International Journal of Foundations of Computer Science

A Universe of Strictly Positive Families

PETER MORRIS

and

THORSTEN ALTENKIRCH

and

NEIL GHANI

School of Computer Science and IT
Jubilee Campus, Wollaton Road
Nottingham, NG8 1BB, UK

Received (received date)
Revised (revised date)
Communicated by Editor's name

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 and that generic programs can naturally be written over the universe of SPFs.

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

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

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 $\mathsf{Fin}\ (\mathsf{s}\ n)$. In effect the type $\mathsf{Fin}\ (\mathsf{s}\ (\mathsf{s}\ (\mathsf{s}\ \mathsf{z})))$ contains elements that look much like 0,1 and 2: 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 $\mathsf{Fin}\ (\mathsf{s}\ n)$ 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 [20, 24] 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.

- We give a full implementation of all our constructions in Epigram which is a dependently typed programming language [22, 21, 6]. Not only does Epigram provide a language to implement SPFs, but SPFs are also sufficiently expressive to provide a meta-language 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: Generic programming within the functional programming, especially in the language Haskell, had many successes with a succession of systems such as PolyP [16, 17] and Generic Haskell [8, 19] proving very popular the latter being based on the work of Hinze [12], who with Peyton-Jones [13] implemented the widely employed 'deriving' mechanism for Haskell from a similar idea. It is our intention to show that while these systems are extensions on top of an existing language with dependent types generic programming can be implemented inside the system. Indeed Altenkirch and McBride [4] showed how to simulate Generic Haskell like generics within an Epigram-like system. This paper serves to show that similar techniques extend to dependent data-structures. More recent developments in generic programming with Haskell have exploited the addition of generalized algebraic data-types (GADTs) [18] to some versions of the language to implement some of the techniques used by the dependently typed programming community, the Scrap Your Boilerplate [14, 15] work and Weirich's RepLib [25] show some promise, but ultimately the benefits from having full dependent types are great, the constructions in the first half of this paper deal with Haskell-like data-types but cannot be translated into a language with only GADTs.

This work is an extension of previous work with a universe of regular tree types [23], a proper sub-set of the strictly positive types. We will revisit this work in the first section of this paper. Others have exploited universes for generic programming purposes, for instance Dybjer and Setzer's axiomatization's of induction recursion [10, 11] have been the inspiration for advances in generic programming [7] within the AGDA [9] system.

Structure of the Paper: The rest of the paper is structured as follows: Since our use of dependent types and universes 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.

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:

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

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.

2.1. 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 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:

$$\underline{\text{data}} \quad \frac{}{\mathsf{Nat} : \star} \quad \underline{\text{where}} \quad \frac{}{\mathsf{z} : \mathsf{Nat}} \quad \frac{n : \mathsf{Nat}}{\mathsf{s}\ n : \mathsf{Nat}}$$

$$\underline{\text{let}} \ \frac{m, n : \mathsf{Nat}}{\mathbf{plus}\ m\ n : \mathsf{Nat}} \ ; \ \begin{array}{l} \mathbf{plus}\ m\ n \Leftarrow \underline{\text{rec}}\ m \\ \quad \mathbf{plus} \quad \mathsf{z} \quad n \Rightarrow n \\ \quad \mathbf{plus}\ (\mathsf{s}\ m)\ n \Rightarrow \mathsf{s}\ (\mathbf{plus}\ m\ n) \end{array}$$

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 $\mathsf{Fin}\ \mathsf{z}$ 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.

$$\underline{\text{data}}\quad \frac{n : \mathsf{Nat}}{\mathsf{Fin}\ n : \star}\quad \underline{\text{where}}\quad \frac{}{\mathsf{fz} : \mathsf{Fin}\ (\mathsf{s}\ n)}\quad \frac{i : \mathsf{Fin}\ n}{\mathsf{fs}\ i : \mathsf{Fin}\ (\mathsf{s}\ n)}$$

$$\underline{\text{data}}\quad \frac{A : \star \quad n : \mathsf{Nat}}{\mathsf{Vec}\ n\ A : \star}\quad \underline{\text{where}}\quad \frac{}{\varepsilon : \mathsf{Vec}\ \mathsf{z}\ A}\quad \frac{as : \mathsf{Vec}\ n\ A \quad a : A}{as ▷ a : \mathsf{Vec}\ (\mathsf{s}\ n)\ A}$$

$$\underline{\text{let}}\ \frac{as : \mathsf{Vec}\ n\ A \quad i : \mathsf{Fin}\ n}{as !! i : A}\ ;\ \begin{array}{l} as !! i \Leftarrow \underline{\text{rec}}\ i \\ \quad (as ▷ a) !! \mathsf{fz} \quad\ \Rightarrow a \\ \quad (as ▷ a) !! (\mathsf{fs}\ i) \Rightarrow as !! i \end{array}$$

Any universe to capture the strictly positive families would need to include these examples, but shouldn't be limited to Nat indexed families.

2.2. A Universe for Strictly Positive Types

$$\underline{\text{data}}\quad \frac{n : \mathsf{Nat}}{\mathsf{SPT}\ n : \star}\quad \underline{\text{where}}\quad \frac{}{\mathsf{vz} : \mathsf{SPT}\ (\mathsf{s}\ n)}\quad \frac{T : \mathsf{SPT}\ n}{\mathsf{vs}\ T : \mathsf{SPT}\ (\mathsf{s}\ n)}$$

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

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

$$\underline{\text{data}}\quad \frac{n : \mathsf{Nat}}{\mathsf{Tel}\ n : \star}\quad \underline{\text{where}}\quad \frac{}{\varepsilon : \mathsf{Tel}\ \mathsf{z}}\quad \frac{\vec{T} : \mathsf{Tel}\ n \quad T : \mathsf{SPT}\ n}{\vec{T} ▷ T : \mathsf{Tel}\ (\mathsf{s}\ n)}$$

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

$$\frac{e : \mathsf{El}\ \vec{T}\ T}{\mathsf{top}\ e : \mathsf{El}\ (\vec{T} ▷ T)\ \mathsf{vz}}\qquad \frac{e : \mathsf{El}\ \vec{T}\ T}{\mathsf{pop}\ e : \mathsf{El}\ (\vec{T} ▷ S)\ (\mathsf{vs}\ T)}\qquad \frac{f : K \rightarrow \mathsf{El}\ \vec{T}\ T}{\mathsf{fun}\ f : \mathsf{El}\ \vec{T}\ (K\ \text{`}\rightarrow\text{'}\ T)}$$

$$\frac{}{\mathsf{void} : \mathsf{El}\ \vec{T}\ \text{`1'}}\qquad \frac{s : \mathsf{El}\ \vec{T}\ S}{\mathsf{inl}\ s : \mathsf{El}\ \vec{T}\ (S\ \text{`+'}\ T)}\qquad \frac{t : \mathsf{El}\ \vec{T}\ T}{\mathsf{inr}\ t : \mathsf{El}\ \vec{T}\ (S\ \text{`+'}\ T)}$$

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

Figure 1: The SPT Universe

The traditional, informal, definition of SPTs given above is insufficent when we want to write generic programs which manipulate SPTs.

Codes for SPTs: To address this issue, 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 - with vz and vs as the zero and successor for variables. The empty type 0 and unit type 1 are represented by ‘0’[a] 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.

$$\underline{\text{let}}\ \text{‘Nat’} : \mathsf{SPT}\ \mathsf{z}\ ;\ \text{‘Nat’} \Rightarrow \text{‘}\mu\text{’}\ (\text{‘1’}\ \text{‘+’}\ \mathsf{vz})$$

$$\underline{\text{let}}\ \text{‘List’} : \mathsf{SPT}\ (\mathsf{s}\,\mathsf{z})\ ;\ \text{‘List’} \Rightarrow \text{‘}\mu\text{’}\ (\text{‘1’}\ \text{‘+’}\ ((\mathsf{vs}\,\mathsf{vz})\ \text{‘}\times\text{’}\ \mathsf{vz}))$$

$$\underline{\text{let}}\ \text{‘RT’} : \mathsf{SPT}\ (\mathsf{s}\,\mathsf{z})$$
$$\text{‘RT’} \Rightarrow \text{‘}\mu\text{’}\ \left(\begin{array}{l}(\mathsf{vs}\,\mathsf{vz}) \\ \quad \text{‘}\times\text{’}\ (\text{‘}\mu\text{’}\ ((\text{‘1’}\ \text{‘+’}\ ((\mathsf{vs}\,\mathsf{vz})\ \text{‘}\times\text{’}\ \mathsf{vz}))))\end{array}\right)$$

$$\underline{\text{let}}\ \text{‘Ord’} : \mathsf{SPT}\ \mathsf{z}$$
$$\text{‘Ord’} \Rightarrow \text{‘}\mu\text{’}\ (\text{‘1’}\ \text{‘+’}\ (\mathsf{vz}\ \text{‘+’}\ (\mathsf{Nat}\ \text{‘}\rightarrow\text{’}\ \mathsf{vz})))$$

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 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.

2.3. 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

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

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.

$$\underline{\text{data}}\ \frac{\vec{S}, \vec{T} : \mathsf{Tel}\ n}{\mathsf{Morph}\ \vec{S}\ \vec{T} : \star}\ \underline{\text{where}}\ \frac{\phi : \mathsf{Morph}\ \vec{S}\ \vec{T} \quad f : \mathsf{El}\ \vec{S}\ S \to \mathsf{El}\ \vec{T}\ T}{\mathsf{mF}\ \phi\ f : \mathsf{Morph}\ (\vec{S} :\!\!: S)\ (\vec{T} :\!\!: T)}$$

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

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

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

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

In our work on the regular tree types [23], 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:

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

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. We call a indexed family of types $F : O \to \star$ an O-indexed family — 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.

Instead of coding $O \to \star$, we choose to keep the function spaces and replace $\star$ by a universe of indexed strictly positive types ISPT. The type of ISPTs will be similar to that for SPTs, except each input will require an index type:

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

Families are represented as functions from the output index to an ISPT over the families input index types:

$$\underline{\text{let}}\ \frac{\vec{I} : \mathsf{Vec}\ \star\ n \quad O : \star}{\mathbf{SPF}\ \vec{I}\ O : \star}\ ;\ \mathbf{SPF}\ \vec{I}\ O \Rightarrow O \to \mathsf{ISPT}\ \vec{I}$$

Our intuition is that each element of $\mathsf{SPF}\ \vec{I}\ O$ will represent an SPF which takes as input families whose indexes are in $\vec{I}$ and will return a family indexed by O. We choose to create these definitions mutually, so we can refer to **SPF** in the constructors of ISPT.

As with SPTs, we also define an interpretation for ISPTs which gives rise to an interpretation for families. As before the crucial ingredient in for this interpretation is the type of telescopes Tel which contain families indexed by the types contained in $\vec{I}$:

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

The type of the interpretation is then:

$$\underline{\text{data}}\ \frac{T : \mathsf{ISPT}\ \vec{I} \quad \vec{T} : \mathsf{Tel}\ \vec{I}}{[\![T]\!]\ \vec{T} : \star}$$

for ISPTs and we also lift this to families in the obvious way:

$$\underline{\text{let}}\ \frac{F : \mathbf{SPF}\ \vec{I}\ O \quad \vec{T} : \mathsf{Tel}\ \vec{I} \quad o : O}{[\![F]\!]\ \vec{T}\ o : \star}\ ;\ [\![F]\!]\ \vec{T}\ o \Rightarrow [\![F\ o]\!]\ \vec{T}$$

Again for convenience we refer to this function in the definition of $[\![-]\!]$.

The base constructors of ISPT are the constant types '0' and '1' as before, these give rise to families which are empty and contain one element at each possible index.

We can exploit the fact that codes for families are a function space to transform codes by composition with out adding syntax, given a code $F : \mathbf{SPF}\ \vec{I}\ O$ and a function $f : O' \to O$ we can form $F.f : \mathbf{SPF}\ \vec{I}\ O'$. The meaning of this is clear, an element of this new family indexed at $o' : O'$ is an element of F indexed at $(f\ o')$ i.e. $[\![F.f]\!]\ \vec{I}\ o'$ is definitional equal to $[\![F]\!]\ \vec{I}\ (f\ o')$.

An example of a construction like this can be seen in the abs constructor for Lam which could alternatively been defined by composing the recursive reference to Lam with s:

$$\frac{b\ :\ (\mathsf{Lam.s})\ n}{\mathsf{abs}\ b\ :\ \mathsf{Lam}\ n}$$

While composition has no syntax, there are two related constructions which will and they deal with the cases that arise if we combine an O indexed family with a function of type $O \to O'$ to create an O' indexed family. There are two ways to do this and we denote the 'Σ' and 'Π' since they correspond to dependent tuples and dependent functions. To see an example of the first consider the fs constructor of Fin:

$$\frac{i\ :\ \mathsf{Fin}\ n}{\mathsf{fs}\ i\ :\ \mathsf{Fin}\ (\mathsf{s}\ n)}$$

Here, unlike the abs example we are placing a requirement on the index to the family we are creating, the function is in the opposite direction, in some sense. We can alter this definition so that it applies to all indexes n : Nat by using Epigram's dependent tuples ($\exists$) and built-in equality:

$$\frac{i\ :\ \exists n' : \mathsf{Nat} \Rightarrow (\mathsf{s}\ n' = n) \times \mathsf{Fin}\ n'}{\mathsf{fs}\ i\ :\ \mathsf{Fin}\ n}$$

We formalise this construction for SPFs by adding this constructor to ISPT:

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

The interpretation reflects the idea contained in the alternative fs construction:

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

While in the fs example there is only one possible value for n' in the general construction this is not the case, there is often a choice of possible indexes that satisfy the equation, where 'Σ' selects on one these possibilities 'Π' considers all

such possibilities:

$$\frac{f \;:\; O \to O' \quad T \;:\; \mathbf{SPF}\ \vec{I}\ O \quad o' \;:\; O'}{\text{`}\Pi\text{'}f\ T\ o' \;:\; \mathsf{ISPT}\ \vec{I}}$$

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

We will come across examples of ‘Π’ families later. Note that ‘Σ’, composition and ‘Π’ are very closely related, in fact ‘Σ’ is the left adjoint of composition and ‘Π’ is its right adjoint.

We can also take the fixed point of a family as before, so we have this construct of ISPT:

$$\frac{F \;:\; \mathbf{SPF}\ (\vec{I} :: O)\ O \quad o \;:\; O}{\text{`}\mu\text{'}\ F\ o \;:\; \mathsf{ISPT}\ \vec{I}}$$

$$\frac{v \;:\; [\![T]\!](\vec{T} :: (\text{`}\mu\text{'}\ T))\ o}{\mathsf{in}\ v \;:\; [\![ \text{`}\mu\text{'}\ T\ o]\!]\, \vec{T}}$$

Note that the family we add to the interpreting telescope leaves off the top index since ‘μ’ F o is an ISPT $\vec{I}$ but ‘μ’ F is and SPF $\vec{I}$ O, this is a useful pattern and also applies to ‘Π’ and ‘Σ’.

Finanlly, we add variables, these resemble the the simple typed SPT variables, except that Tel now contains families, so at a vz code we must provide the index at which to interpret the top of the context:

$$\frac{i \;:\; I}{\mathsf{vz}\ i \;:\; \mathsf{ISPT}\ (\vec{I} :: I)} \qquad \frac{T \;:\; \mathsf{ISPT}\ \vec{I}}{\mathsf{vs}\ T \;:\; \mathsf{ISPT}\ (\vec{I} :: I)}$$

$$\frac{v \;:\; [\![T]\!]\, \vec{T}\ i}{\mathsf{top}\ v \;:\; [\![\mathsf{vz}\ i]\!](\vec{T} :: T)} \qquad \frac{v \;:\; [\![T]\!]\, \vec{T}}{\mathsf{pop}\ v \;:\; [\![\mathsf{vs}\ T]\!](\vec{T} :: S)}$$

The full definition of SPF is given in figure 2.

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.

$$\underline{\text{let}}\ \frac{A, B \;:\; \mathsf{ISPT}\ \vec{I}}{A\ \text{`}{+}\text{'}\ B \;:\; \mathsf{ISPT}\ \vec{I}}\ ;\quad A\ \text{`}{+}\text{'}\ B \;\Rightarrow\; \text{`}\Sigma\text{'}(\lambda x \Rightarrow ())\ \begin{pmatrix} \lambda & \mathsf{fz} & \Rightarrow A \\ & (\mathsf{fs}\ \mathsf{fz}) & \Rightarrow B \end{pmatrix}\ ()$$

$$\underline{\text{let}}\ \frac{a \;:\; [\![A]\!]\, \vec{T}}{\mathbf{inl}\ a \;:\; [\![A\ \text{`}{+}\text{'}\ B]\!]\, \vec{T}}\ ;\quad \mathbf{inl}\ a \;\Rightarrow\; \sigma\ (\mathsf{fz}; \mathsf{refl}; a)$$

$$\underline{\text{let}}\ \frac{b \;:\; [\![B]\!]\, \vec{T}}{\mathbf{inr}\ b \;:\; [\![A\ \text{`}{+}\text{'}\ B]\!]\, \vec{T}}\ ;\quad \mathbf{inr}\ b \;\Rightarrow\; \sigma\ (\mathsf{fs}\ \mathsf{fz}; \mathsf{refl}; b)$$

ISPT and **SPF** codes:

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

$$\mathbf{SPF}\ \vec{I}\ O\ \Rightarrow\ O \to \mathsf{ISPT}\ \vec{I}$$

$$\frac{i\ :\ I}{\mathsf{vz}\ i\ :\ \mathsf{ISPT}\ (\vec{I} :\!: I)} \qquad \frac{T\ :\ \mathsf{ISPT}\ \vec{I}}{\mathsf{vs}\ T\ :\ \mathsf{ISPT}\ (\vec{I} :\!: I)} \qquad \frac{}{\text{`0'}, \text{`1'}\ :\ \mathsf{ISPT}\ \vec{I}}$$

$$\frac{f\ :\ O \to O' \quad T : \mathbf{SPF}\ \vec{I}\ O \quad o'\ :\ O'}{\text{`}\Sigma\text{'}f\ T\ o', \text{`}\Pi\text{'}f\ T\ o'\ :\ \mathsf{ISPT}\ \vec{I}} \qquad \frac{F\ :\ \mathbf{SPF}\ (\vec{I} :\!: O)\ O \quad o\ :\ O}{\text{`}\mu\text{'}\ F\ o\ :\ \mathsf{ISPT}\ \vec{I}}$$

The Interpretation of ISPT and **SPF**:

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

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

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

$$[\![F]\!]\ \vec{T}\ o\ \Rightarrow\ [\![F\ o]\!]\ \vec{T}$$

$$\frac{v\ :\ [\![T]\!]\ \vec{T}\ i}{\mathsf{top}\ v\ :\ [\![\mathsf{vz}\ i]\!]\ (\vec{T} :\!: T)} \qquad \frac{v\ :\ [\![T]\!]\ \vec{T}}{\mathsf{pop}\ v\ :\ [\![\mathsf{vs}\ T]\!]\ (\vec{T} :\!: S)} \qquad \frac{}{\mathsf{void}\ :\ [\![\text{`1'}]\!]\ \vec{T}}$$

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

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

Figure 2: The **SPF** Universe

where

$$\begin{pmatrix} \lambda & \mathsf{fz} & \Rightarrow & A \\ & \mathsf{fs}\ \mathsf{fz} & \Rightarrow & B \end{pmatrix}$$

denotes the function whose domain is $\mathsf{Fin}\ (\mathsf{s}\ (\mathsf{s}\ \mathsf{z}))$ and which returns A on fz and B

on $\mathsf{fs}\,\mathsf{fz}$. For the products we have

$$\underline{\mathsf{let}}\ \frac{A, B : \mathsf{ISPT}\ \vec{I}}{A\ \text{`}\times\text{'}\ B : \mathsf{ISPT}\ \vec{I}}\ ;\ \ A\ \text{`}\times\text{'}\ B\ \Rightarrow\ \text{`}\Pi\text{'}\mathbf{fst}\ \left(\lambda\ \begin{array}{lll}\mathsf{fz} & \Rightarrow & A\\ (\mathsf{fs}\,\mathsf{fz}) & \Rightarrow & B\end{array}\right)\ ()$$

$$\underline{\mathsf{let}}\ \frac{a : [\![A]\!]\,\vec{T}\ o \quad b : [\![B]\!]\,\vec{T}\ o}{\mathbf{pair}\ a\ b : [\![A\ \text{`}\times\text{'}\ B]\!]\,\vec{T}\ o}\ ;\ \ \mathbf{pair}\ a\ b\ \Rightarrow\ \pi\left(\lambda\ \begin{array}{llll}\mathsf{fz} & \mathsf{refl} & \Rightarrow & a\\ (\mathsf{fs}\,\mathsf{fz}) & \mathsf{refl} & \Rightarrow & b\end{array}\right)$$

For convenience we also define the family of variables **var**:

$$\underline{\mathsf{let}}\ \frac{\vec{I} : \mathsf{Vec}\ n\ \star \quad i : \mathsf{Fin}\ n \quad x : \vec{I}!!i}{\mathbf{var}\ i\ x : \mathsf{ISPT}\ \vec{I}}\ ;\ \ \begin{array}{l}\mathbf{var}\ i\ x \Leftarrow \underline{\mathsf{rec}}\ i\ \{\\ \quad \mathbf{var}\ \mathsf{fz}\ x\ \Rightarrow\ \mathsf{vz}\ x\\ \quad \mathbf{var}\ (\mathsf{fs}\ i')\ \Rightarrow\ \mathsf{vs}\,(\mathbf{var}\ i'\ x)\ \}\end{array}$$

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^b:

$$\underline{\mathsf{let}}\ \text{`}\mathbf{Fin}\text{'} : \mathbf{SPF}\ []\ \mathsf{Nat}\ ;\ \ \text{`}\mathbf{Fin}\text{'} \Rightarrow\ \text{`}\mu\text{'}\ \ (\text{`}\Sigma\text{'}\mathsf{s}\ (\lambda n \Rightarrow\ \text{`}1\text{'}\ \text{`}+\text{'}\ \mathbf{var}\ \mathsf{fz}\ n'))$$

$$\begin{array}{l}\underline{\mathsf{let}}\ \text{`}\mathbf{Vec}\text{'} : \mathbf{SPF}\ [\mathsf{One}]\ \mathsf{Nat}\\ \quad \text{`}\mathbf{Vec}\text{'}\ \Rightarrow\ \text{`}\mu\text{'}\ \left(\lambda n \Rightarrow\ \begin{array}{l}(\text{`}\Sigma\text{'}(\lambda x : \mathsf{One} \Rightarrow \mathsf{z})\ (\mathbf{const}\ \text{`}1\text{'})\ n)\\ \quad \text{`}+\text{'}\quad (\text{`}\Sigma\text{'}\mathsf{s}\ (\lambda n' \Rightarrow (\mathbf{var}\ (\mathsf{fs}\,\mathsf{fz})\ ())\ \text{`}\times\text{'}\ (\mathbf{var}\ \mathsf{fz}\ n'))\ n)\end{array}\right)\end{array}$$

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 $\text{`}\Sigma\text{'}(\mathbf{const}\ \mathsf{z})$ forces the empty vector to always have index zero; in the ('$::$') case, $\text{`}\Sigma\text{'}\mathsf{s}$ forces the vector $a\ \text{`}::\text{'}\ as$ to have index/length $\mathsf{s}\ n$ if as has index/length n. We can encode values of the finite sets and vectors using generic constructors such as these:

$$\underline{\mathsf{let}}\ \frac{}{\text{`}\mathbf{fz}\text{'}_n : [\![\text{`}\mathbf{Fin}\text{'}]\!]\,[]\ (\mathsf{s}\ n)}$$
$$\text{`}\mathbf{fz}\text{'}_n\ \Rightarrow\ \mathsf{in}\ (\sigma\ (n; \mathsf{refl}; \mathbf{inl}\ \mathsf{void}))$$

$$\underline{\mathsf{let}}\ \frac{i : [\![\text{`}\mathbf{Fin}\text{'}]\!]\,[]\ n}{\text{`}\mathbf{fs}\text{'}_n\ i : [\![\text{`}\mathbf{Fin}\text{'}]\!]\,[]\ (\mathsf{s}\ n)}$$
$$\text{`}\mathbf{fs}\text{'}_n\ i\ \Rightarrow\ \mathsf{in}\ (\sigma\ (n; \mathsf{refl}; \mathbf{inl}\ (\mathsf{top}\ i)))$$

$$\underline{\mathsf{let}}\ \frac{A : \mathsf{SPF}\ []\ \mathsf{One}}{\text{`}\varepsilon\text{'} : [\![\text{`}\mathbf{Vec}\text{'}]\!]\,[A]\ \mathsf{z}}$$
$$\text{`}\varepsilon\text{'}\ \Rightarrow\ \mathsf{in}\ (\text{`}\mathbf{inl}\text{'}\ (\sigma\ ((); \mathsf{refl}; \mathsf{void})))$$

$$\underline{\mathsf{let}}\ \frac{as : [\![\text{`}\mathbf{Vec}\text{'}]\!]\,[A]\ n \quad a : [\![A]\!]\,[]\ ()}{(as\ \text{`}::\text{'}_n\ a) : [\![\text{`}\mathbf{Vec}\text{'}]\!]\,[A]\ (\mathsf{s}\ n)}$$
$$(as\ \text{`}::\text{'}_n\ a)\ \Rightarrow\ \mathsf{in}\ (\text{`}\mathbf{inr}\text{'}\ (\sigma\ (n; \mathsf{refl}; \text{`}\mathbf{pair}\text{'}\ (\mathsf{pop}\ (\mathsf{top}\ a))\ (\mathsf{top}\ as))))$$

[b] Since we have to treat types uniformly the type A becomes a family whose index carries no information.

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.

$$\underline{\text{let}}\ \ \text{`}\mathbf{Lam}\text{'} : \mathbf{SPF}\ []\ \mathsf{Nat}$$
$$\text{`}\mathbf{Lam}\text{'} \Rightarrow \text{`}\mu\text{'}\ \left(\lambda n \Rightarrow \begin{array}{l}((\mathsf{vs}\ (\text{`}\mathbf{Fin}\text{'}\ n))\ \text{`}+\text{'}\ ((\mathbf{var}\ \mathsf{fz}\ n)\ \text{`}\times\text{'}\ (\mathbf{var}\ \mathsf{fz}\ n)))\\ \text{`}+\text{'}\ (\mathbf{var}\ \mathsf{fz}\ (\mathsf{s}\ n))\end{array}\right)$$

$$\underline{\text{let}}\ \ \frac{i : [\![\text{`}\mathbf{Fin}\text{'}]\!]\ []\ n}{\text{`}\mathbf{var}\text{'}\ i : [\![\text{`}\mathbf{Lam}\text{'}]\!]\ []\ n}$$
$$\text{`}\mathbf{var}\text{'}\ i\ \Rightarrow \mathsf{in}\ (\text{`}\mathbf{inl}\text{'}\ (\text{`}\mathbf{inl}\text{'}\ (\mathsf{pop}\ i)))$$

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

$$\underline{\text{let}}\ \ \frac{f : [\![\text{`}\mathbf{Lam}\text{'}]\!]\ []\ (\mathsf{s}\ n)}{\text{`}\mathbf{abs}\text{'}\ f : [\![\text{`}\mathbf{Lam}\text{'}]\!]\ []\ n}$$
$$\text{`}\mathbf{abs}\text{'}\ f\ \Rightarrow \mathsf{in}\ (\text{`}\mathbf{inr}\text{'}\ (\mathsf{top}\ f))$$

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:

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

The translation of this definition is not completely straightforward as the type Vec appears inside NBrTree. This *composition* of types can be seen as a generalisation of Monadic substitution; 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 (here Vec) with the code for the payload type (NBRTree). Note that just as $(>\!>\!=)$ is the bind of the ISPT 'monad' so **var** is the

return. The definition of composition is given as follows:

$$\underline{\text{let}}\ \frac{\vec{I} : \mathsf{Vec}\ m\ \star \quad \vec{J} : \mathsf{Vec}\ n\ \star \qquad S : \mathsf{ISPT}\ \vec{I} \quad \vec{T} : \forall i{:}\mathsf{Fin}\ n \Rightarrow \mathbf{SPF}\ \vec{J}\ (\vec{I}\,!!\,i)}{S \gg\!= \vec{T} : \mathsf{ISPT}\ \vec{J}}$$

$$\begin{array}{l}
S \gg\!= \vec{T} \Leftarrow \underline{\text{rec}}\ S\ \{ \\
\quad \text{`0'} \gg\!= \vec{T} \ \Rightarrow\ \text{`0'} \\
\quad \text{`1'} \gg\!= \vec{T} \ \Rightarrow\ \text{`1'} \\
\quad (\text{`}\Sigma\text{'}\ f\ F\ o') \gg\!= \vec{T} \ \Rightarrow\ \text{`}\Sigma\text{'}\ f\ (\lambda o \Rightarrow (F\ o) \gg\!= \vec{T})\ o' \\
\quad (\text{`}\Pi\text{'}\ f\ F\ o') \gg\!= \vec{T} \ \Rightarrow\ \text{`}\Pi\text{'}\ f\ (\lambda o \Rightarrow (F\ o) \gg\!= \vec{T})\ o' \\
\quad (\mathsf{vz}\ i) \gg\!= \vec{T} \ \Rightarrow\ \vec{T}\ \mathsf{fz}\ i \\
\quad (\mathsf{vs}\ T) \gg\!= \vec{T} \ \Rightarrow\ T \gg\!= (\vec{T}.\mathsf{fs}) \\
\quad (\text{`}\mu\text{'}\ F\ o) \gg\!= \vec{T} \ \Rightarrow\ \text{`}\mu\text{'}\ (\lambda o' \Rightarrow (F\ o') \gg\!= \left(\lambda \begin{array}{ll} \mathsf{fz} & o'' \Rightarrow \mathsf{vz}\ o'' \\ (\mathsf{fs}\ j)\ i & \Rightarrow \mathsf{vs}\ (\vec{T}\ j\ i) \end{array}\right))\ o \\
\}
\end{array}$$

To simplify the construction we replace all variables simultaneously and switch contexts as we do, the value $\vec{T}$ explains how to replace a variable in S a type in context $\vec{I}$ with a new type in the context $\vec{J}$. We then perform the substitution by traversing S, when we hit a variable we look up the appropriate code, when we hit a weakening we simply forget the replacement for the top variable. When we traverse under a 'μ' binder we extend $\vec{T}$ to leave the new top variable intact, the other terms in $\vec{T}$ must be weakened to account for the new variable in the context, note that this is only possible because we allow weakenings of arbitary terms, without the vs constructor this weakening would have to be defined mutually with $(\gg\!=$. '0', '1', 'Π' and 'Σ' are all structural.

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

$$\begin{array}{l}
\underline{\text{let}}\ \text{`}\mathbf{NBrTree}\text{'} : \mathbf{SPF}\ [\mathsf{One}]\ \mathsf{Nat} \\
\quad \text{`}\mathbf{NBrTree}\text{'} \ \Rightarrow\ \text{`}\mu\text{'}\ \left(\lambda n \Rightarrow \begin{array}{l} (\mathbf{var}\ (\mathsf{fs}\ \mathsf{fz})\ ()) \\ \text{`}+\text{'}\ ((\text{`}\mathbf{Vec}\text{'}\ n) \gg\!= (\lambda \mathsf{fz}\ () \Rightarrow \mathbf{var}\ \mathsf{fz}\ n)) \end{array}\right)
\end{array}$$

5. Generic Programs

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

5.1. Modalities, map and find

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

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

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

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

$$\frac{p\ :\ \Diamond\mathsf{List}\ P\ as}{\mathsf{later}\ p\ :\ \Diamond\mathsf{List}\ P\ (a{::}as)}$$

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.

Firstly we define $\Box$, which we will give this type:

$$\underline{\text{let}}\ \frac{\begin{array}{l}\vec{I}\ :\ \mathsf{Vec}\ n\ \star\\ T\ :\ \mathsf{ISPT}\ \vec{I}\\ P\ :\ \forall i{:}\mathsf{Fin}\ n \Rightarrow \mathbf{SPF}\ \vec{J}\ (\exists x{:}\vec{I}!!i \Rightarrow [\![\mathbf{var}\ i\ x]\!]\ \vec{T})\\ v\ :\ [\![T]\!]\ \vec{T}\end{array}}{\Box T\ P\ v\ :\ \mathsf{ISPT}\ \vec{J}}$$

The predicate P is defined as a collection strictly positive families, one for each variable in the code T. The function is then defined by recursion over the code:

$$\begin{array}{lllll}
\Box T\ P\ v & \Leftarrow & \underline{\text{rec}}\ v & & \\
\Box(\mathsf{vz}\ x) & P & v & \Rightarrow & P\ \mathsf{fz}\ (x; v)\\
\Box(\mathsf{vs}\ T) & P & (\mathsf{pop}\ v) & \Rightarrow & \Box T\ (\lambda i\ (x\ w) \Rightarrow P\ (\mathsf{fs}\ i)\ (x; \mathsf{pop}\ w))\ v\\
\Box\text{`1'} & P & \mathsf{void} & \Rightarrow & \text{`1'}\\
\Box(\text{`}\Sigma\text{'}f\ F\ (f\ o)) & P & (\sigma\ (o; \mathsf{refl}; v)) & \Rightarrow & \Box(F\ o)\ P\ v\\
\Box(\text{`}\Pi\text{'}f\ F\ o') & P & (\pi\ \vec{v}) & \Rightarrow & \text{`}\Pi\text{'}(\lambda((o; p){:}\exists o{:}O \Rightarrow f\ o{=}o') \Rightarrow f\ o)\\
 & & & & \quad (\lambda(o; p) \Rightarrow \Box(F\ o)\ (\vec{v}\ o\ p))\ o'\\
\Box_{\vec{I}\ \vec{T}}(\text{`}\mu\text{'}\ F\ o) & P & v & \Rightarrow & \\
\multicolumn{5}{l}{\quad \text{`}\mu\text{'}\ (\lambda((o'; \mathsf{in}\ w){:}\exists o'{:}O \Rightarrow [\![F]\!]\ \vec{T}\ o') \Rightarrow}\\
\multicolumn{5}{l}{\quad\quad \Box(F\ o')\ \left(\lambda\ \begin{array}{lll}\mathsf{fz} & o'' & (\mathsf{top}\ v') \Rightarrow \mathsf{vz}\ (o''; v')\\ (\mathsf{fs}\ i) & x & (\mathsf{pop}\ v') \Rightarrow P\ i\ x\ v'\end{array}\right)\ w)\ (o; v)}
\end{array}$$

The key thing to notice here is that the case for `μ' here we the code we produce is a fix point over values in the original type.

The definition of $\Diamond$ follows much the same pattern, but with alternative results at `1' and `Π':

$$\begin{array}{lllll}
\Diamond T\ P\ v & \Leftarrow & \underline{\text{rec}}\ v & & \\
\Diamond(\mathsf{vz}\ x) & P & v & \Rightarrow & P\ \mathsf{fz}\ (x; v)\\
\Diamond(\mathsf{vs}\ T) & P & (\mathsf{pop}\ v) & \Rightarrow & \Diamond T\ (\lambda i\ (x\ w) \Rightarrow P\ (\mathsf{fs}\ i)\ (x; \mathsf{pop}\ w))\ v\\
\Diamond\text{`1'} & P & \mathsf{void} & \Rightarrow & \text{`0'}\\
\Diamond(\text{`}\Sigma\text{'}f\ F\ (f\ o)) & P & (\sigma\ (o; \mathsf{refl}; v)) & \Rightarrow & \Diamond(F\ o)\ P\ v\\
\Diamond(\text{`}\Pi\text{'}f\ F\ o') & P & (\pi\ \vec{v}) & \Rightarrow & \text{`}\Sigma\text{'}(\lambda((o; p){:}\exists o{:}O \Rightarrow f\ o{=}o') \Rightarrow f\ o)\\
 & & & & \quad (\lambda(o; p) \Rightarrow \Diamond(F\ o)\ (\vec{v}\ o\ p))\ o'\\
\Diamond_{\vec{I}\ \vec{T}}(\text{`}\mu\text{'}\ F\ o) & P & v & \Rightarrow & \\
\multicolumn{5}{l}{\quad \text{`}\mu\text{'}\ (\lambda((o'; \mathsf{in}\ w){:}\exists o'{:}O \Rightarrow [\![F]\!]\ \vec{T}\ o') \Rightarrow}\\
\multicolumn{5}{l}{\quad\quad \Diamond(F\ o')\ \left(\lambda\ \begin{array}{lll}\mathsf{fz} & o'' & (\mathsf{top}\ v') \Rightarrow \mathsf{vz}\ (o''; v')\\ (\mathsf{fs}\ i) & x & (\mathsf{pop}\ v') \Rightarrow P\ i\ x\ v'\end{array}\right)\ w)\ (o; v)}
\end{array}$$

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.

We can also define generic programs that use these types, for instance we can generalise the type of **map** to dependent functions, and that if we do the type of this function involves $\Box$.

$$\underline{\text{let}}\ \frac{\begin{array}{l} P \,:\, \forall i{:}\mathsf{Fin}\ n \Rightarrow \mathbf{SPF}\ \vec{J}\ (\exists x{:}\vec{I}!!i \Rightarrow [\![\mathbf{var}\ i\ x]\!]\ \vec{S}) \\ f \,:\, \forall i{:}\mathsf{Fin}\ n; t{:}(\exists x{:}\vec{I}!!i \Rightarrow [\![\mathbf{var}\ i\ x]\!]\ \vec{S}) \Rightarrow [\![P\ i\ t]\!]\ \vec{T} \\ v \,:\, [\![T]\!]\ \vec{T} \end{array}}{\mathbf{dMap}\ f\ v \,:\, [\![\Box T\ P\ v]\!]\ \vec{T}}$$

$$\begin{array}{llll} \mathbf{dMap}_{(\mathsf{vz}\ x)} & f & (\mathsf{top}\ v') & \Rightarrow f\ \mathsf{fz}\ (x, v') \\ \mathbf{dMap} & f & (\mathsf{pop}\ v') & \Rightarrow \mathbf{dMap}\ (f.\mathsf{fs})\ v' \\ \mathbf{dMap} & f & \mathsf{void} & \Rightarrow \mathsf{void} \\ \mathbf{dMap} & f & (\sigma\ (o; \mathsf{refl}; v')) & \Rightarrow \mathbf{dMap}\ f\ v' \\ \mathbf{dMap} & f & (\pi\ \vec{v}) & \Rightarrow \pi\ (\lambda(x; p)\ q \Rightarrow \mathbf{dMap}\ f\ (v'\ o\ p)) \\ \mathbf{dMap} & f & (\mathsf{in}\ v') & \Rightarrow \mathsf{in}\ (\mathbf{dMap} \begin{pmatrix} \lambda & \mathsf{fz}\ (o; w) \Rightarrow \mathbf{dMap}\ f\ w \\ & (\mathsf{fs}\ i)\ t \Rightarrow f\ i\ t \end{pmatrix} v') \end{array}$$

This program is not directly structurally recursive, but it can be made so in the same manner as with SPTs, as before we begin our definition by creating a type of morphisms that satisfy a predicate P:

$$\underline{\text{data}}\ \frac{\begin{array}{l} \vec{I} \,:\, \mathsf{Vec}\ m\ \star \quad \vec{J} \,:\, \mathsf{Vec}\ n\ \star \quad \vec{S} \,:\, \mathsf{Tel}\ \vec{I} \quad \vec{T} \,:\, \mathsf{Tel}\ \vec{J} \\ P \,:\, \forall i{:}\mathsf{Fin}\ m \Rightarrow \mathbf{SPF}\ \vec{J}\ (\exists x{:}\vec{I}!!i \ \Rightarrow\ [\![\mathbf{var}\ i\ x]\!]\ \vec{T}) \end{array}}{\blacksquare \vec{I}\ \vec{J}\ \vec{S}\ \vec{T}\ P \,:\, \star}\ \underline{\text{where}}$$

$$\frac{}{\varepsilon \,:\, \blacksquare\varepsilon\ \vec{J}\ \varepsilon\ \vec{T}\ (\lambda x \Leftarrow \underline{\text{case}}\ x)}$$

$$\frac{\begin{array}{l} \phi \,:\, \blacksquare \vec{I}\ \vec{J}\ \vec{S}\ \vec{T}\ P \\ X \,:\, \mathbf{SPF}\ \vec{J}\ (\exists x{:}I \Rightarrow [\![\mathbf{var}\ \mathsf{fz}\ x]\!]\ (\vec{T}{::}T)) \\ f \,:\, \forall v{:}(\exists x{:}I \Rightarrow [\![\mathbf{var}\ \mathsf{fz}\ x]\!]\ (\vec{T}{::}T)) \Rightarrow [\![X]\!]\ \vec{S}\ v \end{array}}{\phi{::}_X f \,:\, \blacksquare(\vec{I}{::}I)\ \vec{J}\ (\vec{T}{::}T)\ \vec{S} \begin{pmatrix} \lambda & \mathsf{fz} & \Rightarrow X \\ & (\mathsf{fs}\ i) & \Rightarrow P\ i \end{pmatrix}}$$

$$\frac{\phi \,:\, \blacksquare \vec{I}\ \vec{J}\ \vec{S}\ \vec{T}\ P \quad F \,:\, \mathbf{SPF}\ (\vec{I}{::}O)\ O}{\phi \blacktriangle F \,:\, \blacksquare(\vec{I}{::}O)\ \vec{J}\ (\vec{T}{::}F)\ \vec{S} \begin{pmatrix} \lambda & \mathsf{fz} & (o; t) \Rightarrow \Box(F\ o)\ P\ t \\ & (\mathsf{fs}\ i) & v \Rightarrow P\ i\ v \end{pmatrix}}$$

The key feature here is the $\blacktriangle$ constructor, which we will use to mark recursive variables, just as we did in the SPT case, the extra information in the type is there to make the following definition type check.

$$\underline{\text{let}}\ \frac{\phi : \blacksquare \vec{I}\ \vec{J}\ \vec{S}\ \vec{T}\ P \quad v : [\![T]\!]\ \vec{S}}{\mathbf{dMap}_T\ \phi\ v : [\![\Box T\ P\ v]\!]\ \vec{T}}$$

$$\begin{array}{l}
\mathbf{dMap}\ \phi\ v \Leftarrow \underline{\text{rec}}\ v\ \{ \\
\quad \begin{array}{lllll}
\mathbf{dMap}_{(\mathsf{vz}\ x)} & (\phi' :\!\!_X f) & (\mathsf{top}\ v') & \Rightarrow & f\ (x, v') \\
\mathbf{dMap} & (\phi' \blacktriangle F) & (\mathsf{top}\ v') & \Rightarrow & \mathbf{dMap}\ \phi'\ v' \\
\mathbf{dMap} & (\phi' :\!\!_X f) & (\mathsf{pop}\ v') & \Rightarrow & \mathbf{dMap}\ \phi'\ v' \\
\mathbf{dMap} & (\phi' \blacktriangle F) & (\mathsf{pop}\ v') & \Rightarrow & \mathbf{dMap}\ \phi'\ v' \\
\mathbf{dMap} & \phi & \mathsf{void} & \Rightarrow & \mathsf{void} \\
\mathbf{dMap} & \phi & (\sigma\ (o; \mathsf{refl}; v')) & \Rightarrow & \mathbf{dMap}\ \phi\ v' \\
\mathbf{dMap} & \phi & (\pi\ \vec{v}) & \Rightarrow & \pi\ (\lambda(x; p)\ q \Rightarrow \mathbf{dMap}\ \phi\ (v'\ o\ p)) \\
\mathbf{dMap} & \phi & (\mathsf{in}\ v') & \Rightarrow & \mathsf{in}\ (\mathbf{dMap}\ (\phi \blacktriangle F)\ v')
\end{array} \\
\}
\end{array}$$

We have said that $\Diamond$ is the dual of $\Box$, that being the case it is reasonable to ask what the dual of **dMap** is, returning to the type of the incorrect definition of **dMap**, we dualise thus:

$$\underline{\text{let}}\ \frac{\begin{array}{l} P : \forall i : \mathsf{Fin}\ n \Rightarrow \mathbf{SPF}\ \vec{J}\ (\exists x : \vec{I} !! i \Rightarrow [\![\mathbf{var}\ i\ x]\!]\ \vec{S}) \\ v : [\![T]\!]\ \vec{T} \\ p : [\![\Box T\ P\ v]\!]\ \vec{T} \end{array}}{\mathbf{dFind}\ v\ p : \exists i : \mathsf{Fin}\ n; t : (\exists x : \vec{I} !! i \Rightarrow [\![\mathbf{var}\ i\ x]\!]\ \vec{S}) \Rightarrow [\![P\ i\ t]\!]\ \vec{T}}$$

That is, if the $\Diamond P$ property holds for v the **dFind** will extract a sub-tree and a proof that P holds for that value. We would like this function to have this behaviour:

$$\begin{array}{llll}
\mathbf{dFind}_{(\mathsf{vz}\ x)} & (\mathsf{top}\ v') & p & \Rightarrow (\mathsf{fz}; x; \mathsf{top}\ v'; p) \\
\mathbf{dFind} & (\mathsf{pop}\ v') & p & \Rightarrow (\mathsf{fs}; -; \mathsf{pop}; -)\ (\mathbf{dFind}\ v'\ p) \\
\mathbf{dFind} & (\sigma\ (o; \mathsf{refl}; v')) & p & \Rightarrow \mathbf{dFind}\ v'\ p \\
\mathbf{dFind} & (\pi\ \vec{v}) & (\sigma(o; q; p')) & \Rightarrow \mathbf{dFind}\ (v'\ o\ q)\ p' \\
\mathbf{dFind} & (\mathsf{in}\ v') & (\mathsf{in}\ p') & \left\| \begin{array}{l} \mathbf{dFind}\ v'\ p' \\ (\mathsf{fz}; o; w; q) \Rightarrow \mathbf{dFind}\ w\ q \\ (\mathsf{fs}\ i; x; w; q) \Rightarrow (i; x; w; q) \end{array} \right.
\end{array}$$

Again, however, this is not structurally recursive as shown, when we go under a 'μ' constructor the property that might hold is the $\Diamond$ property at some arbitary vz node, on which we have to call **dFind** again, in the above code there is no guarantee that what comes out is a strict sub-tree of the input and so we have to do more work to justify this process. Luckily we can play a trick similar to the one used to fix **dMap** to recover the functionality we want with structural recursion.

5.2. Equality of RFs

Another key example for any generic programming system is how it deals with value equality. Clearly, however the universe of **SPF**s is too large to support a decidable equality, since it contains codes for infinitely branching data-types. Clearly

to have any sort of generic value equality we will need to dispense with the ‘Π’ constructor, just as we did by removing the ‘→’ construct from the SPT to arrive at the regular-tree types, with support equality in previous work. Here we would need to replace the function constructor with a code for cartesian product, since the latter is defined in terms of the former. Even this, howevfer, this would not be sufficient, to see why we must return to the definition of disjoint union by ‘Σ’:

$$\underline{\text{let}}\ \frac{A, B\ :\ \mathsf{ISPT}\ \vec{I}}{A\ \text{`+'}\ B\ :\ \mathsf{ISPT}\ \vec{I}}\ ;\ \ A\ \text{`+'}\ B\ \Rightarrow\ \text{`}\Sigma\text{'}(\lambda x \Rightarrow ())\ \begin{pmatrix} \lambda\ \mathsf{fz} & \Rightarrow A \\ \quad (\mathsf{fs}\ \mathsf{fz}) & \Rightarrow B \end{pmatrix}\ ()$$

$$\underline{\text{let}}\ \frac{a\ :\ [\![A]\!]\ \vec{T}}{\mathbf{inl}\ a\ :\ [\![A\ \text{`+'}\ B]\!]\ \vec{T}}\ ;\ \ \mathbf{inl}\ a\ \Rightarrow\ \sigma\ (\mathsf{fz}; \mathsf{refl}; a)$$

$$\underline{\text{let}}\ \frac{b\ :\ [\![B]\!]\ \vec{T}}{\mathbf{inr}\ b\ :\ [\![A\ \text{`+'}\ B]\!]\ \vec{T}}\ ;\ \ \mathbf{inr}\ b\ \Rightarrow\ \sigma\ (\mathsf{fs}\ \mathsf{fz}; \mathsf{refl}; b)$$

If we use this definition to create a code for Booleans:

$$\underline{\text{let}}\ \frac{}{\text{`}\mathbf{Bool}\text{'}\ :\ \mathbf{SPF}\ \vec{I}\ \mathsf{One}}\ ;\ \ \text{`}\mathbf{Bool}\text{'}\ ()\ \Rightarrow\ \text{`1'}\ \text{`+'}\ \text{`1'}$$

$$\underline{\text{let}}\ \frac{}{\text{`}\mathbf{true}\text{'}, \text{`}\mathbf{false}\text{'}\ :\ [\![\text{`}\mathbf{Bool}\text{'}]\!]\ \vec{T}\ ()}$$

$$\begin{array}{l} \text{`}\mathbf{true}\text{'}\ \Rightarrow\ \mathbf{inl}\ \mathsf{void} \\ \text{`}\mathbf{false}\text{'}\ \Rightarrow\ \mathbf{inr}\ \mathsf{void} \end{array}$$

The fully evaluated codes for ‘**true**’ and ‘**false**’ are:

$$\begin{array}{lcl} \text{`}\mathbf{true}\text{'} & \mapsto & \sigma(\mathsf{fz}; \mathsf{refl}; \mathsf{void}) \\ \text{`}\mathbf{false}\text{'} & \mapsto & \sigma(\mathsf{fs}\ \mathsf{fz}; \mathsf{refl}; \mathsf{void}) \end{array}$$

We can tell these codes apart because we fz and fs fz are different, in fact exactly because Fin has a decidable equality, but in general the new indexing information introduced by a ‘Σ’ constructor need have no decidable equality, so a truly generic equality could not distinguish ‘**true**’ from ‘**false**’

To define a syntactic equality we must therefore replace ‘Π’ with a code for cartesian product and add a dual code for disjoint union to create IRT, the indexed regular types:

$$\frac{A, B\ :\ \mathsf{IRT}\ \vec{I}}{A\ \text{`+'}\ B\ :\ \mathsf{IRT}\ \vec{I}} \qquad \frac{A, B\ :\ \mathsf{IRT}\ \vec{I}}{A\ \text{`}\times\text{'}\ B\ :\ \mathsf{IRT}\ \vec{I}}$$

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

$$\frac{a\ :\ [\![A]\!]\ \vec{T} \quad b\ :\ [\![B]\!]\ \vec{T}}{\mathsf{pair}\ a\ b\ :\ [\![A\ \text{`+'}\ B]\!]\ \vec{T}}$$

We also create the regular families **RF** in the way you would expect, and we overload the Scott bracket interpretation notation. In this way we can recast all our examples above into this new universe, by swapping out the defined '+' and '×' for the codes with the same names.

We need to keep the original 'Σ' constructor in place otherwise we lose expressive power, but we now have two places where choices are made, we will distinguish between choices at '+' codes but not at 'Σ' codes, the intention is that the former is used to encode choice between constructors the latter encode choice between indicies.

We can now define our equality, which we do heterogeneously:

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

$$\begin{array}{lll}
\mathbf{gEq}\ a\ b & \Leftarrow\ \underline{\text{rec}}\ a & \\
\quad \mathbf{gEq} & (\mathsf{top}\ a) \quad (\mathsf{top}\ b) & \Rightarrow\ \mathbf{gEq}\ a\ b \\
\quad \mathbf{gEq} & (\mathsf{pop}\ a) \quad (\mathsf{pop}\ b) & \Rightarrow\ \mathbf{gEq}\ a\ b \\
\quad \mathbf{gEq} & (\mathsf{inl}\ a) \quad (\mathsf{inl}\ b) & \Rightarrow\ \mathbf{gEq}\ a\ b \\
\quad \mathbf{gEq} & (\mathsf{inl}\ a) \quad (\mathsf{inr}\ b) & \Rightarrow\ \mathsf{false} \\
\quad \mathbf{gEq} & (\mathsf{inr}\ a) \quad (\mathsf{inl}\ b) & \Rightarrow\ \mathsf{false} \\
\quad \mathbf{gEq} & (\mathsf{inr}\ a) \quad (\mathsf{inr}\ b) & \Rightarrow\ \mathbf{gEq}\ a\ b \\
\quad \mathbf{gEq} & \mathsf{void} \quad \mathsf{void} & \Rightarrow\ \mathsf{true} \\
\quad \mathbf{gEq} & (\mathsf{pair}\ ax\ ay) \quad (\mathsf{pair}\ bx\ by) & \Rightarrow\ \mathbf{gEq}\ ax\ bx\ \&\&\ \mathbf{gEq}\ ax\ bx \\
\quad \mathbf{gEq} & (\sigma\ a) \quad (\sigma\ b) & \Rightarrow\ \mathbf{gEq}\ a\ b \\
\quad \mathbf{gEq} & (\mathsf{in}\ a) \quad (\mathsf{in}\ b) & \Rightarrow\ \mathbf{gEq}\ a\ b \\
\quad \mathbf{gEq} & _ \quad _ & \Rightarrow\ \mathsf{false}
\end{array}$$

We have cheated slightly here, by omitting all the off-diagonal cases. Within Epigram this could have been achieved by creating a view which justifies the case-split above.

That we have different codes for the types of the two arguments is necessary again in the 'Σ' case where we don't know that the new indexes are the same so we cannot know whether the types of the sub-trees are the same.

We decide the equality of values based entirely on their syntax, we therefore equate values at different output indexes as long as the syntax is the same; so, for instance $\mathsf{fz} : \mathsf{Fin}\ n = \mathsf{fz} : \mathsf{Fin}\ (\mathsf{s}\ n)$. In practice it might be better to restrict ourselves only to comparing things for equality at the top level, though we can no better as we traverse under 'Σ' constructors.

It remains very difficult to specify this equality function, something that would be very useful in a dependent programming language, if two things are equal for instance you would like to be able to substitute one for the other. It is future work to see if we can do any better with alternative definitions of these regular families. One possibility is to tie a recursive knot and restrict output types to be indexed regular types themselves, though this raises real questions about universe hierarchies which would need careful study.

6. Future Work and Conclusions

We have tied the knot by presenting a universe construction which is powerful enough to encode all inductive types needed in Epigram, including the construction itself. While encoding types by hand is a rather cumbersome process, we can translate the high level Epigram syntax mechanically into the SPFs. We plan to integrate the universe directly into Epigram giving the programmer direct access to the internal representations of types for generic programming as part of the system. This approach also has the benefit that it allows a more flexible and extensible positivity test as we have demonstrated in the example of n-branching trees. Exploiting Observational Type Theory [5] we are also planning to include co-inductive definitions in the universe.

In as yet unpublished work [3] we have extended the existing notion of data types as containers [1] to families of data types and *indexed containers*. The codes in the universe (**SPF**,$[\![-]\!]$) can be given normal forms as indexed containers, just as the universe of (SPT,EI) was reflected in containers. This gives us a justification of the constants chosen here, but also alternative access to generic programming using a metaphor of shapes and positions, see [2] for simply-typed examples.

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

1. Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Containers - constructing strictly positive types. *Theoretical Computer Science*, 342:3–27, September 2005. Applied Semantics: Selected Topics.
2. Michael Abbott, Thorsten Altenkirch, Neil Ghani, and Conor McBride. ∂ for data. *Fundamentae Informatica*, 65(1,2):1 – 28, March 2005. Special Issue on Typed Lambda Calculi and Applications 2003.
3. Thorsten Altenkirch, Neil Ghani, Peter Hancock, Conor McBride, and Peter Morris. Indexed containers. Manuscript, available online, April 2007.
4. Thorsten Altenkirch and Conor McBride. Generic programming within dependently typed programming. In *Generic Programming*, 2003. Proceedings of the IFIP TC2 Working Conference on Generic Programming, Schloss Dagstuhl, July 2002.
5. Thorsten Altenkirch and Conor McBride. Towards observational type theory. Manuscript, available online, February 2006.
6. Thorsten Altenkirch, Conor McBride, and James McKinna. Why dependent types matter. Manuscript, available online, April 2005.

7. Marcin Benke, Peter Dybjer, and Patrik Jansson. Universes for generic programs and proofs in dependent type theory. *Nordic Journal of Computing*, 10:265–269, 2003.
8. D. Clarke, R. Hinze, J. Jeuring, A. Loh, and J. de Wit. The Generic Haskell user's guide. Technical report, Technical Report UU-CS-2001-26, Utrecht University, 2001.
9. C. Coquand. The AGDA proof assistant. *URL: http://www.cs.chalmers.se/ catarina/agda, Department of Computer Science, Chalmers University of Technology and Goteborgs Universitet*, 2000.
10. P. Dybjer and A. Setzer. A finite axiomatization of inductive-recursive definitions. *Typed Lambda Calculi and Applications*, 1581:129–146, 1999.
11. Peter Dybjer and Anton Setzer. Indexed induction-recursion. In Reinhard Kahle, Peter Schroeder-Heister, and Robert F. Stärk, editors, *Proof Theory in Computer Science*, volume 2183 of *Lecture Notes in Computer Science*, pages 93–113. Springer, 2001.
12. R. Hinze. Generic programs and proofs. *Habilitation Thesis, University of Bonn, Germany*, 2000.
13. R. Hinze and S.P. Jones. Derivable Type Classes. *Electronic Notes in Theoretical Computer Science*, 41(1), 2001.
14. R. Hinze, A. Löh, and B.C.S. Oliveira. Scrap Your Boilerplate Reloaded. *Proceedings of the Eighth International Symposium on Functional and Logic Programming (FLOPS 2006)*, pages 24–26, 2006.
15. Ralf Hinze and Andres Löh. "Scrap Your Boilerplate" Revolutions. In Tarmo Uustalu, editor, *Mathematics of Program Construction, 2006*, volume 4014 of *LNCS*, pages 180–208. Springer-Verlag, 2006.
16. P. Jansson and J. Jeuring. PolyP-a polytypic programming language extension. *POPL*, 97:470–482, 1997.
17. Patrik Jansson. *Functional Polytypic Programming*. PhD thesis, Computing Science, Chalmers University of Technology and Göteborg University, Sweden, May 2000.
18. S.P. Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADT s. In *The 11th ACM SIGPLAN International Conference on Functional Programming (ICFP 2006)*, 2006.
19. Andres Löh. *Exploring Generic Haskell*. PhD thesis, Utrecht University, Netherlands, September 2004.
20. Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis·Napoli, 1984.
21. Conor McBride. Epigram, 2004. `http://www.e-pig.org/`.
22. Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1), 2004.
23. Peter Morris, Thorsten Altenkirch, and Conor McBride. Exploring the regular tree types. In Christine Paulin-Mohring Jean-Christophe Filliatre and Benjamin Werner, editors, *Types for Proofs and Programs (TYPES 2004)*, Lecture Notes in Computer Science, 2006.
24. Bengt Nordström, Kent Petersson, and Jan Smith. *Programming in Martin-Löf's Type Theory: an introduction*. Oxford University Press, 1990.
25. Stephanie Weirich. RepLib: A library for derivable type classes. In Andres Löh, editor, *Proceedings of the ACM Haskell Workshop, 2006*, 2006. to appear.