# Indexed Containers

Thorsten Altenkirch[1], Neil Ghani[1], Peter Hancock[1], Conor McBride[1], and Peter Morris[1]

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

**Abstract.** The search for an expressive calculus of datatypes in which canonical algorithms can be easily written and proven correct has proved to be an enduring challenge to the theoretical computer science community. Approaches such as polynomial types, strictly positive types and inductive types have all met with some success but they tend not to cover important examples such as types with variable binding, types with constraints, nested types, dependent types etc.

In order to compute with such types, we generalise from the traditional treatment of types as free standing entities to families of types which have some form of indexing. The hallmark of such indexed types is that one must usually compute not with an individual type in the family, but rather with the whole family simultaneously. We implement this simple idea by generalising our previous work on *containers* to what we call *indexed containers* and show that they cover a number of sophisticated datatypes and, indeed, other computationally interesting structures such as the refinement calculus and interaction structures. Finally, and rather surprisingly, the extra structure inherent in indexed containers simplifies the theory of containers and thereby allows for a much richer and more expressive calculus.

## 1 Introduction

**Why Indexed Containers?** The search for an expressive calculus of datatypes 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 datatypes which allows generic algorithms such as searching, traversal and differentiation to be written. This calculus should cover as many examples of datatypes as possible so that the generic algorithms can be applied widely, but should exclude types such as $\mu X.(X \to 2) \to 2$ which do not support such algorithms. Approaches such as polynomial types, strictly positive types and inductive types have all met with much success but they tend not to cover important examples such as types with variable binding, types with constraints, nested types, dependent types etc.

In previous work, we introduced the concept of *containers* ([3], [4], and [2]) as a theory of datatypes based upon the metaphor that datatypes consist of shapes and positions where data can be stored — pathological examples such $\mu X.(X \to 2) \to 2$ are therefore excluded. Consequently, the large number of algorithms whose essence is the manipulation of shapes and positions can be uniformly written for containers. Examples of such algorithms are traversal, searching and differentiation. Unfortunately, while delivering generic algorithms, containers suffer from the same problem of limited expressivity as strictly positive types etc.

This paper seeks to address this problem by generalising the traditional treatment of types as free standing entities to families of types which have some form of indexing. To understand this idea, consider the inductive type $\mathsf{List}(X)$ of lists of $X$. It is clear that the definition of $\mathsf{List}(X)$ does not require an understanding of $\mathsf{List}(Y)$ for for any $Y \neq X$. Since each instance $\mathsf{List}(X)$ is, in isolation, an inductive type we consider $\mathsf{List}$ to be a family of inductive types. In contrast, consider the following definition of the Nat-indexed type $\mathrm{Fin}[n]$ of finite sets

$$0 : \mathrm{Fin}[n+1] \qquad\qquad \frac{x : \mathrm{Fin}[n]}{\mathrm{S}(x) : \mathrm{Fin}[n+1]}$$

Similarly one could consider the Nat-indexed type $\mathrm{Lam}(n)$ of $\lambda$-terms [19, 8, 13] by means of the following introduction rules.

$$\frac{i : \mathrm{Fin}[n]}{\mathrm{Var}(i) : \mathrm{Lam}[n]} \qquad \frac{f : \mathrm{Lam}[n] \quad a : \mathrm{Lam}[n]}{\mathrm{App}(f, a) : \mathrm{Lam}[n]} \qquad \frac{b : \mathrm{Lam}[n+1]}{\mathrm{Abs}(b) : \mathrm{Lam}[n]}$$

Here, the type $\mathrm{Lam}[n]$ consists of untyped $\lambda$-terms up to $\alpha$-equivalence with free variables drawn from $\{\, \mathrm{Var}(0), \ldots, \mathrm{Var}(n-1) \,\}$. Note that in $\mathrm{Abs}(b)$, the bound variable is $\mathrm{Var}(0)$. The key point in both of these examples is that, unlike the case with lists, the type $\mathrm{Fin}[n]$ and the type $\mathrm{Lam}[n]$ cannot be defined in isolation with recourse only to the elements of $\mathrm{Fin}[n]$ and $\mathrm{Lam}[n]$ that have already been built. Rather we need elements of the type $\mathrm{Fin}[n]$ to build elements of $\mathrm{Fin}[n+1]$ and, similarly, elements of the type $\mathrm{Lam}[n+1]$ to build elements of the type $\mathrm{Lam}[n]$. In effect, the Nat-indexed families $\mathrm{Fin}[n]$ and $\mathrm{Lam}[n]$ have to be inductively built up simultaneously for every $n$ and thus we have an inductive family of types rather than a family of inductive types.

These kinds of indexed datatypes are becoming increasingly important. Of course, one can define them in a dependently typed programming language or they can be approximated using nested types or GADTs in Haskell. However, these approaches by themselves do not highlight the shapes and positions metaphor which, as remarked above, is crucial for defining many generic algorithms. Thus it is natural to try to combine the shapes and positions metaphor of containers with the expressivity of indexed families of datatypes. Such *indexed containers* consist of containers with additional indexing information detailing the the indexes that each shape can target, and for each input position indication of how

the input's type should be indexed, which index a piece of data to be stored there must come from. As we will comment on later, indexed containers are a serious candidate to be used as the theory of datatypes in the programming language Epigram. We hasten to add that these indexed containers have been previously studied by Hyland and Gambino [20] under the name of *dependent polynomials* — see the section on related research for more details.

This kind of indexing information is actually very natural and is already widely used in many sorted algebra. In such a setting, each operator is assigned an output sort and, for each operand position, the sort of the expression that is required in that position is given. For instance, in the following two sorted signature

$$\begin{array}{ll} \text{if} & : \text{Bool} \times \text{Int} \times \text{Int} \to \text{Int} \\ \text{succ} & : \text{Int} \to \text{Int} \\ \text{not} & : \text{Bool} \to \text{Bool} \end{array} \qquad \begin{array}{ll} \text{zero} & : \text{Int} \\ \text{eq} & : \text{Int} \times \text{Int} \to \text{Bool} \\ \text{or} & : \text{Bool} \times \text{Bool} \to \text{Bool} \end{array}$$

the if-operator produces something of sort Int and has three input positions requiring data of sort $\text{Bool}, \text{Int}$ and Int respectively. More surprisingly, indexed containers encompass a number of other computationally interesting examples which go well beyond examples driven by intuition based upon datatypes with shapes and positions. Indeed, in section 7.3, we show how *interaction structures* and the *refinement calculus* can be simply analysed with indexed containers.

**Our Contributions:** This paper develops the concept of indexed containers. In detail, we provide

- A definition of indexed containers and their semantic extensions as functors over slice categories.
- A new grammar of indexed strictly positive types which forms a calculus of indexed containers. We prove a completeness theorem for this grammar.
- A collection of operations such as differentiation, monoidal closure and modal operators which are definable as operations on indexed containers.
- A notion of morphism between indexed containers. We prove that they are in bijective correspondence with the natural transformations between containers.
- A different notion of morphism based upon simulation arising from interaction structures.
- Applications to i) the semantics of datatypes as in *Epigram*; ii) interaction structures; and iii) the refinement calculus

This paper is designed to be read as a set theoretic, type theoretic and categorical development of the theory of indexed containers so the paper can be read by as wide an audience as possible of those interested in program construction. By working in the category of $\mathsf{Sets}$, readers can follow our arguments, constructions and examples without needing a categorical or type theoretic background. However, to ensure our constructions are valid in any standard model of a programming language the paper is written using type theory and category theory.

There are occasions when the one presentation is simpler than the other and hence, by use of an internal language linking the two, we can enjoy the best, and avoid the worst, of both worlds.
Formally, the internal language links the extensional type theory MLW-EXT (see[7]) with finite types, $W$-types, a proof true $\neq$ false (but without universes) to locally cartesian closed categories with disjoint coproducts and initial algebras of container functors in one variable. Categorically, we follow standard practise and represent an $I$-indexed family by an arrow $f : X \to I$ which we think of mapping every element of $X$ to its index. Type theoretically, we represent an $I$-indexed family by a judgement $i : I \vdash X[i]$ TYPE — henceforth we omit the TYPE annotation. Given such an $f$, its representation in type theory is the judgement $i : I \vdash f[i]$ where $f[i]$ is defined to be $\Sigma x : X.f(x) = i$. On the other hand, give a type theoretic $I$-indexed family $i : I \vdash X[i]$, its representation categorically is the function $\pi_0 : (\Sigma i : I.X[i]) \to I$. Note the use of round brackets to denote function application and square brackets to denote the fibre over an index. We often define an $I$-indexed family $f$ by giving, for each $i$, the fibre $f[i]$ and note that an indexed family $f$ is equivalent to the indexed family $\Sigma i : I.f[i]$. More details of the internal language, and its interpretation in the category of Sets are given in the appendix.

The structure of the paper is as follows. Section 2 briefly recapitulates the notion of unindexed containers, and their extension as endofunctors. Section 3 contains definitions of containers indexed over given input and output types, their extensions as functors between slice categories over those types, and the three basic forms of endofunctor from which they are composed. Section 4 gives a grammar of indexed strictly positive types, and establishes that it is both sound and complete with respect to indexed containers. (The treatment of fixed points is postponed to section 6.) Section 5 turns to morphisms between indexed containers with given index types, and defines a notion that precisely captures natural transformations between their extensions, thus extending the representation theorem for unindexed containers established in previous papers. In section 6 we treat fixed points (both initial algebras and terminal coalgebras) for indexed containers, completing the semantics for the calculus of indexed strictly positive types begun in section 4. Section 7 considers applications: firstly to the foundations of the dependently typed programming language Epigram, and secondly to the refinement calculus and finally to the specification of imperative interfaces, such as may be required for modules written in Epigram to interact with modules written in other languages. Finally, in sections 8 and 9 we comment on related research, and those respects in which we have gone further, and remark on some possibilities for further research.

## 2 A Brief History of Containers

Containers capture the idea that concrete datatypes consist of memory locations where data can be stored. For example, any element of the type $\mathsf{List}(X)$ of lists of $X$ can be uniquely written as a natural number $n$ given by the length of the

list, together with a function $\{0, \ldots, n-1\} \to X$ which labels each position within the list with an element from $X$:

$$n : \mathrm{Nat} \vdash f : \{0 \ldots n-1\} \to X \ .$$

We think of the set $\{0, \ldots, n-1\}$ as $n$ memory locations while the function $f$ attaches to each of these memory locations the data to be stored there. To avoid commitment to any specific semantic domain, we are led to consider datatypes which are given by shapes $S$ and, for each $s : S$, a type of positions $P[s]$.

**Definition 1 (Container).** *A container $S \lhd P$ consists of a $S$-indexed family $P$. That is, either a judgement $s : S \vdash P[s]$ or an arrow $f : P \to S$.*

This definition does not restrict us to the category of sets but is designed to be interpreted in any locally cartesian closed category, as well as certain forms of fibration such as comprehension categories or models of Martin-Löf type theory [5, 2, 1]. Nevertheless, if the reader wishes, he/she can work in Sets and regard a container as a set of positions $S$, and for each $s : S$, a set of positions $P[s]$. As suggested above, lists can be presented as a container with shapes Nat and positions $n : \mathrm{Nat} \vdash \mathrm{Fin}[n]$.

The extension of a container is an endofunctor on the underlying LCCC defined as follows:

**Definition 2 (Extension of a Container).** *Let $S \lhd P$ be a container. Its extension, is the functor $T_{S \lhd P}$ defined by*

$$T_{S \lhd P}(X) = \Sigma s : S.\ P[s] \to X$$

Set theoretically, an element of $T_{S \lhd P}(X)$ is thus a pair $(s, f)$ where $s : S$ is a shape and $f : P[s] \to X$ is a labelling of the positions over $s$ with elements from $X$. Note that $T_{S \lhd P}(X)$ really is a functor since its action on a map $g : X \longrightarrow Y$ sends the element $(s, f)$ to the element $(s, g \cdot f)$. Thus for example, the extension of the container for lists is the functor mapping $X$ to $\Sigma n : \mathrm{Nat}.\mathrm{Fin}[n] \to X$ - such pairs clearly bijectively correspond to lists.

The theory of containers was developed in a series of recent papers [5, 2, 1] which showed that containers encompass a wide variety of types, as they are closed under various type forming operations such as sums, products, constants, fixed exponentiation, (nested) least fixed points and (nested) greatest fixed points. Therefore containers provide an algebraic treatment of strictly positive types. The generalisation to indexed containers aims to go further and develop the meta-theory required to compute with the much larger and more expressive class of inductive families of types.

## 3 Indexed Containers

An agreeably straightforward notion of a indexed container is that of a container $S \lhd P$ together with an assignment of an index sort for each shape, and an assignment of an index sort to each input position in that shape.

**Definition 3 (Indexed Containers).** *If $I$ and $O$ are input sorts and output sorts respectively, an $(I, O)$-indexed container $(S, P, q, r) : \mathrm{IC}(I, O)$ is given type theoretically by judgements*

$$\begin{array}{ll} \vdash S & s : S \vdash P[s] \\ q : S \to O & r : (\Sigma s : S.P[s]) \to I \end{array}$$

*or (alternatively) categorically as a container $f : P \to S$ together with indexing information given by arrows $q : S \to O$ and $r : P \to I$.*

$$\begin{array}{ccc} P & \xrightarrow{f} & S \\ {\scriptstyle r}\downarrow & & \downarrow{\scriptstyle q} \\ I & & O \end{array}$$

The Nat-indexed families Fin and Lam we met in the introduction arise as fixed points of indexed containers $(S_F, P_F, q_F, r_F)$ and $(S_L, P_L, q_L, r_L)$ defined as follows:

*Example 4 (Fin and $\lambda$-terms).* There are two ways of producing an element of type $\mathrm{Fin}[n+1]$. The first (corresponding to the constructor 0) requires no input, while the second (corresponding to the constructor $S$) requires one input indexed by $n$. Thus we set

$$\begin{array}{llll} S_F[n+1] & = \{0, \mathrm{S}\} & P_F[n+1, 0] & = 0 \quad P_F[n+1, \mathrm{S}] = 1 \\ q_F(n+1, -) = n+1 & & r_F(n+1, 0, -) = ! & r_F(n+1, \mathrm{S}, *) = n \end{array}$$

A similar analysis for $\lambda$-terms suggests defining

$$\begin{array}{lll} S_L[n] & = \mathrm{Fin}[n] + \{\mathrm{App}, \mathrm{Abs}\} & q_L(n, -) = n \\ P_L[n, i] & = 0 & P_L[n, \mathrm{App}] = 2 \quad P_L[n, \mathrm{Abs}] = 1 \\ r_L(n, i, -) = ! & & r_L(n, \mathrm{App}, x) = n \quad r_L(n, \mathrm{Abs}, *) = n+1 \end{array}$$

Next, we give the extension of an indexed container as a functor. Since the inputs are $I$-sorted and the outputs are $O$-sorted, this extension will map $I$-indexed families to $O$-indexed families as follows

**Definition 5 (Extension of an Indexed Container).** *Let $(S, P, q, r) : \mathrm{IC}(I, O)$ be an indexed container. Its extension is the functor $[\![S, P, q, r]\!] : \mathcal{C}/I \to \mathcal{C}/O$ whose action on objects maps an $I$-indexed family $k$ to the $O$-indexed family whose fibre at $o : O$ is given by*

$$[\![S, P, q, r]\!]\ k\ o = \Sigma s : S.\ q\ s = o \wedge \Pi p : P[s].\ k[r(s, p)]$$

*The action of $[\![S, P, q, r]\!]$ on morphisms is analogous to that for containers.*

Set theoretically, this definition says that to produce an element of $[\![S, P, q, r]\!]\ k$ which is indexed by $o$, we must pick a shape that will produce something of type $o$ and then assign to every input position of that shape, a piece of data from $k$

whose input type is that expected by $r$. Presented categorically, the extension of an indexed container takes on the following particularly simple and elegant form which highlights the fact that the extension of an indexed container is the composite of simpler functors:

$$\mathcal{C}/I \xrightarrow{\Delta_r} \mathcal{C}/P \xrightarrow{\Pi_f} \mathcal{C}/S \xrightarrow{\Sigma_q} \mathcal{C}/O$$

That the type theoretic and categorical definitions agree is proven by translating the categorical definition into the internal language. This shows the following

**Lemma 6.** *Let $f : I \to I'$. Then the reindexing functor $\Delta_f$ and its left and right adjoints $\Sigma_f$ and $\Pi_f$ are extensions of the indexed containers $(id : I \to I, id, f)$, $(id : I \to I, f, id)$ and $(f : I \to I', id, id)$ respectively.*

In summary, indexed containers can easily be seen as embellishments of containers with indexing information. Apart from making the concept of a indexed container straightforward, this metaphor of indexing has the practical benefit that much of the technical development of indexed containers can be inherited from that for containers. Formally, a container $S \triangleleft P$ can be considered as a indexed container with $I = O = 1$ with $q$ and $r$ being the unique maps into 1. In the other direction, if $(S, P, q, r)$ is a indexed container, there is an obvious 'forgetful' functor which forgets the indexing information, leaving the underlying container $S \triangleleft P$. There is another embedding of a container $S \triangleleft P$ as an indexed container $(S, P, id, !) : \mathrm{IC}(1, S)$ which records the shape information as the index. This can be useful on occassions, — for example when specifying *zip* to work on lists of equal length, or operations on matrices (13). Indexed Containers of this form have one shape for every output index and hence can be thought of as indexed reader monads as they just read data into the positions.

## 4 Indexed Strictly Positive Types

Surprisingly the grammar of indexed strictly positive types which we use to generate indexed containers is very different from that of strictly positive types which plays the analogous role for containers. While strictly positive types consist of nested fixed points of polynomial functors built from sum and times, indexed strictly positive types are based around nested fixed points of reindexing $\Delta_f$ and its left and right adjoints $\Sigma_f$ and $\Pi_f$.

**Definition 7 (Indexed Strictly Positive Types).** *If $I$ and $O$ are input and output sorts respectively, the indexed strictly positive types (*ISPT*s) from $I$ to $O$ are denoted* $\mathrm{ISPT}(I, O)$ *and are defined in Figure 1.*

We shall see that all ISPTs define indexed containers and hence that ISPTs represent functors over slice categories. Because fixed points are inherently more complex, we treat them in a separate section and so define $\mathrm{ISPT}_0(I, O)$ to be those $T : \mathrm{ISPT}(I, O)$ which do not contain any fixed points.

$$\frac{}{\text{Id} : \text{ISPT}(O,O)} \qquad \frac{k : X \to O}{\text{K}\ k : \text{ISPT}(I,O)} \qquad \frac{f : O' \to O \quad T : \text{ISPT}(I,O)}{\Delta f\, T : \text{ISPT}(I,O')}$$

$$\frac{A : \text{ISPT}(I,O) \quad B : \text{ISPT}(I,O)}{\text{Tag}\ A\ B : \text{ISPT}(I, O+O)} \qquad \frac{f : O \to O' \quad T : \text{ISPT}(I,O)}{\Sigma f\, T : \text{ISPT}(I,O')}$$

$$\frac{T : \text{ISPT}(I+O,O)}{\mu T : \text{ISPT}(I,O)} \qquad \frac{T : \text{ISPT}(I+O,O)}{\nu T : \text{ISPT}(I,O)} \qquad \frac{f : O \to O' \quad T : \text{ISPT}(I,O)}{\Pi f\, T : \text{ISPT}(I,O')}$$

**Fig. 1.** Indexed Strictly Positive Types

**Lemma 8.** *Every* $T : \text{ISPT}_0(I,O)$ *defines a functor* $[\![T]\!] : \mathcal{C}/I \to \mathcal{C}/O$ *which is the extension of an indexed container in* $\text{IC}(I,O)$

*Proof.* We go through the cases. $\text{Id} : \text{ISPT}_0(O,O)$ represents the identity functor on the slice category $\mathcal{C}$. This is the extension of the indexed container

$$O \xleftarrow{id} O \xrightarrow{id} O \xrightarrow{id} O$$

Next, $\text{K}\ k : \text{ISPT}_0(I,O)$ represents the constant $k$-valued functor which is the extension of the indexed container

$$I \xleftarrow{!} 0 \xrightarrow{!} X \xrightarrow{k} O$$

For a tagged sum $\text{Tag}\ A\ B : \text{ISPT}_0(I, O+O)$, let $[\![A]\!]$ be the extension of the indexed container $(f_A : P_A \to S_A, q_A, r_A)$ and $[\![B]\!]$ be the extension of the indexed container $(f_B : P_B \to S_B, q_B, r_B)$. Then

$$[\![\text{Tag}\ A\ B]\!]\, k\, [\text{inl}\ o] = [\![A]\!]\, [o] \quad [\![\text{Tag}\ A\ B]\!]\, k\, [\text{inr}\ o] = [\![B]\!]\, [o]$$

This is the extension of the indexed container

$$I \xleftarrow{[r_A, r_B]} P_A + P_B \xrightarrow{f_A + f_B} S_A + S_B \xrightarrow{q_A + q_B} O + O$$

For $\Delta f\, T : \text{ISPT}_0(I,O')$, let $[\![T]\!]$ be the extension of the indexed container $(S, P, q, r)$. Then $[\![\Delta f\, T]\!]$ represents the functor $\Delta_f . [\![T]\!]$ which maps an $I$-indexed family $k$ to the $O'$-indexed family with the following fibres

$$[\![\Delta f\, T]\!] k[o'] = (\Delta_f . [\![T]\!]) k[o'] = [\![T]\!] k[f o']$$

Thus $[\![\Delta f\, T]\!]$ is the extension of the indexed container $(S', P', q', r')$ defined by

$$\begin{array}{ll} S'[o'] = S[f\, o'] & P'[o', s] = P[s] \\ q'(o', s) = o' & r'((o', s), p) = r(s, p) \end{array}$$

For $\Sigma f T : \text{ISPT}_0(I,O')$, let $[\![T]\!]$ be the extension of the indexed container $(f : P \to S, q, r)$. Then $[\![\Sigma f\, T]\!]$ represents the functor $\Sigma_f . [\![T]\!]$ which maps an $I$-indexed family $k$ to the $O'$-indexed family with the following fibres

$$[\![\Sigma f\, T]\!]\, k\, [o'] = (\Sigma_f . [\![T]\!])\, k\, [o'] = \Sigma o : O . f\, o = o' \times [\![T]\!]\, k\, o$$

Thus $[\![\Sigma f\, T]\!]$ is the extension of the indexed container

$$I \xleftarrow{r} P \xrightarrow{f} S \xrightarrow{f.q} O'$$

Finally, for $\Pi f\, T : \mathrm{ISPT}_0(I, O')$, let $[\![T]\!]$ be the extension of the indexed container $(S, P, q, r)$. Then $[\![\Pi f\, T]\!]$ represents the functor $\Pi_f.[\![T]\!]$ which maps an $I$-indexed family $k$ to the $O'$-indexed family with the following fibres

$$[\![\Pi f\, T]\!]\, k\, [o'] = (\Pi_f.[\![T]\!])\, k\, [o'] = \Pi o : O.(f\, o = o') \to [\![T]\!]\, k\, o$$

Thus $[\![\Pi f\, T]\!]$ is the extension of the indexed container $(S', P', q', r')$ defined by

$$\begin{array}{ll} S'[o'] & = (\Pi\, o : O)\ (f o = o') \to S[o] \\ P'[o', g] & = (\Sigma\, o : O)\ (\Sigma\, q : f\, o = o')\ P[g(o, q)] \\ q'(o', g) & = o' \\ r'((o', g), (o, q, p)) & = r(g(o, q), p) \end{array}$$

Although appearing minimal at first, a whole host of other operators such as sums, products, fixed exponentials and composition are definable for ISPTs and hence for indexed containers

**Lemma 9.** $\mathrm{ISPT}_0$*s are closed under sums, products, fixed exponentials and composition.*

*Proof.* Let $A$ and $B$ be two $\mathrm{ISPT}_0$s. Then the coproduct $[\![A]\!]+[\![B]\!]$ is represented by the $\mathrm{ISPT}_0$ $\Sigma[id, id]\,(\mathrm{Tag}\ A\ B)$, while the product $[\![A]\!] \times [\![B]\!]$ is represented by $\Pi[id, id]\,(\mathrm{Tag}\ A\ B)$. If $K$ is a fixed object, then the fixed exponential $K \to [\![A]\!]$ is represented by $\Pi\pi_1\,(\Delta\pi_1\, A)$.

To prove that $[\![A]\!]\cdot[\![B]\!]$ is represented by an $\mathrm{ISPT}_0$, use induction on the structure of $A$. For example, if $A = \Pi f A'$, then

$$[\![\Pi f\ A']\!] \cdot [\![B]\!] = (\Pi f\ \cdot [\![A']\!]) \cdot [\![B]\!] = \Pi f\ \cdot ([\![A']\!] \cdot [\![B]\!])$$

By induction, $[\![A']\!] \cdot [\![B]\!]$ is represented by an ISPT we are done. Other cases are similarly easy.

We have already seen that $\mathrm{ISPT}_0$s are indexed containers. Now we can prove the reverse and thereby lift the closure properties of $\mathrm{ISPT}_0$s to indexed containers - this forms a completeness theorem in that all indexed containers can be defined as $\mathrm{ISPT}_0$s.

**Lemma 10.** *The grammar for* $\mathrm{ISPT}_0$*s is complete in that all indexed containers are* $\mathrm{ISPT}_0$*s. Further, indexed containers are closed under composition.*

*Proof.* Every indexed container has an extension of the form $\Sigma_q \cdot \Pi_f \cdot \Delta_r$. Each of these composites are clearly $\mathrm{ISPT}_0$s and, since $\mathrm{ISPT}_0$s are closed under composition, the indexed container is representable by a $\mathrm{ISPT}_0$. The composite of two indexed containers is thus representable as the composite of two $\mathrm{ISPT}_0$s which is therefore given by an $\mathrm{ISPT}_0$ and hence an indexed container.

Note that the ISPT constructors Tag, $\Delta, \Sigma$ and $\Pi$ act upon the output. A natural question is why we do not have similar constructors to act upon the input of a container. The composition theorem provides the answer to this question by ensuring that constructors to act on the input are already definable. Thus, to summarise, indexed containers and ISPTs define the same class of functors over slice categories. However, the former give a closed algebraic form of such functors as a triple of arrows, while the latter give a calculus for building indexed containers.

## 5 Categories of Indexed Containers

What are the interesting notions of morphism between indexed containers, with the same sets $I, O$ of input and output sorts? We clearly want to capture natural transformations between the extensions of the indexed containers in the same way that we did for containers. A natural set theoretical definition is that an indexed container morphism from $(S, P, q, r)$ to $(S', P', q', r')$ is a container morphism $(S, P) \longrightarrow (S', P')$ between the underlying containers that preserves the sorting information. This intuition provides the basis for the folloing definition.

**Definition 11.** *Given sets $I$ and $O$, a morphism between indexed containers from $(S, P, q, r)$ to $(S', P', q', r')$ is given by*

- *Underlying Morphism: A container morphism $\langle u, f \rangle$ of plain containers from $(S, P)$ to $(S', P')$. That is, a pair of $u : S \to S'$ and a reindexing morphism $f : \Pi s : S.P'[u\, s] \to P[s]$*
- *Output index preservation: $q' \cdot u = q : S \to O$*
- *Input index preservation: $(\Pi\, s : S, p : P'[u\, s])\ r'(u\, s, p) = r(s, f(s, p))$*

Categorically, these conditions amount to the following diagrams commuting.

$$\begin{array}{ccc} S & \xrightarrow{u} & S' \\ {\scriptstyle r}\downarrow & & \downarrow{\scriptstyle r'} \\ O & = & O \end{array} \qquad \begin{array}{ccc} P'[u\, s] & \xrightarrow{f\, s} & P[s] \\ {\scriptstyle r'(s,-)}\downarrow & & \downarrow{\scriptstyle r(s,-)} \\ I & = & I \end{array}$$

This definition of indexed container morphisms means that, for each $I$ and $O$, the indexed containers $\mathrm{IC}(I, O)$ form a category. The representation theorem of [4, 1] says that there is a full and faithful functor from the category of containers and container morphisms to the category of endofunctors on $\mathcal{C}$ and natural transformations. In other words, morphisms between containers correspond bijectively to natural transformations between their extensions. Note the generalisation to indexed containers is not an immediate generalisation of the bijection between container morphisms and their extensions as we cannot appeal to Yoneda.

**Lemma 12.** *There is a full and faithful embedding $[\![-]\!] : \mathrm{IC}(I, O) \to [\mathcal{C}/I, \mathcal{C}/O]$. That is, there is a bijection between natural transformations between the extensions of indexed containers and indexed container morphisms.*

Since naturality is straightforward, we concentrate on the bijection. A natural transformation between indexed containers $[\![f : S \to P, q, r]\!]$ and $[\![f' : S' \to P', q', r']\!]$. Then, for every $I$-indexed family $k$, there is a map in $\mathcal{C}/O$

$$\Sigma_q \Pi_f \Delta_r k \longrightarrow ([\![S', P', q', r']\!])k$$

Using the adjointness $\Sigma_q \dashv \Delta q$, this gives a map

$$\Pi_f \Delta_r k \longrightarrow \Delta_q([\![S', P', q', r']\!]k)$$

Now, we choose $k$ to be $r : P \to I$ and note that the identities over $P$ form a cone over $P \to^r I \leftarrow^r P$ and hence the universal property of the pullback $\Delta_r r$ gives a map $\delta_p : id_P \to \Delta_r r$. Noting that $id_P$ is the terminal object in $\mathcal{C}/P$, we can apply $\Pi_f$ to $\delta_p$ and then compose with the transpose of $\alpha_r$ above to give a map in $\mathcal{C}/S$

$$id_S \longrightarrow \Pi_f \Delta_r r \longrightarrow \Delta_q([\![S', P', q', r']\!]r)$$

Since any map $h : f \to g$ in a slice category is equivalent to one of the form $\Pi s : S.f[s] \to g[s]$, we thus have a term of type

$$\begin{aligned}
&\Pi s : S.id_s[s] \to \Delta_q([\![S', P', q', r']\!]r)[s] &= \\
&\Pi s : S.\Delta_q([\![S', P', q', r']\!]r)[s] &= \\
&\Pi s : S.[\![S', P', q', r']\!]P[q\,s] &= \\
&\Pi s : S.\Sigma s' : S'.q'\,s' = q\,s \ \wedge\ \Pi p' : P'[s']P[r'(s', p')] &= \\
&\Sigma u : S \to S'.\Pi s : S.q'(u\,s) = q\,s \ \wedge\ \Pi p' : P'[u\,s].P[r'(u\,s, p')] &= \\
&\Sigma u : S \to S'.\Pi s : S.q'(u\,s) = q\,s \ \wedge\ \Pi p' : P'[u\,s].\Sigma p : P[s].r(s, p) = r'(u\,s, p') = \\
&\Sigma u : S \to S'.\Pi s : S.q'(u\,s) = q\,s \wedge \\
&\qquad \Sigma f : P'[u\,s] \to P[s].\Pi p' : P'[u\,s].r(s, f\,p') = r'(u\,s, p')
\end{aligned}$$

which produces exactly an indexed container morphism as required.

In the reverse direction, note we can follow the chain of equalities backwards. That is, given a $u$ and $f$ as above, we create for every $I$-indexed family $k$, and arrow $\alpha_k : \Pi_f \Delta_r k \to \Delta_q([\![S', P', q', r']\!]k)$ in $\mathcal{C}/S$ by setting

$$\alpha_k s \phi = (u\,s, \phi') \text{ where } \phi'(p') = \phi(f\,p')$$

The bijection between container morphisms and natural transformations is very useful for practical reasoning as it allows us to reduce problems concerning polymorphic functions to problems concerning arithmetic on shapes and positions, for an clear example see the proof that reversing a list twice returns the original list [4]. More substantially, in the AI field, containers seem to be exactly the right notion to increase the expressive power of Bundy's *ellipses* [14] notation and to generalise ellipses from lists to other data structures. Similarly, we now give an example of reasoning with indexed container morphisms.

*Example 13 (Matrix Transposition is involutive).* Let $M_{m,n}(X)$ be the Nat $\times$ Nat-indexed family of $m$ by $n$ matrices which store data of type $X$. For every $m$ and $n$, there is only one shape of matrices which contains $\mathrm{Fin}[m] \times \mathrm{Fin}[n]$ locations for data of type $X$. Thus such matrices are defined by the indexed container $M : \mathrm{IC}(1, \mathrm{Nat} \times \mathrm{Nat})$ as follows

$$\begin{array}{ll} S & = \mathrm{Nat} \times \mathrm{Nat} \\ P[m,n] & = \mathrm{Fin}[m] \times \mathrm{Fin}[n] \end{array} \qquad \begin{array}{ll} q & = id \\ r(m,n,-) & = ! \end{array}$$

For each type $X$, matrix transpose should have type $M_{m,n}(X) \to M_{n,m}(X)$ and is represented by the indexed container morphism which sends the shape $(m,n)$ to the shape $(n,m)$ and a position $(j,k) : \mathrm{Fin}[m] \times \mathrm{Fin}[n]$ to the position $(k,j) : \mathrm{Fin}[n] \times \mathrm{Fin}[m]$. This is clearly and idempotent operation. While appearing overly simple, it is worth noticing that the simplicity is due to the presence of indexing information and the shapes and positions metaphor - formalisms lacking these concepts typically have to reinvent them and the proofs can quickly become cumbersome.

## 6 Fixed Points

We show how to construct initial and terminal coalgebras for indexed containers. We start with a non-parameterized construction, which in the case of initial algebras can be found in [20]. We observe that the construction dualizes, i.e. also works for terminal coalgebras. We extend this to a parameterized construction of initial algebras, which uses a modality $\Diamond$ on trees constructed using non-parameterized initial algebras. The parameterized construction also dualizes. Consequently, this gives parameterized initial algebras and terminal coalgebras of all indexed containers — this allows us to interpret the corresponding constructors for ISPTs. We provide sketch proofs but defer the detailed verification that the constructed objects have the required universal properties to the journal version of this work.

We first construct the least fixpoint of a non-parameterized container $(S,P,q,r) : \mathrm{IC}(O,O)$, that is an $O$-indexed family $\mu(S,P,q,r) : \mathcal{C}/O$ which is the carrier of the initial $[\![S,P,q,r]\!]$-algebra. Clearly, the initial algebra of the underlying container $S \triangleleft P$ is $W_{S \triangleleft P}$. Our task is to select those trees in $W_{S \triangleleft P}$ which respect the typing constraints of $q$ and $r$. That is, we must ensure that at every node the index of the shape stored there must be what is expected. Thus, we construct $\mu(S,P,q,r)[o] = \Sigma w : W_{S \triangleleft P}.\mathrm{Good}_{q,r}\,[(w,o)]$ as a subobject of $W_{S \triangleleft P}$ by defining an indexed family $\mathrm{Good}_{p,q} : \mathcal{C}/(W_{S \triangleleft P}) \times O$. First, define functions by primitive recursion over $W_{S \triangleleft P}$:

$$\begin{aligned} \mathrm{ideal}_r &: W_{S \triangleleft P} \to O \to W_{S \times O \triangleleft \Delta_{\pi_0} P} \\ \mathrm{actual}_q &: W_{S \triangleleft P} \to W_{S \times O \triangleleft \Delta_{\pi_0} P} \end{aligned}$$

by

$$\begin{aligned}\mathrm{ideal}_r\,(\sup s\,f) &= \lambda o.\sup\,(s,o)\,(\lambda p.\mathrm{ideal}_r\,(f\,p)\,(r\,(s,p)))\\ \mathrm{actual}_q\,(\sup s\,f) &= \sup\,(s,q\,s)\,(\lambda p.\mathrm{actual}_q\,(f\,p))\end{aligned}$$

$\mathrm{ideal}_r$ calculates a tree with labels assigned to subtrees by $r$, while $\mathrm{actual}_q$ assigns a label to each node in the tree based upon $q$. $\mathrm{Good}_{r,q}$ holds if both relabelled trees agree:

$$\mathrm{Good}_{q,r}\,[(w,o)] \triangleq (\mathrm{ideal}_r\,w\,o = \mathrm{actual}_q\,w)$$

Categorically, this corresponds to an equalizer.

We have shown in [4] that we can construct $M_{S \lhd P}$ that is the terminal coalgebra of the extension of $S \lhd P$ using only W-types via an internal limit construction. Given this, we can construct $\nu(S,P,q,r) : \mathcal{C}/O$ which is the carrier of the terminal $[\![S,P,q,r]\!]$-coalgebra by modifying the construction given above: we define $\nu(S,P,q,r)\,[o] = \Sigma w : M_{S \lhd P}.\mathrm{Good}'_{q,r}\,[(w,o)]$ as a subobject of $M_{S \lhd P}$. We observe that the recursive equations defining $\mathrm{ideal}_r$ and $\mathrm{actual}_q$ are also guarded and hence give rise to

$$\begin{aligned}\mathrm{ideal}'_r &: M_{S \lhd P} \to O \to M_{S \times O \lhd \Delta_{\pi_0} P}\\ \mathrm{actual}'_q &: M_{S \lhd P} \to M_{S \times O \lhd \Delta_{\pi_0} P}\end{aligned}$$

and we can define as above:

$$\mathrm{Good}'_{q,r}\,[(w,o)] \triangleq (\mathrm{ideal}'_r\,w\,o = \mathrm{actual}'_q\,w)$$

An example for the non-parameterized initial algebra construction is the definition of the *modality* $\Diamond\,R : \mathcal{C}/W_{S \lhd P}$ given $R : \mathcal{C}/S$ which will turn out to be useful for the parameterized case. Intuitively, $\Diamond\,R\,[w]$ corresponds to the proof-relevant interpretation of the predicate that $R$ holds for some shape $s : S$ in the tree $w : W_{S \lhd P}$. $\Diamond\,R$ is the initial algebra of

$$\begin{array}{ccc} P_\Diamond & \xrightarrow{f_\Diamond} & S_\Diamond \\ {\scriptstyle r_\Diamond}\downarrow & & \downarrow{\scriptstyle \pi_0} \\ W_{S \lhd P} & & W_{S \lhd P} \end{array}$$

where

$$\begin{aligned} S_\Diamond &= \Sigma \sup s\,f : W_{S \lhd P}.R\,[s] + P\,[s]\\ P_\Diamond\,[(w, \mathrm{inr}\,p)] &= 1\\ P_\Diamond\,[(w, \mathrm{inl}\,p)] &= 0\\ r_\Diamond\,(\sup s\,f, p) &= f\,p\end{aligned}$$

Type-theoretically $\Diamond\,R : \mathcal{C}/W_{S \lhd P}$ is given inductively by the following constructors:

$$\frac{r : R\,s}{\mathrm{stop}\,r : \Diamond\,R\,(\sup s\,f)} \qquad \frac{p : P\,s \quad d : \Diamond\,R\,(f\,p)}{\mathrm{step}\,p\,d : \Diamond\,R\,(\sup s\,f)}$$

In general inductive and coinductive definitions may be nested and hence we need a more general, parameterized, version of initial algebras and terminal coalgebras of indexed containers than that afforded by the non-parameterised case. That is, given an indexed container $(S, P, q, r) : \mathrm{IC}(I + O, O)$ and whose extension is a functor $\mathcal{C}/(I+O) \to \mathcal{C}/O$ or isomorphically $F : \mathcal{C}/I \to \mathcal{C}/O \to \mathcal{C}/O$, we construct an indexed container $\mu(S_\mu, P_\mu, q_\mu, r_\mu) : \mathrm{IC}(I, O)$ such that $[\![\mu(S, P, q, r)]\!]\, g$ is the initial $[\![S, P, q, r]\!]\, g$-algebra for every $g : \mathcal{C}/I$. To calculate the shapes $S_\mu$ and output indexing function $q_\mu$ of the least fixed point, note that we need not consider the parameterisation by $I$. Hence, we partition $P$ into positions above $I$ and $O$, say $P = P_O + P_I$ with indexing functions $r_I, r_O$ and then define $S_\mu$ and $q_\mu$ to be the non-parameterised least fixed point of the following container:

$$\begin{array}{ccccc} P_O & \xrightarrow{\mathrm{inr}} & P_O + P_I & \xrightarrow{f} & S \\ {\scriptstyle r_O}\downarrow & & & & \downarrow{\scriptstyle q} \\ O & & & & O \end{array}$$

We still have to construct the $I$-indexed family of positions given by $P_\mu$ and the map $r_\mu : P_\mu \to I$. Intuitively, $P_\mu$ is given by calculating for each node in the tree, the positions $P_I$ at that node. Hence we set

$$P_\mu[(w, g)] = \Diamond\, P_I\, [w]$$

Note here that $g$ is the proof that $w$ is a good tree arising from the non-parameterised initial algebra construction. Finally, define $r_\mu : \Pi(w, -) : S_\mu.\Diamond\, P_1\, w \to I$ using derivable primitive recursion over $\Diamond\, P_1\, [w]$:

$$\begin{aligned} r_\mu(\sup\, s\, f, g)\, (\mathrm{stop}\, p_1) &= r_I\, p_1 \\ r_\mu(\sup\, s\, f, g)\, (\mathrm{step}\, p_0\, d) &= r_\mu(f p_0, g')\, d \end{aligned}$$

where $g'$ is obtained from $g$ by noting that any branch of a good tree is good. This finishes the construction of the parameterized initial algebra.

The parameterized terminal coalgebra is constructed analogously, i.e. use the non-parameterised final coalgebra construction to calculate $S_\nu = \nu(S, P_O, q, r_O) : \mathcal{C}/O$, and notice that the inductive definition of $\Diamond$ also works for $M_{S \lhd P}$ giving rise to $\Diamond' R \in \mathcal{C}/MSP$ for $R : \mathcal{C}/S$. However, note that $\Diamond'$ is still constructed as an initial algebra reflecting the fact that paths in a potentially infinite tree are inductively given. Hence the definition of $r'$ by primitive recursion also works in the terminal coalgebra case.

Finally, we finish by completing the treatment of ISPTs. Note that we wanted to prove that the composition of ISPTs was an $IS$ but only did that for those ISPTs which contained no fixed points. We now rectify that situation

**Lemma 14.** ISPT*s are closed under composition.*

*Proof.* Consider the case of a composition $[\![\mu F \cdot G]\!]$ where $F : \mathrm{ISPT}(I + O, O)$ and $G : \mathrm{ISPT}(I', I)$. Notice that

$$[\![\mu F \cdot G]\!] = \mu[\![F]\!] \cdot [\![G + id_O]\!]$$

where $G + id_0 : \mathrm{IC}(I' + O, I' + O)$ is the obvious extension of $G$. By induction, $[\![F]\!] \cdot [\![G + id_O]\!]$ is the extension of an ISPT and hence so is $\mu F \cdot G$.

As a result the proofs at the end of section 4, lift to all ISPTs.

# 7 Applications

## 7.1 Indexed Containers in Epigram

One motivation for studying indexed containers is that they offer a straightforward foundation for inductive families [18], the key data structures supported by our dependently typed functional programming language Epigram [33]. As things stand, Epigram accepts inductive definitions which conform to the syntactic criteria for strict positivity set out by Luo [31], but indexed containers have the potential to yield a much more flexible and compositional approach, plus, their semantic character means that, the criteria for strict positivity may be expressed in Epigram itself.

In Epigram syntax, our finite set example looks like this:

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

As we have seen, this is readily represented as the least fixpoint of an indexed container. Indeed, any inductive family which Luo's rules admit may be expressed in this way. Luo admits inductive families $\mathsf{IFam}$ with a *telescope* [16] of indices, $\vec{o} : \vec{O}$, and constructors $\mathsf{c}_i$. Each constructor has a telescope of *non-recursive* arguments $\vec{a}$ and a some *recursive* arguments; each recursive argument $f_j$ has a telescope of parameters, $\vec{h}$, allowing infinitary branching. We write

$$\text{data } \frac{\vec{o} : \vec{O}}{\mathsf{IFam}\ \vec{o} : \star} \quad \text{where } \cdots \quad \frac{\vec{a} : \vec{A}_i \quad \cdots \quad f_j : \Pi \vec{h} : \vec{H}_{ij}.\ \mathsf{IFam}\ \vec{r}_{ij} \quad \cdots}{\mathsf{c}_i\ \vec{a}\ \vec{f} : \mathsf{IFam}\ \vec{q}_i} \quad \cdots$$

where neither the $\vec{A}_i$ nor the $\vec{H}_{ij}$ may refer to $\mathsf{IFam}$. In our setting, we can capture each dependent telescope $\vec{x} : \vec{X}$ as an iterated $\Sigma$-type, $\Sigma\vec{X}$. Luo's rules fix a sum-of-tuples representation for both shapes and positions: a shape is a choice of constructor, then a tuple of non-recursive arguments; a position is a choice of recursive argument, then a tuple of branching parameters. $\mathsf{IFam}$ becomes the least fixpoint of $[\![S, P, q, r]\!] : \mathcal{C}/\Sigma\vec{O} \to \mathcal{C}/\Sigma\vec{O}$ where

$$S = \Sigma i.\Sigma\vec{A}_i \qquad P[(i, \vec{a})] = \Sigma j.\Sigma\vec{H}_{ij}$$
$$q\ (i, \vec{a}) = (\vec{q}_i) \qquad r\ ((i, \vec{a}), (j, \vec{h})) = (\vec{r}_{ij})$$

Choosing a container representation of datatypes adds no *expressive* power. Any least fixpoint of an indexed container $[\![S, P, q, r]\!] : \mathcal{C}/(I + O) \to \mathcal{C}/O$ can be

turned into a clumsy but legitimate Epigram datatype:

$$\text{data}\ \frac{F : I \to \star \quad o : O}{\mathsf{Ind}\ F\ o : \star}\ \text{where}\ \frac{\begin{array}{c} s : S \quad f : \forall p{:}P[s]\,;\, i{:}I \Rightarrow r\ s\ p = \mathsf{inl}\ i \to F\ i \\ g : \forall p{:}P[s]\,;\, o{:}O \Rightarrow r\ s\ p = \mathsf{inr}\ o \to \mathsf{Ind}\ F\ o \end{array}}{\mathsf{con}\ s\ f\ g : \mathsf{Ind}\ F\ (q\ s)}$$

Rather, one strength of the container approach is its *modularity*. Luo's fixed format for strict positivity is somewhat conservative. It excludes, for example, the re-use of standard lists in the formation of *rose trees*, given by $\mathsf{Tree} = \mathsf{node}\,(\mathsf{List}\,\mathsf{Tree})$, and hence the re-use of library functionality. However, by broadening the notion of strictly positive operator to indexed containers in general, such convenient definitions become acceptable.

Moreover, identifying data structures as containers gives us direct access to functionality and theorems *for free*. We certainly get their functoriality. We also gain useful predicate transformers lifting predicates $\Phi$ from individual elements to containers of them, eg., '**somewhere** $\Phi$' and '**everywhere** $\Phi$'. From the programming side of the Curry-Howard correspondence, this is also very useful: if we are writing an interpreter for an embedded language, it is natural to define values as a family $\mathsf{Value} : \mathsf{Type} \to \star$; if contexts are containers of $\mathsf{Type}$, then **everywhere** $\mathsf{Value}$ equips them with a suitable notion of environment.

Containers have a great deal of structure ripe for exploitation by functional programmers. The techniques of *datatype generic programming* [11] can be used to compute instances of programming patterns, given the structure of the data to which they are being applied. Where Generic Haskell [15] delivers this functionality via a preprocessor, it is possible to *model* it via universes in a dependent type theory [12]. By identifying—and reflecting as data—the *grammar* of indexed containers (something similar to ISPTs above) to use as the basis for Epigram's datatype definitions, we provide generic programming directly for Epigram itself, not just for an Epigram model of another type system.

A striking example of generic programming is the *derivative* construction which lies at the heart of Huet's 'zipper' representation of one-hole contexts in tree-like structures [27]. The generic construction, observed in [32] has been implemented in Generic Haskell [24], and in Epigram [35]. Our container based analysis [6] readily extends to indexed containers and hence dependent types. Where we had

$$\partial(s : S \lhd P[s]) = (s : S\ ;\ p : P[s] \lhd \Sigma p' : P[s].\ p' \neq p)$$

we now have $\nabla$, taking an $\mathrm{IC}(I, O)$ container to an $\mathrm{IC}(I, O \times I)$ container:

$$\nabla(S, P, q, r) = \begin{pmatrix} s : S\ ;\ p : P[s], & \Sigma p' : P[s].\ p' \neq p, \\ \lambda(s, p).\ (q\ s, r\ (s, p)), & \lambda((s, p), (p', \_)).\ r\ (s, p')) \end{pmatrix}$$

### 7.2 The Refinement Calculus

In constructive type theory with a universe $*$, the contravariant powerset functor is defined as $\mathcal{P}(I) = I \to *$ and we often think of an element $p$ of $\mathcal{P}(I)$ as a

predicate over $I$ such that $p(i)$ is the type of proofs that $p$ holds of $i$. Categorically, $p$ corresponds to an object $f : X \to I$ in the slice over $I$ with the proofs that $i$ holds being given by the fibre $f\,[i]$. Using this notation, we can now give a computational representation of monotone (because functorial) predicate transformers as in the refinement calculus of Back and von Wright [9, 34]. A relation $R : O \to \mathcal{P}(I)$ between $O$ and $I$ determines two predicate transformers $\langle R\rangle$ and $[R]$, where

$$\begin{aligned}&\langle R\rangle, [R] : \mathcal{P}I \to \mathcal{P}O\\ &\langle R\rangle(U) \triangleq \{\, o : O \mid R(o) \between U \,\}\\ &[R](U) \triangleq \{\, o : O \mid R(o) \subseteq U \,\}\end{aligned}$$

Here $U \between V$ is shorthand for $(\exists i : I)U(i) \wedge V(i)$, while $\subseteq$ corresponds to fibrewise inclusion in the slice category interpretation. These are known as angelic and demonic nondeterministic update. When the relation happens to be the graph of a function $f$ (that is, total and single valued), it is direct that $\langle f\rangle = [f]$ (where the coercion of a function to its graph is left implicit). This predicate transformer is just substitution along $f$, sometimes called a functional or deterministic predicate transformer (and used to model assignment).

The key adjunction that relates these predicate transformers is

$$\frac{\langle R^{\mathrm{op}}\rangle U \subseteq V}{U \subseteq [R]V}$$

The adjunction above then immediately specialises to the familiar adjunctions $\exists_f = \langle f^{\mathrm{op}}\rangle \dashv (\cdot f) \dashv \forall_f = [f^{\mathrm{op}}]$.

These modalities can be transferred directly into the framework of slice categories in a locally cartesian closed category, in the following way. First, we may represent a relation from $O$ and $I$ as a *span*, which is to say a pair of morphisms $O \xleftarrow{p} X \xrightarrow{q} I$ sharing a common domain. In other terms, a span may be identified with a slice over the product $O \times I$. Such a span determines two degenerate indexed containers, namely

$$\begin{array}{ccc} X & \xrightarrow{q} & I \\ {\scriptstyle p}\downarrow & & \downarrow{\scriptstyle id} \\ I & & I \end{array} \qquad\qquad \begin{array}{ccc} X & \xrightarrow{id} & X \\ {\scriptstyle p}\downarrow & & \downarrow{\scriptstyle q} \\ I & & I \end{array}$$

The extensions of these containers may be computed by definition 5 and are respectively $\Sigma_q \cdot \Delta_p$ and $\Pi_q \cdot \Delta_p$. The converse of a span $O \xleftarrow{p} X \xrightarrow{q} I$ is obtained simply as $I \xleftarrow{q} X \xrightarrow{p} O$, and then the key adjunction of the refinement calculus is $\Sigma_p \cdot \Delta_q \dashv \Pi_q \cdot \Delta_p$. To establish this, we need only find a natural bijection

$$\frac{\Sigma_p(\Delta_q s) \longrightarrow t}{s \longrightarrow \Pi_q(\Delta_p)t}$$

This is immediate from the adjunctions $\Sigma_f \dashv \Delta_f \dashv \Pi_f$.

There is an immediate payoff for treating predicate transformers as indexed containers. The main normal form theorem in the refinement calculus ([10, Thm 13.10]) says that any monotone predicate transformer can be expressed as the composition of an angelic after a demonic update, which is to say in the form $\langle Q \rangle \cdot [R]$. So in fact, in the context of functors on slice categories of a locally cartesian closed category, we obtain a strengthened form of this theorem. Since $\langle Q \rangle$ and $[R]$ are extensions of indexed containers, so is their composite. Thus every monotone predicate transformer can be factored in the form $\langle r^{\mathrm{op}} \rangle \cdot [f^{\mathrm{op}}] \cdot \Delta_q$. That is, the relations used in the angelic and demonic updates may be chosen to be converse graphs of functions, with a third component which is a deterministic update.

### 7.3 Imperative Interfaces and Container Drivers

Notice that in definition 11 of container morphism, the input indices $I$ and output sorts $O$ are fixed, and we have a category $\mathrm{IC}(I, O)$. This is perfectly natural when the application for containers we have in mind is to represent data types or signatures over fixed spaces of input and output sorts. There is however another application to which containers and indexed containers may be put, namely to represent imperative interfaces, specifically command response interfaces. (This application has been explored in [22], [28], [23], and [21].) Then the indices represent states, and the containers are used to represent coalgebras over state spaces. In such a setting, it is natural to investigate notions of morphism between containers over different state spaces: between $\mathrm{IC}(I, O)$ and $\mathrm{IC}(I', O')$, particularly when $I = O$ and $I' = O'$. This reflects the fact that there may well be interfaces at higher or lower levels of abstraction reflected by the indexes.

A *command-response* interface consists of a client or *angel* and a server or *demon*. The angel starts by issuing an instruction or command, that the demon commits or performs, returning a response. The structure used in [21] to represent such an interface in constructive type theory is a quadruple $(O, A, D, n)$, called an *interaction structure* where

| | |
|---|---|
| $O : *,$ | *Current states* |
| $A : O \to *,$ | *Command sets/shapes* |
| $D : (\Pi\, o : O)\ A(o) \to *,$ | *Response sets/positions* |
| $n : (\Pi\, o : O, a : A(o))\ D(o, a) \to O$ | *Next state function* |

Replacing the use of universes with slices as mentioned at the beginning of section 7.2 means that such an interaction structure is simply an indexed container $(A', D', q', r') : \mathrm{IC}(O, O)$ where

$$\begin{array}{lll} A' & = \Sigma o : O.A(o) & q' = \pi_0 \\ D'[(o, a)] & = D(o, a) & r'((o, a), d) = n(o, a, d) \end{array}$$

Further, the predicate transformer $\mathcal{P}(O) \to \mathcal{P}(O)$ induced by such an interaction structure, namely

$$(\lambda\, U)\, (\lambda\, o)\ (\Sigma\, a : A(o))\ (\Pi\, d : D(o, a))\ U(n(o, a, d))$$

is (when we translate to slices) simply the extension of the indexed container defined above. This translation of interaction structures into indexed containers gives the following translation between the 'dynamic' metaphor of commands and responses, and the 'static' metaphor of shapes and positions:

| *static* | *dynamic* |
|---|---|
| shape | command |
| position | response |
| index $\begin{cases}\text{input}\\ \text{output}\end{cases}$ | state $\begin{cases}\text{next}\\ \text{current}\end{cases}$ |

As an example of how the dynamic and static views of indexed containers can provide structure not apparent in the other, we consider a different notion of morphism from that already considered. Among predicate transformers between different sets we find pairs

$$A : \mathcal{P}(X) \to \mathcal{P}(Y) \qquad D : \mathcal{P}(Y) \to \mathcal{P}(X)$$

such that $A \cdot D \subseteq 1_Y$ and $1_X \subseteq D \cdot A$. Then we call $(A, D)$ a simulation pair, or adjoint pair, with $A$ the lower and $D$ the upper. As a lower adjoint, $A$ commutes with unions, and so is determined with its values at singletons. It may therefore be identified with a relation $Q = A \cdot \{\_\} : X \to \mathcal{P}(Y)$. (The transformer $A$ is in fact the relational image determined by $Q$, sometimes written $\langle Q^{\text{op}} \rangle$ in refinement calculus notation.) As an upper adjoint, $D$ has the form $[Q^{\text{op}}]$, where $Q^{\text{op}}$ denotes the transpose of $Q$. The part $A$ as it were chooses a low-level encoding of a high-level state, while $D$ suffers a choice of a high-level decoding of a low-level state.

We take a morphism of indexed containers from $h$ to $l$ to be a simulation pair $(A, D)$ such that $h \subseteq D \cdot l \cdot A$ . This can be read: $h$ can be implemented by letting the environment pick a low-level 'encode' state, running the low-level program, then picking a suitable high-level 'decode' state. This inclusion (natural transformation) can be expressed in any of the equivalent forms $A \cdot h \subseteq l \cdot A$, $h \cdot D \subseteq D \cdot l$, and $A \cdot h \cdot D \subseteq l$. (We leave implicit the semantic 'extension' operator.)

We noted above that a simulation may be identified with a relation. It is shown in [21] that the relation representing a simulation between containers $h$ and $l$ which are homogeneous on $H$ and $L$ respectively is a coalgebra or post-fixed point for a certain container $h \multimap l$ homogeneous on $HL$. When the containers for the interfaces are given in the form of quadruples $(H, A_h, D_h, n_h)$ and $(L, A_l, D_l, n_l)$ form, the container for the simulation has the following extension

$$\begin{array}{rl|l} [\![h \multimap l]\!](Q)\langle s, s'\rangle \triangleq & \Sigma & t \;: A_h(s) \to A_l(s') \\ & & f : D_l(s') \cdot t \subseteq D_h(s) \\ \hline & \Pi & a \;: A_h(s) \\ & & d' : D_l(s', t(a)) \\ \hline & Q\langle n_h(s, a, f(a, d')), n_l(s', t(a), d')\rangle & \end{array}$$

This notion of simulation is known as "forward simulation" or "downward simulation" in the refinement calculus literature [17].

An interesting property of simulations concerns a notion of *safety*. Call $U : \mathcal{P}(S)$ *safe* (or an invariant) with respect to a predicate transformer $F : \mathcal{P}(S) \to \mathcal{P}(S)$ if accompanied by a morphism $U \subseteq F(U)$ in $\mathcal{P}(S)$. Then a simulation (which is itself a relation $Q$ safe with respect to $h \multimap l$) maps an $h$-safe predicate $U$ to the weakest $l$-safe predicate $\langle Q \rangle(U)$ necessary to guarantee $U$ at the high level.

The use of the symbol '$\multimap$' for linear implication is not an accident. It turns out (see [29] for full details), that $\multimap$ is right adjoint to the operator $\otimes$ between interaction structures, that can be considered to be a form of synchronous, or lock-step composition. Given interfaces $i_1(S_1, A_1, D_1, n_1)$ and $i_2 = (S_2, A_2, D_2, n_2)$, their synchronous composition $i_1 \otimes i_2 = (S, A, D, n)$ can be defined by

$$
\begin{array}{l}
S = S_1 \times S_2, \\
A(\langle s_1, s_2 \rangle) = A_1(s_1) \times A_2(s_2), \\
D(\langle s_1, s_2 \rangle, \langle a_1, a_2 \rangle) = D_1(s_1, a_1) \times D_2(s_2, a_2), \\
n(\langle s_1, s_2 \rangle, \langle a_1, a_2 \rangle, \langle d_1, d_2 \rangle) = \langle n_1(s_1, a_1, d_1) n_2(s_2, a_2, d_2) \rangle
\end{array}
$$

Another word for simulation might be 'driver'. In a operating system for a computer, there is usually a component of a type known as a driver. It's task is to implement a generic device 'driver' interface, and simulate various instances of it using the specific device interfaces supplied by the manufacturers of the devices. For example, the driver interface supplied by a disk driver may be a linear array of disk blocks, while the interface supplied by the manufacturer of a specific disk device may be in terms of seeks, head switching, sector sparing, and so on. One way to understand this notion of implementation is as a simulation. This explains the word 'driver' that occurs in the heading of this subsection.

As we have seen, interaction structures are indexed containers and hence this monoidal closed structure on interaction structures gives a monoidal closed structure on indexed containers. We hope in the future to find further examples of structure in either either the dynamic world of interaction structures or the static world of indexed families of datatypes that can be transported to the other world.

## 8 Related Research

There are many strands of research to which our paper is intimately connected. We comment on them but also explain where our work differs.

Indexed containers have previously been considered by Hyland and Gambino under the name of *dependent polynomials*. They proved that the least fixed point of an indexed container is an inductive family which can be constructed by $W$-types. However, at a technical level our work goes beyond that by considering a wider class of least fixed points, by considering greatest fixed points, defining a grammar for indexed containers, defining morphisms and proving the associated representation theorem and giving the above mentioned collection of definable

operations on indexed containers.

Fiore's work on generalised species extend indexed containers with certain structural quotients. However, dealing with these quotients involves the use of sophisticated mathematical structure such as coends etc which makes the resulting theory rather inaccessible to the broader community. By focusing on indexed containers we obtain a much simpler theory which is nevertheless expressive in covering most datatypes which arise naturally. At a technical level, unlike generalised species, our theory of indexed containers is not restricted to the category of Sets, our grammar for container constructors is different from Fiore's for generalised species and our work on container morphisms is completely new.

Perhaps the earliest publication about indexed containers (though not under that name) occurs in Petersson and Synek's paper [36] from 1989. They present rules extending Martin-Löf's type theory with a set constructor for 'tree sets' : families of mutually defined inductive sets, over a fixed index set. The context parameterising this constructor is essentially a homogeneous indexed container.They give a number of interesting examples (particularly an application to formal grammar), and sketch a reduction of the tree set constructor to W-types. They do not consider any notion of morphism between contexts, nor a coinductive counterpart of their constructor.

Inspired in part by Petersson and Synek's constructor, Hancock, Hyvernat and Setzer applied indexed (and unindexed) containers, under the name 'interaction structures' to the task of modeling imperative interfaces such as command-response interfaces in a number of publications. However they worked in Martin-Löf type theory with a universe, rather than a categorical framework, exploiting the universe in an essential way. The thrust of this work was to develop in such a fully constructive (ie. predicative) analysis of predicate transformers. They presented a coinductive counterpart of Petersson and Synek's constructor, and developed Moreover they considered a quite different notion of morphism between indexed containers, related more to the notion of 'forward simulation' in the refinement calculus of Back and von Wright [9] and Morgan [34], as well as to the notion of approximable map in formal topology. To a large extent, the precise connection of this notion with the one described in this paper remains to be understood. Hyvernat in particular discovered a model of intuitionistic multiplicative linear logic, which can be seen as a refinement of the relational model. Since linear phenomena also occur in the use of containers to model cursors in data structures [6], it may be interesting to gain a clear perspective on Hyvernat's analysis in the context of the differential calculus of indexed containers.

Finally, although related to our work, most of the research detailed above is much more theoretical than ours and consequently less accessible to the broader computer science audience. Our motivations are more computational as demonstrated by our applications and hence we believe this paper has substantial merit

in taking the developing theory of indexed containers and showing how it can be applied to more mainstream computer science.

## 9 Conclusions and Further Work

In this paper, we have shown that indexed containers provide a secure foundation, both type-theoretic and categorical for an expressive calculus in which to write and reason about a rich variety of datatypes. These comprise both inductively and coinductively defined families of types indexed over given datatypes. We have indicated, necessarily briefly, some applications to which we intend to apply this foundation. These are directly concerned with dependently typed programming.

We see this paper as setting the foundations to develop these applications. We are currently using indexed continers as the basis for the system of datatypes in Epigram and hope, in particular, to use them to define generic programs in Epigram which can be applied to all indexed containers. We also want to further investigate the relationship with interaction structures, formal topology and the refinement calculus. Ordered indexed containers also seem a natural object of study - either for generic traversal algorithms or as the basis of a theory of well founded orders for data structures for use in, say, rewriting. Finally, the use of position sets as primitve in the definition of indexed containers makes us think there is a relation to implicit computational space complexity.

## References


1. Michael Abbott. *Categories of Containers*. PhD thesis, University of Leicester, 2003.
2. Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Categories of containers. In Andrew Gordon, editor, *Proceedings of FOSSACS 2003*, number 2620 in Lecture Notes in Computer Science, pages 23–38. Springer-Verlag, 2003.
3. Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Representing nested inductive types using W-types. In *Automata, Languages and Programming, 31st International Colloqium (ICALP)*, pages 59 – 71, 2004.
4. Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Containers - constructing strictly positive types. *Theoretical Computer Science*, 342:3–27, September 2005. Applied Semantics: Selected Topics.
5. Michael Abbott, Thorsten Altenkirch, Neil Ghani, and Conor McBride. Derivatives of containers. In Martin Hofmann, editor, *Typed Lambda Calculi and Applications, TLCA 2003*, number 2701 in Lecture notes in Computer Science, pages 16–30. Springer, 2003.
6. Michael Abbott, Thorsten Altenkirch, Neil Ghani, and Conor McBride. $\partial$ for data: derivatives of data structures. *Fundamenta Informaticae*, 65(1,2):1–128, March 2005.
7. Peter Aczel. On relating type theories and set theories. *Lecture Notes in Computer Science*, 1657:1–??, 1999.
8. Thorsten Altenkirch and Bernhard Reus. Monadic presentations of lambda terms using generalized inductive types. In *Computer Science Logic*, 1999.

9. Ralph-Johan Back and Joakim von Wright. *Refinement calculus, A systematic introduction*. Graduate Texts in Computer Science. Springer-Verlag, New York, 1998.
10. Ralph-Johan Back and Joakim von Wright. *Refinement calculus, A systematic introduction*. Graduate Texts in Computer Science. Springer-Verlag, New York, 1998.
11. Roland Backhouse, Patrik Jansson, Johan Jeuring, and Lambert Meertens. Generic Programming—An Introduction. In S. Doaitse Sweierstra, Pedro R. Henriques, and José N. Oliveira, editors, *Advanced Functional Programming, Third International Summer School (AFP '98); Braga, Portugal*, volume 1608 of *Lecture Notes in Computer Science*, pages 28–115. Springer-Verlag, 1998.
12. Marcin Benke, Peter Dybjer, and Patrik Jansson. Universes for generic programs and proofs in dependent type th eory. *Nordic Journal of Computing*, 10:265–269, 2003.
13. Richard Bird and Ross Paterson. Generalised folds for nested datatypes. *Formal Aspects of Computing*, 11(3), 1999.
14. Alan Bundy and Julian Richardson. Proofs about lists using ellipsis. In *Logic Programming and Automated Reasoning*, pages 1–12, 1999.
15. Dave Clarke, Ralf Hinze, Johan Jeuring, Andres Lö h, and Jan de Wit. The Generic Haskell user's guide. Technical Report UU-CS-2001-26, Utrecht University, 2001.
16. Nicolas G. de Bruijn. Telescopic Mappings in Typed Lambda-Calculus. *Information and Computation*, 91:189–204, 1991.
17. W. DeRoever and Kai Engelhardt. *Data Refinement: Model-Oriented Proof Methods and Their Comparison*. Cambridge University Press, New York, NY, USA, 1999.
18. Peter Dybjer. Inductive Sets and Families in Martin-Löf's Type Theory. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*. CUP, 1991.
19. M. Fiore, G. D Plotkin, and D. Turi. Abstract syntax and variable binding. In *Proc. of 14th Ann. IEEE Symp. on Logic in Comp. Sci., LICS'99*, pages 193–202. IEEE CS Press, 1999.
20. Nicola Gambino and Martin Hyland. Wellfounded trees and dependent polynomial functors. In S. Berardi, M. Coppo, and F. Damiani, editors, *Types for Proofs and Programs (TYPES 2003)*, Lecture Notes in Computer Science, 2004.
21. Peter Hancock and Pierre Hyvernat. Programming interfaces and basic topology. *Annals of Pure and Applied Logic*, 2006.
22. Peter Hancock and Anton Setzer. Interactive Programs in Dependent Type Theory. In P.G. Clote and H. Schwichtenberg, editors, *Proceedings of the 14th International Workshop on Computer Science Logic (CSL 2000)*, number LNCS 1862 in Lecture Notes in Computer Science. Springer, August 2000.
23. Peter Hancock and Anton Setzer. Specifying interactions with dependent types. In *Workshop on subtyping and dependent types in programming, Portugal, 7 July 2000*, 2000. Electronic proceedings, available via http://www-sop.inria.fr/oasis/DTP00/Proceedings/proceedings.html.
24. Ralf Hinze, Johan Jeuring, and Andres Löh. Type-indexed data types. *Science of Computer Programmming*, 51:117–151, 2004.
25. Martin Hofmann. On the interpretation of type theory in locally cartesian closed categories. In *CSL*, pages 427–441, 1994.
26. Martin Hofmann. Syntax and semantics of dependent types. In A. M. Pitts and P. Dybjer, editors, *Semantics and Logics of Computation*, volume 14, pages 79–130. Cambridge University Press, Cambridge, 1997.

27. Gérard Huet. The Zipper. *Journal of Functional Programming*, 7(5):549–554, 1997.
28. Pierre Hyvernat. Predicate transformers and linear logic: yet another denotational model. In Jerzy Marcinkowski and Andrzej Tarlecki, editors, *18th International Workshop CSL 2004*, volume 3210 of *Lecture Notes in Computer Science*, pages 115–129. Springer-Verlag, September 2004.
29. Pierre Hyvernat. *A Logical Investigation of Interaction Systems*. PhD, Institut Mathmatique de Luminy, 2005.
30. Bart Jacobs. *Categorical Logic and Type Theory*. Number 141 in Studies in Logic and the Foundations of Mathematics. Elsevier, 1999.
31. Zhaohui Luo. *Computation and Reasoning: A Type Theory for Computer Science*. Oxford University Press, 1994.
32. Conor McBride. The Derivative of a Regular Type is its Type of One-Hole Contexts. Available at `http://www.dur.ac.uk/c.t.mcbride/diff.ps`, 2001.
33. Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1), 2004.
34. C. C. Morgan. *Programming from Specifications*. Prentice Hall International Series in Computer Science, 2nd edition, 1994.
35. 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.
36. Kent Petersson and Dan Synek. A set constructor for inductive sets in Martin-Löf's type theory. In *Proceedings of the 1989 Conference on Category Theory and Computer Science, Manchester, UK*, volume 389 of *LNCS*. Springer Verlag, 1989.
37. Robert Seely. Locally cartesian closed categories and type theory. *Mathematical Proceedings of the Cambridge Philosophical Society*, 95:33–48, 1984.
38. Thomas Streicher. *Semantics of Type Theory*. Birkhauser Boston Inc., 1991.

## A The Internal Language

In this conference paper we do not have the space to give a full definition of the internal language and so we give the basics and refer the reader to the literature for more details. Formally, the internal language links the extensional type theory MLW-EXT (see [7]) with finite types, $W$-types, a proof $true \neq false$ but without universes to locally cartesian closed categories with disjoint coproducts and initial algebras of container functors in one variable.

Categorically, we follow standard practise and represent an $I$-indexed family by an arrow $f : X \to I$ of the slice over $I$ which we think of mapping every element of $X$ to its index. On the other hand, type theoretically, we represent an $I$-indexed family by a judgement $i : I \vdash X[i]$ TYPE. Substitution is a fundamental operation in both the categorical and type theoretic domains - if $r : I' \to I$, then the $I'$ indexed family is computed categorically as the pullback $\Delta_r f : \Delta_r X \to I'$ or type theoretically as $i' : I' \vdash X[f\, i\, i']$. Pullback along $r$ extends categorically to a functor $\Delta_r : \mathcal{C}/I \to \mathcal{C}/I'$ and local cartesian closure implies that $\Delta_r$ has a left and right adjoint which we write $\Sigma_r$ and $\Pi_r$ - these correspond to the $\Sigma$ and

$\Pi$ types in the type theory. Similarly, $W$-types in the type theory correspond exactly to the categorical construction of initial algebras of container functors. Disjoint coproducts are required to ensure that coproducts behave well categorically - formally the pullback of distinct coprojections into a coproduct is always the initial object 0. Well behaved coproducts are guaranteed automatically in the type theory.

To translate from category theory to type theory we represent an $I$-indexed family $f : X \to I$ type theoretically either as $i : I \vdash f[i]$ or as $i : I \vdash X[i]$ depending which is more meaningful in the context. These are often known as the fibres above $i$ and are defined by $f[i] = X[i] = \Sigma x : X.fx = i$. The translations of $\Delta_r, \Sigma_r$ and $\Pi_r$ can similarly be given as

$$\begin{array}{l} \Delta_r f\ [i'] = f[r\ i'] \\ \Sigma_r f\ [i] \ = \Sigma i' : r[i].f[i'] \\ \Pi_r f\ [i] \ = \Pi i' : r[i].f[i'] \end{array}$$

In the category of $\mathsf{Sets}$, if $B$ is an $A$ indexed set, that is we have a set $B(a)$ for every $a \in A$, then

$$\begin{array}{l} \Sigma a : A.B = \{(a, b) | a \in A, b \in B(a)\} \\ \Pi a : A.B = \{f : A \to \bigcup_{a \in A} B(a) | \forall a \in A.f(a) \in B(a)\} \end{array}$$

In the reverse direction we construct a category from the type theory with objects given by types and morphisms given by terms. A judgement $i : I \vdash X[i]$ is represented in the slice over $I$ by the projection $\pi_1 : \Sigma i : I.X[i] \to I$ with substitution interpreted as pullback and the type theoretic quantifiers $\Sigma$ and $\Pi$ by their categorical counterparts. Full details of the use of internal languages to link type theory and its categorical semantics can be found in [37], [38], [25, 26], [30] and [1].