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

Our intuition is that each element of $\mathsf{SPF}\ \vec{I}\ O$ will be an SPF which takes as input families whose indexes are represented by $\vec{I}$ and will return a family indexed by O. The definition of SPF is given in figure 2

As with SPTs, we now define an interpretation for SPFs. As before the crucial ingredient in for this interpretation is the type of telescopes Tel which must now include indexing information.

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

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

Once we have defined telescopes we can define the type of elements inductively by giving the value constructors associated to each family constructor. As it has to be expected, there are none for the empty type '0'. This construction can also be found in Figure 2 and follows exactly the intuition developed in the first half of section 3.

If we allow arbitrary functions for 'Π' we obtain strictly positive families which are potentially infinitely branching and, similarly to the case with SPTs, this limits the range of definable generic operations. For example there is no generic equality on SPFs. An alternative, in line with our previous work on regular tree types, is to define the type of regular families $\mathsf{RF}\ \vec{I}\ O$ which is obtained by replacing 'Π' by

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

The SPF Type codes:

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

$$\frac{}{\mathsf{Vz} : \mathsf{SPF}\ (\vec{I} :\!: O)\ O} \qquad \frac{T : \mathsf{SPF}\ \vec{I}\ O}{\mathsf{Vs}\ T : \mathsf{SPF}\ (\vec{I} :\!: I)\ O}$$

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

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

The Interpretation of SPF:

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

$$\frac{v : [\![T]\!]\vec{T}\ o}{\mathsf{top}\ v : [\![\mathsf{Vz}]\!](\vec{T} :\!: T)\ o} \qquad \frac{v : [\![T]\!]\vec{T}\ o}{\mathsf{pop}\ v : [\![\mathsf{Vs}\ T]\!](\vec{T} :\!: S)\ o}$$

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

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

Figure 2: The SPF Universe

whose elements can be constructed by

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

There is an obvious embedding of the finite $\text{`}\Pi^{<\omega}\text{'}$ in to the possibly infinite `Π` given by:

$$\begin{aligned} &\text{`}\Pi^{<\omega}\text{'}n\ T : \mathsf{RF}\ \vec{I}\ O \ \mapsto \\ &\text{`}\Pi\text{'}(\lambda(o; i) : O \times \mathsf{Fin}\ n \Rightarrow o)\ T : \mathsf{SPF}\ \vec{I}\ O \\ &\pi^{<\omega}\ \vec{v} \ \mapsto \ \pi(\lambda(o; i); \mathsf{refl} \Rightarrow \vec{v}\ i) \end{aligned}$$

where refl is the only constructor for the equality type as we defined earlier.

4 Examples of SPFs

To give examples of data types 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. Firstly for sums we have

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

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

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

$$\text{`}\mathbf{inl}\text{'}\ a \ \Rightarrow\ \sigma\ (\mathsf{tag}\ \mathsf{fz}\ a)$$

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

$$\text{`}\mathbf{inr}\text{'}\ b \ \Rightarrow\ \sigma\ (\mathsf{tag}\ (\mathsf{fs}\ \mathsf{fz})\ b)$$

where

$$\left(\lambda\ \begin{matrix}\mathsf{fz} & \Rightarrow & A\\ \mathsf{fs}\ \mathsf{fz} & \Rightarrow & B\end{matrix}\right)$$

denotes the function whose domain is Fin 2 and which returns A on fz and B on fs fz. For the products we have

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

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

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

$$\text{`}\mathbf{pair}\text{'}\ a\ b \ \Rightarrow\ \pi^{<\omega}\left(\lambda\ \begin{matrix}\mathsf{fz} & \Rightarrow & \mathsf{tag}\ \mathsf{fz}\ a\\ \mathsf{fs}\ \mathsf{fz} & \Rightarrow & \mathsf{tag}\ (\mathsf{fs}\ \mathsf{fs})\ b\end{matrix}\right)$$

We can now encode some of our examples from above, we encode $\mathsf{Fin} : \mathsf{Nat} \to \star$ as an element of $\mathsf{RF}\ []\ \mathsf{Nat}$ and $\mathsf{Vec}\ A\ n : \star$ as an instance of $\mathsf{RF}\ [\mathsf{One}]\ \mathsf{Nat}$ denoting that it is a Nat indexed family with one type of 'input' which is indexed by One[2]:

$$\underline{\text{let}}\ \ \text{`}\mathbf{Fin}\text{'} : \mathsf{RF}\ []\ \mathsf{Nat}$$

$$\text{`}\mathbf{Fin}\text{'} \Rightarrow \text{`}\mu\text{'}\ ((\text{`}\Sigma\text{'}1 + \text{`}1\text{'})\ \text{`}+\text{'}\ (\text{`}\Sigma\text{'}1 + \mathsf{Vz}))$$

$$\underline{\text{let}}\ \ \text{`}\mathbf{Vec}\text{'} : \mathsf{RF}\ [\mathsf{One}]\ \mathsf{Nat}$$

$$\text{`}\mathbf{Vec}\text{'} \Rightarrow \text{`}\mu\text{'}\ \left(\begin{matrix}(\text{`}\Sigma\text{'}\ (\mathbf{const}\ 0)\ \text{`}1\text{'})\\ \text{`}+\text{'}\ \left(\begin{matrix}(\text{`}\Delta\text{'}(\mathbf{const}\ ())\ (\mathsf{Vs}\ \mathsf{Vz}))\\ \text{`}\times\text{'}\ (\text{`}\Sigma\text{'}1 + \mathsf{Vz})\end{matrix}\right)\end{matrix}\right)$$

We use the definitions above to present the type in a 'sums of products' style, with added indexing information. In the `ε` case for vectors $\text{`}\Sigma\text{'}(\mathbf{const}\ 0)$ forces

[2] since we have to treat types uniformly the type A becomes a family whose index carries no information

the empty vector to always have index zero; in the (‘::’) case, ‘Σ’1+ forces the vector a‘::’as to have index/length 1+ n if as has index/length n. We can encode values of the finite sets and vectors using generic constructors such as these:

$$\underline{\text{let}}\ \frac{}{\text{‘fz’} : [\![\text{‘Fin’}]\!]\,[]\ (1{+}\ n)}$$
$$\text{‘fz’} \Rightarrow \mathsf{in}\ (\text{‘inl’}(\sigma\ \mathsf{void}))$$

$$\underline{\text{let}}\ \frac{i : [\![\text{‘Fin’}]\!]\,[]\ n}{\text{‘fs’}\ i : [\![\text{‘Fin’}]\!]\,[]\ (1{+}\ n)}$$
$$\text{‘fs’}\ i \Rightarrow \mathsf{in}\ (\text{‘inr’}(\sigma\ (\mathsf{top}\ i)))$$

$$\underline{\text{let}}\ \frac{A : \mathsf{SPF}\ []\ \mathsf{One}}{\text{‘}\varepsilon\text{’} : [\![\text{‘Vec’}]\!][A]\ 0}$$
$$\text{‘}\varepsilon\text{’} \Rightarrow \mathsf{in}\ (\text{‘inl’}\ (\sigma\ \mathsf{void}))$$

$$\underline{\text{let}}\ \frac{a : [\![A]\!][]\ ()\quad as : [\![\text{‘Vec’}]\!][A]\ n}{(a\text{‘::’}as) : [\![\text{‘Vec’}]\!][A]\ (1{+}\ n)}$$
$$(a\text{‘::’}as) \Rightarrow \mathsf{in}\ (\text{‘inr’}\ (\text{‘pair’}\ (\delta\ a)\ (\sigma\ (\mathsf{pop}\ (\mathsf{top}\ as)))))$$

As another example, we can encode lambda terms $\mathsf{Lam}\ n : \star$, whose Epigram definition was given above, as the following SPF and generic constructors.

$$\underline{\text{let}}\ \text{‘Lam’} : \mathsf{RF}\ []\ \mathsf{Nat}$$
$$\text{‘Lam’} \Rightarrow \text{‘}\mu\text{’}\ \left(\begin{array}{l}((\mathsf{Vs}\ \text{‘Fin’})\ \text{‘+’}\ (\mathsf{Vz}\ \text{‘}\times\text{’}\ \mathsf{Vz}))\\ \quad \text{‘+’}\ (\text{‘}\Delta\text{’}1{+}\ \mathsf{Vz})\end{array}\right)$$

$$\underline{\text{let}}\ \frac{i : [\![\text{‘Fin’}]\!]\,[]\ n}{\text{‘var’}\ i : [\![\text{‘Lam’}]\,[]\ n]\!]}$$
$$\text{‘var’}\ i \Rightarrow \mathsf{in}\ (\text{‘inl’}\ (\text{‘inl’}\ (\mathsf{pop}\ (\mathsf{top}\ i))))$$

$$\underline{\text{let}}\ \frac{f : [\![\text{‘Lam’}]\!]\,[]\ n\quad a : [\![\text{‘Lam’}]\!]\,[]\ n}{\text{‘app’}\ f\ a : [\![\text{‘Lam’}]\,[]\ n]\!]}$$
$$\text{‘app’}\ f\ a \Rightarrow \mathsf{in}\ (\text{‘inl’}\ (\text{‘inr’}\ (\text{‘pair’}\ (\mathsf{top}\ f)\ (\mathsf{top}\ a))))$$

$$\underline{\text{let}}\ \frac{f : [\![\text{‘Lam’}]\!]\,[]\ (1{+}\ n)}{\text{‘abs’}\ f : [\![\text{‘Lam’}]\,[]\ n]\!]}$$
$$\text{‘abs’}\ f \Rightarrow \mathsf{in}\ (\text{‘inr’}\ (\delta\ \ (\mathsf{top}\ f)))$$

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:

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

The translation of this definition is not completely straightforward as the type Vec appears inside NBrTree. At the categorical level, this operation is modeled by the *composition* of functors and so we define a composition operator on SPFs. If we regard SPFs as syntax trees with constructors at the nodes and variables at the leaves, then this composition operator will replace the variables of the outer SPF with expressions whose type is that of the the inner SPF. The definition of composition is given as follows, with F standing for either SPF or for RF

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

$$\begin{array}{rcl}
C{\circ}\vec{D} & \Leftarrow & \underline{\text{rec}}\ C\\
\mathsf{Vz}{\circ}\vec{D} & \Rightarrow & \text{‘}\Delta\text{’}(\lambda x \Rightarrow (x;x))\ (\vec{D}\ \mathsf{fz})\\
(\mathsf{Vs}\ T){\circ}\vec{D} & \Rightarrow & \mathsf{Vs}\,(T{\circ}\vec{D}\cdot \mathsf{fs})\\
\text{‘0’}{\circ}\vec{D} & \Rightarrow & \text{‘0’}\\
\text{‘1’}{\circ}\vec{D} & \Rightarrow & \text{‘1’}\\
(\text{‘Tag’}f){\circ}\vec{D} & \Rightarrow & \\
 & & \text{‘Tag’}(\lambda i \Rightarrow (f\ i){\circ}(\mathbf{map}\ (\text{‘}\Delta\text{’}\mathbf{fst})\ \vec{D}))\\
(\text{‘}\Sigma\text{’}f\ T){\circ}\vec{D} & \Rightarrow & \text{‘}\Sigma\text{’}f\ (T{\circ}(\mathbf{map}\ (\text{‘}\Delta\text{’}(\mathbf{id};f))\ \vec{D}))\\
(\text{‘}\Delta\text{’}f\ T){\circ}\vec{D} & \Rightarrow & \text{‘}\Delta\text{’}f\ (T{\circ}(\mathbf{map}\ (\text{‘}\Sigma\text{’}(\mathbf{id};f))\ \vec{D}))\\
(\text{‘}\Pi^{<\omega}\text{’}n\ T){\circ}\vec{D} & \Rightarrow & \\
 & & \text{‘}\Pi^{<\omega}\text{’}n\ (T{\circ}(\mathbf{map}\ (\text{‘}\Delta\text{’}(\mathbf{id};\mathbf{fst}))\ \vec{D}))\\
(\text{‘}\Pi\text{’}f\ T){\circ}\vec{D} & \Rightarrow & \text{‘}\Pi\text{’}f\ (T{\circ}(\mathbf{map}\ (\text{‘}\Delta\text{’}(\mathbf{id};f))\ \vec{D}))\\
(\text{‘}\mu\text{’}\ F){\circ}\vec{D} & \Rightarrow & \text{‘}\mu\text{’}\ (F{\circ}\vec{\mathbf{D}}\mathbf{Ext})\\
\multicolumn{3}{l}{\quad \underline{\text{where}}\ \ \vec{\mathbf{D}}\mathbf{Ext}\ \mathsf{fz} \Rightarrow \text{‘}\Sigma\text{’}(\lambda x \Rightarrow (x;x))\ \mathsf{Vz}}\\
\multicolumn{3}{l}{\qquad\qquad \vec{\mathbf{D}}\mathbf{Ext}\ (\mathsf{fs}\ i) \Rightarrow \vec{D}\ i}
\end{array}$$

There is not enough space to fully explain the above definitions. However, note that i) in the first case we choose the last family in $\vec{D}$ and then re-index; ii) in the second case we throw away the last family of $\vec{D}$, compose and then weaken; iii) the unit and empty families compose as expected; iv) since ‘Δ’, ‘Σ’, ‘Π’ and ‘Tag’ change only the output indexes, but not the families, it is natural that composition distributes over them; and v) the composition of a fixed point is well known to be a fixed point.

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

$$\underline{\text{let}}\ \text{‘NBrTree’} : \mathsf{RF}\ [\mathsf{One}]\ \mathsf{Nat}$$
$$\text{‘NBrTree’} \Rightarrow \text{‘}\mu\text{’}\ \left(\begin{array}{l}(\text{‘}\Delta\text{’}(\mathbf{const}\ ())\ (\mathsf{Vs}\,\mathsf{Vz}))\\ \quad \text{‘+’}\,(\text{‘Vec’}{\circ}(\lambda \mathsf{fz} \Rightarrow \text{‘}\Delta\text{’}\mathbf{snd}\ \mathsf{Vz}))\end{array}\right)$$

5 Generic Programs

We can now use the universe of SPFs to write generic programs over SPFs.

5.1 Equality of RFs

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

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

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

where $\wedge$ is defined thus[3]:

$$\underline{\text{let}}\ \frac{f : \mathsf{Fin}\ n \to \mathsf{Bool}}{\wedge n\ f : \mathsf{Bool}}$$

$$\begin{array}{lll} \wedge n\ f & \Leftarrow & \underline{\text{rec}}\ n \\ \quad \wedge 0\ f & \Rightarrow & \mathsf{true} \\ \quad \wedge(1{+}\ n)\ f & \Rightarrow & (f\ \mathsf{fz})\&\&(\wedge n\ (\lambda i \Rightarrow f\ (\mathsf{fs}\ i))) \end{array}$$

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 might 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 $F\ A$.

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

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

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

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

$$\frac{p : P\ a}{\mathsf{here}\ p : \Diamond\mathsf{List}\ P\ A\ (a{::}as)} \qquad \frac{ps : \Diamond\mathsf{List}\ P\ as}{\mathsf{there}\ ps : \Diamond\mathsf{List}\ P\ (a{::}as)}$$

It seems that the idea of both $\Box$ and $\Diamond$ fit nicely with our abstraction of data types as SPFs and, indeed, we can give *generically* the types $\Box F\ P$ and $\Diamond F\ P$ for any SPFs in the appropriate form. Moreover, we can define a notion of generic dependent map using the generic $\Box$ and find that **map** has a dual for $\Diamond$, which we call **find**.

Firstly we define $\Box$, we assume that the *target* carries no information on its output index (it is indexed by One) and that is last input index to the data structure:

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

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

[3] note that the size n in the $\pi^{<\omega}$ case comes from the code T, there's not enough room to make this explicit but it is simple in practice

You'll notice the 'μ' 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:

$$\begin{array}{lllll} \Box_0\mathsf{Vz} & P\ (\mathsf{top}\ a) & \Rightarrow & P\ a \\ \Box_0(\mathsf{Vs}\ T) & P\ (\mathsf{pop}\ v) & \Rightarrow & \text{`1'} \\ \Box_{(1+\ n)}\mathsf{Vz} & P\ (\mathsf{top}\ v) & \Rightarrow & \mathsf{Vz} \\ \Box_{(1+\ n)}(\mathsf{Vs}\ T) & P\ (\mathsf{pop}\ v) & \Rightarrow & \mathsf{Vs}\ (\Box_n T\ P\ v) \\ \qquad\qquad \vdots & \qquad\qquad \vdots & & \\ \Box_n(\text{`}\mu\text{'}\ \ F) & P\ \ (\mathsf{in}\ v) & \Rightarrow & \text{`}\mu\text{'}\ \ (\Box_{(1+\ n)} F\ P\ v) \end{array}$$

and $\Box = \Box_0$.

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

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

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 $a : A$.

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

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

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

6 SPFs are Indexed Containers

We have seen how inductive families can be used to represent a wide range of advanced data types and how SPFs provide a grammar for defining such inductive families. By defining a universe of SPFs we further showed how generic programming over SPFs can be reduced to programming over the universe of SPFs. In this section, we relate SPFs to another device for defining and programming with inductive families, namely *indexed containers*.

Containers were introduced to capture the idea that concrete data types consist of memory locations where data can be stored and can thus be seen as a refinement of shapely types (**?**). For example, any element of the type $\mathsf{List}\ A$ of lists of A can be uniquely written as a natural number n given by the length of the list, together with a function $\mathsf{Fin}\ n \to A$ which labels each position within the list with an element from A:

$$\mathsf{List}A = \exists n : \mathsf{Nat} \Rightarrow \mathsf{Fin}\ n \to A$$

Abstracting from the example of lists, we may define a container in one variable to consist of a type of shapes S to represent the constructors of the data type and, for each constructor $s : S$, a type of positions where data for that constructor can be stored. Thus a container in one type variable is just an S-indexed family $P : S \to \star$. A container in n-type variables is much the same as we have a type of constructors S and then for each constructor, and each input type, a set of positions where data of that input type is stored. Thus we define

$$\underline{\text{data}}\ \frac{n : \mathsf{Nat}}{\mathsf{Cont}\ n : \star}\quad \underline{\text{where}}\quad \frac{S : \star \quad \dfrac{s : S \quad i : \mathsf{Fin}\ n}{P\ s\ i : \star}}{S \triangleleft P : \mathsf{Cont}\ n}$$

The extension of a container in n type variables is a functor $[\![S \triangleleft P]\!] : (\mathsf{Fin}\ n \to \star) \to \star$ and is defined by

$$[\![S \triangleleft P]\!]\vec{X} = \exists s : S \Rightarrow \forall i : \mathsf{Fin}\ n \Rightarrow P\ s\ i \to \vec{X}\ i$$

That is, given n-types $\vec{X}$, we construct an element of $[\![S \triangleleft P]\!]\vec{X}$ by choosing a constructor s from S and, for each input i, assign to each position in $P\ s\ i$ an element of $X\ i$. In previous work, we used containers as a foundational theory of data types using the metaphor that data types consist of shapes and positions where data can be stored. Amongst other results, we gave a simple representation theorem for polymorphic functions between containers, showed that all strictly positive types are containers and used containers as a basis for generic programming. While SPTs provide a natural way of defining generic programs where the structure of the type is important — such as equality, other operations, that operate only on the data contained in a structure, rather than the structure itself — such as generic map, are more naturally defined with the extension of a container. As an example consider the definition of generic map above and comparing to the equivalent definition using containers:

$$\underline{\text{let}}\ \frac{\begin{array}{l} C : \mathsf{Cont}\ n \\ \phi : \forall i : \mathsf{Fin}\ n \Rightarrow \vec{X}\ i \to \vec{Y}\ i \\ x : \mathsf{Ext}\ C\ \vec{X} \end{array}}{\mathbf{cMap}\ \phi\ x : \mathsf{Ext}\ C\ \vec{Y}}$$

$$\mathbf{cMap}\ \phi\ (s; f) \Rightarrow (s; (\lambda i; p \Rightarrow \phi\ i\ (f\ i\ p)))$$

A natural question in the light of the generalisation from strictly positive types to strictly positive families is to ask how one would add indexing structure to containers. Further, one would expect such a notion of indexing to be expressive enough to model all SPFs and to allow generic programming based upon an (indexed version) of the shapes and positions metaphor.

This section answers these questions. Firstly, just as both Cont and SPT are Nat-indexed families, so IC and SPF are similarly indexed. That is, an indexed container should expect a finite number of families as input and construct another indexed family as output. And, as with SPFs, these indexes could be arbitrary and so we define

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

The inhabitants of $\mathsf{IC}\ \vec{I}\ O$ are the actual indexed containers which, recall, will produce an O-indexed family. Thus the constructors S must no longer be a type but an O-indexed family $S : O \to \star$ where $S\ o$ is the type of constructors that will return something indexed by o. To incorporate indexing information into the positions, we have to assign to every constructor (which remember is an output index $o : O$ and constructor for that index $s : S\ o$) and every input $i : \mathsf{Fin}\ n$, an $\vec{I}!!i$-indexed family of positions. Thus we may complete the definition of $\mathsf{IC}\ \vec{I}\ O$ as follows:

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

The extension of an indexed container $(S \triangleleft P) : \mathsf{IC}\ \vec{I}\ O$ is a functor which should take as inputs an $\vec{I}$-tuple of indexed families and return an O-indexed family and thus will have type

$$(\forall i : \mathsf{Fin}\ n \Rightarrow (\vec{I}!!i) \to \star) \to (O \to \star)$$

This is extension is defined by

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

since, to produce an element of the O-indexed family with index o, we need to select a constructor which can do so, i.e. an expression of type $S\ o$. Then, for each input i, we must map each position in the $\vec{I}!!\ i$-indexed family $P\ o\ s\ i$ to an input from the actual family $\vec{X}\ i$.

Some examples may help. For example, let us consider the indexed container of type $\mathsf{IC}\ \vec{I}\ O$ which, on any input, returns the O-indexed family $O \to \star$ which maps o to the empty type. This indexed container clearly corresponds to the SPF '0' : $\mathsf{SPF}\ \vec{I}\ O$. To define this indexed container, the shapes must be a function which, for each $o : O$, returns the constructors for elements of the output. But since there is to be no output, this function should return the empty type of constructors. Thus the shapes are (using a wildcard pattern) $\lambda\_ \Rightarrow \mathsf{Zero}$. We must give a type of positions for every constructor but, since there are no constructors, this amounts to defining a function whose domain is the empty type. We write ! to denote any function whose domain is the empty type. Thus, the empty container is

$$(\lambda\_ \Rightarrow \mathsf{Zero}) \triangleleft (\lambda\_ \Rightarrow !)$$

Now let us consider the indexed container which, given appropriate inputs, returns the family with one element above every output index. This indexed container corresponds to the SPF '1' : $\mathsf{SPF}\ \vec{I}\ O$. Since we are to make an element for each output index $o : O$, we should have a constructor to do so. Thus we define the shapes to be the function which always returns the unit type. For the positions, note that we do not store data anywhere - if we did the extension of the container would depend upon the input and hence couldn't be correct. Thus this container is

$$(\lambda\_ \Rightarrow \mathsf{One}) \triangleleft (\lambda\_\ \_\ \_\ \_ \Rightarrow \mathsf{Zero})$$

We have seen how the SPFs '0' and '1' can be represented as indexed containers. In fact all SPFs can be represented as indexed containers as we shall now show. This means that indexed containers bear the same relationship to SPFs as as containers do to SPTs. Further, it means that indexed containers can act as a normal form for SPFs and that one can write

generic programs for SPFs by regarding them as indexed containers and manipulating their shapes and positions.

To show this result, we define a function

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

which maps an SPF to an indexed container. This is done by recursion on the structure of the SPF with the cases of the empty and unit SPF already having been done above

$$\begin{array}{l}\mathbf{ICont}\ T \Leftarrow \underline{\text{rec}}\ T \\ \quad \mathbf{ICont}\ \text{`0'} \Rightarrow (\lambda\_ \Rightarrow \mathsf{Zero}) \triangleleft (\lambda\_ \Rightarrow !) \\ \quad \mathbf{ICont}\ \text{`1'} \Rightarrow (\lambda\_ \Rightarrow \mathsf{One}) \triangleleft (\lambda\_\ \_\ \_\ \_ \Rightarrow \mathsf{Zero})\end{array}$$

Next, we consider the SPF Vz which, as an indexed container, takes a tuple of indexed families as input and returns the last one. So, given an output index $o : O$, there should be one position where we can store any data form the last input family at the index o. Thus we define

$$\begin{array}{l}\mathbf{ICont}\ \mathsf{Vz} \\ \quad \Rightarrow (\lambda\_ \Rightarrow \mathsf{One}) \\ \qquad \triangleleft \left(\lambda \begin{array}{cccccc} o & \_ & \mathsf{fz} & o' & \Rightarrow & (o = o') \\ o & \_ & (\mathsf{fs}\ i) & in & \Rightarrow & \mathsf{Zero} \end{array} \right)\end{array}$$

Here the expression in brackets is the function P which takes as its first two inputs an output index o and the only shape which can make some output for that index. The next input denotes which family data can be stored from. If this is the last family, then we want to store one position for data with index O and no positions otherwise - this is achieved by the test $o = o'$ whose value is One if o and o' are equal and Zero otherwise. On the other hand, if the family is not the last family, then we don't want to store any data from that family and so return the empty type of positions.

The situation with the SPF Vs T is dual. When viewed as a container, **ICont** (Vs T) should take as input a tuple of families, throw the last one away and then behave as **ICont** T on the remaining input families. Thus

$$\begin{array}{l}\mathbf{ICont}\ (\mathsf{Vs}\ T) \\ \quad \Rightarrow S \triangleleft \left(\lambda \begin{array}{cccccc} o & s & \mathsf{fz} & \_ & \Rightarrow & \mathsf{Zero} \\ o & s & (\mathsf{fs}\ i) & o' & \Rightarrow & P\ o\ s\ i\ o' \end{array} \right) \\ \qquad \underline{\text{where}}\ (S \triangleleft P) = \mathbf{ICont}\ T\end{array}$$

As we can see the constructors/shapes of **ICont** T and **ICont** (Vs T) are the same reflecting the fact that there are no new ways of building terms. The first line of the definition of the positions function says that we don't store any data from the last family - this reflects what we said before about discarding this family. On the other families, the number of positions required by **ICont** (Vs T) is clearly whatever is required by **ICont** T.

Next we consider the reindexing of containers which, recall from the beginning of section 3, changes the indexing structure by precomposing with a function. That is, the container **ICont** (`Δ'`f T) will produce an output with index o' by using the container **ICont** T to produce an output with index $f\ o'$. Thus the shapes for **ICont** (`Δ'`f T) are $\lambda o' \Rightarrow S\ (f\ o')$ where S are the shapes of **ICont** T. Since we are not changing the data stored, just the index of the constructors, the positions required by a constructor in **ICont** (`Δ'`f T) will just be the positions required by the constructor in **ICont** T. Thus

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

Next we consider the container **ICont** (`Σ'`f T). Again, recall the construction of `Σ'`-families at the beginning of section 3 which shows that **ICont** (`Σ'`f T) will produce an output indexed by $o' : O'$ by producing output of **ICont** T indexed by a *single* o such that $f\ o = o'$. Thus the shapes which can produce something indexed by o' are triples consisting of a single o such that $f\ o = o'$, a proof that $f\ o = o'$ and a shape of **ICont** T which can produce something indexed by o. This is captured in the following definition:

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

As with `Δ'`-containers, we are not changing the data stored by a shape, simply changing the index of the output. Thus the positions stored in the shape $(o;\ \mathsf{refl};\ s)$ of the indexed container **ICont** (`Σ'`f T) will be the same as the positions of the shape s in the indexed container T. This is also captured in the formula above.

The construction of `Π'`-families at the beginning of section 3 is the same as that of `Σ'`-families except that it will produce an output indexed by $o' : O'$ by producing an output indexed by by *every* o such that $f\ o = o'$. Thus the shapes of **ICont** (`Π'`f T) container will be the same as for the **ICont** (`Σ'`f T) container except that the $\exists$-quantifier will be replaced by the $\forall$-quantifier. This is seen below:

$$\begin{array}{l}\mathbf{ICont}\ (\text{`}\Pi\text{'}f\ T) \\ \quad \Rightarrow (\lambda o' \Rightarrow \forall o : O \Rightarrow (f\ o = o') \to S\ o) \\ \qquad \triangleleft \left(\begin{array}{l} \lambda o'\ f\ i\ in \Rightarrow \\ \quad \exists o : O;\ p : (f\ o = o') \Rightarrow \\ \qquad P\ o\ (f\ o\ p)\ i\ in) \end{array} \right) \\ \qquad \underline{\text{where}}\ (S \triangleleft P) = \mathbf{ICont}\ T\end{array}$$

This formula also shows that, if f is a shape which produces an output indexed by o', then a position for this shape will be a position for the shape given by f for any o such that $f\ o = o'$.

Our penultimate case is that of SPFs `Tag'` f where $f : \mathsf{Fin}\ n \Rightarrow \mathsf{SPF}\ \vec{I}\ O$. In this case, **ICont** (`Tag'` f) should intuitively be the n-fold sum of the containers associated to the SPFs given by f. Thus the shapes for **ICont** (`Tag'` f) should be the fum of the shapes of the containers **ICont** ($f\ i$) and the positions for one of these shapes in **ICont** (`Tag'` f) will be the positions for that shape in **ICont** ($f\ i$). Thus:

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

Finally, we come to the most complex part of the translation, namely that for the fixed point SPFs. This is as follows:

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

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.

We conclude by pointing out that the translation of SPFs to indexed containers preserves the semantics of both SPFs (via telescopes) and indexed containers (using slice categories). While there is not space to establish this here, the translation was given in such a way as to make this obviously the case.

7 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 (Altenkirch & McBride 2006) 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 data types 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

Abbott, M., Altenkirch, T. & Ghani, N. (2005), 'Containers - constructing strictly positive types', *Theoretical Computer Science* **342**, 3–27. Applied Semantics: Selected Topics.

Abbott, M., Altenkirch, T., Ghani, N. & McBride, C. (2005), '∂ for data', *Fundamentae Informatica* **65**(1,2), 1 – 28. Special Issue on Typed Lambda Calculi and Applications 2003.

Altenkirch, T., Ghani, N., Hancock, P., McBride, C. & Morris, P. (2006), 'Indexed containers', Manuscript, available online.

Altenkirch, T. & McBride, C. (2006), 'Towards observational type theory', Manuscript, available online.

Altenkirch, T., McBride, C. & McKinna, J. (2005), 'Why dependent types matter', Manuscript, available online.

Benke, M., Dybjer, P. & Jansson, P. (2003), 'Universes for generic programs and proofs in dependent type theory', *Nordic Journal of Computing* **10**, 265–269.

Dybjer, P. & Setzer, A. (2001), Indexed induction-recursion., *in* R. Kahle, P. Schroeder-Heister & R. F. Stärk, eds, 'Proof Theory in Computer Science', Vol. 2183 of *Lecture Notes in Computer Science*, Springer, pp. 93–113.

Martin-Löf, P. (1984), *Intuitionistic Type Theory*, Bibliopolis·Napoli.

McBride, C. (2004), 'Epigram'. `http://www.e-pig.org/`.

McBride, C. & McKinna, J. (2004), 'The view from the left', *Journal of Functional Programming* **14**(1).

Morris, P., Altenkirch, T. & McBride, C. (2006), Exploring the regular tree types, *in* C. P.-M. Jean-Christophe Filliatre & B. Werner, eds, 'Types for Proofs and Programs (TYPES 2004)', Lecture Notes in Computer Science.

Nordström, B., Petersson, K. & Smith, J. (1990), *Programming in Martin-Löf's Type Theory: an introduction*, Oxford University Press.
URL: *http://www.cs.chalmers.se/Cs/Research/Logic/book/*