A Categorical Treatment of Ornaments[*]

Pierre-Evariste Dagand Conor McBride

Ornaments aim at taming the multiplication of special-purpose datatypes in dependently typed programming languages. In type theory, purpose is logic. By presenting datatypes as the combination of a structure and a logic, ornaments relate these special-purpose datatypes through their common structure. In the original presentation, the concept of ornament was introduced concretely for an example universe of inductive families in type theory, but it was clear that the notion was more general. This paper digs out the abstract notion of ornaments in the form of a categorical model. As a necessary first step, we abstract the universe of datatypes using the theory of polynomial functors. We are then able to characterise ornaments as cartesian morphisms between polynomial functors. We thus gain access to powerful mathematical tools that shall help us understand and develop ornaments. We shall also illustrate the adequacy of our model. Firstly, we rephrase the standard ornamental constructions into our framework. Thanks to its conciseness, this process gives us a deeper understanding of the structures at play. Secondly, we develop new ornamental constructions, by translating categorical structures into type theoretic artefacts.

The theory of inductive types is generally understood as the study of initial algebras in some appropriate category. A datatype definition is therefore abstracted away as a signature functor that admits a least fixpoint. This naturally leads to the study of polynomial functors [Gambino and Kock, 2010], a class of functors that all admit an initial algebra. These functors have been discovered and studied under many guises. In type theory, they were introduced by Martin-Löf under the name of *well-founded trees* [Martin-Löf, 1984, Moerdijk and Palmgren, 2000, Petersson and Synek, 1989], or W-types for short. Containers [Abbott et al., 2005a] and their indexed counterparts [Morris, 2007] generalise these definitions to a fibrational setting. Polynomial functors [Gambino and Hyland, 2004, Gambino and Kock, 2010] are the category theorists' take on containers, working in a locally cartesian-closed category.

There is a significant gap between this unified theoretical framework and the implementations of inductive types: in systems such as Coq [The Coq Development Team] or

[*]Revised April 21, 2013

Agda [Norell, 2007], datatypes are purely syntactic artefacts. A piece of software, the positivity checker, is responsible for checking that the definition entered by the user is valid, *i.e.* does not introduce a paradox. The power of the positivity checker depends on the bravery of its implementers: for instance, Coq's positivity checker is allegedly simple, therefore safer, but rather restrictive. On the other hand, Agda's positivity checker is more powerful, hence more complex, but also less trusted. For example, the latter checks the positivity of functions in datatype declarations, while the former conservatively rejects them. The more powerful the positivity checker, the harder it is to relate the datatype definitions to some functorial model.

An alternative presentation of inductive types is through a universe construction [Martin-Löf, 1984, Dybjer, 1991, Chapman et al., 2010]. The idea is to reflect the grammar of polynomial functors into type theory itself. Having internalised inductive types, we can formally manipulate them and, for example, create new datatypes from old. The notion of ornament [McBride, 2013] is an illustration of this approach. Ornaments arise from the realisation that inductive families can be understood as the integration of a data-*structure* together with a data-*logic*. The structure captures the dynamic, operational behavior expected from the datatype. It corresponds to, say, the choice between a list or a binary tree, which is governed by performance considerations. The logic, on the other hand, dictates the static invariants of the datatype. For example, by indexing lists by their length, thus obtaining vectors, we integrate a logic of length with the data. We can then take an $m \times n$ matrix to be a plainly rectangular m-vector of n-vectors, rather than a list of lists together with a proof that measuring each length yields the same result.

In dependent type theory, logic is purpose: when solving a problem, we want to bake the problem's invariants into the datatype we manipulate. Doing so, our code is correct by construction. The same data-structure will be used for different purposes and will therefore integrate as many logics: we assist to a multiplication of datatypes, each built upon the same structure. This hinders any form of code reuse and makes libraries next to pointless: every task requires us to duplicate entire libraries for our special-purpose datatypes.

Ornaments tame this issue by organising datatypes along their structure: given a datatype, an ornament gives an effective recipe to extend – introducing more information – and refine – providing a more precise indexing – the initial datatype. Applying that recipe gives birth to a new datatype that *shares the same structure* as the original datatype. Hence, ornaments let us evolve datatypes with some special-purpose logic without severing the structural ties between them. In an earlier work[Dagand and McBride, 2012], we have shown how that information can be used to regain code reuse.

The initial presentation of ornaments and its subsequent incarnation [McBride, 2013, Dagand and McBride, 2012] are however very syntactic and tightly coupled with their respective universe of datatypes. We are concerned that their syntactic nature obscures the rather simple intuition governing these definitions. In this paper, we give a semantic account of ornaments, thus exhibiting the underlying structure of the original definitions. To do so, we adopt a categorical approach and study ornaments in the framework of polynomial functors. Our contributions are the following:

- In Section 2, we formalise the connection between a universe-based presentation of datatypes and the theory of polynomial functors. In particular, we prove that the functors represented by our universe are equivalent to polynomial functors. This key result lets us move seamlessly from our concrete presentation of datatypes to the more abstract polynomial functors.
- In Section 3, we give a categorical presentation of ornaments as cartesian morphisms of polynomial functors. This equivalence sheds some light on the original definition of ornaments. It also connects ornaments to a mathematical object that has been widely studied: we can at last organise our universe of datatypes and ornaments on them into a category – in fact a framed bicategory [Shulman, 2009] – and start looking for categorical structures that would translate into interesting type theoretic objects.
- In Section 4, we investigate the categorical structure of ornaments. The contribution here is twofold. On one hand, we translate the original, type theoretic constructions – such as the ornamental algebra and the algebraic ornament – in categorical terms and uncover the building blocks out of which they were carved out. On the other hand, we interpret the mathematical properties of ornaments into type theory – such as the pullback of ornaments and the ornamentation of derivatives – to discover meaningful software artefacts that were previously unknown.

Being at the interface between type theory and category theory, this paper targets both communities. To the type theorist, we offer a more semantic account of ornaments and use the intuition thus gained to introduce new type theoretic constructions. Functional programmers of the non-dependent kind will find a wealth of examples that should help them grasp the more abstract concepts, both type theoretic and category theoretic. To the category theorist, we present a type theory, *i.e.* a programming language, that offers an interesting playground for categorical ideas. Our approach can be summarised as *categorically structured programming*. For practical reasons, we do not work on categorical objects directly: instead, we materialise these concepts through universes, thus reifying categorical notions through computational objects. Ornaments are merely an instance of that interplay between a categorical concept – cartesian morphism of polynomial functor – and an effective, type theoretic presentation – the universe of ornaments. To help bridge the gap between type theory and category theory, we have striven to provide the type theorist with concrete examples of the categorical notions and the category theorist with the computational intuition behind the type theoretic objects.

1 Categorical Toolkit

In this section, we recall a few definitions and results from category theory that will be used throughout this paper. None of these results are new – most of them are folklore

– we shall therefore not dwell on the details. However, to help readers not familiar with these tools, we shall give many examples, thus providing an intuition for these concepts.

1.1 Locally cartesian-closed categories

Locally cartesian-closed categories (LCCC) were introduced by Seely [1983] to give a categorical model of (extensional) dependent type theory. A key idea of that presentation is the use of adjunctions to model Π-types and Σ-types.

Definition 1 (Locally cartesian-closed category)**.** A locally cartesian-closed category is a category $\mathcal{E}$ that is pullback complete and such that, for $f : \mathcal{E}(X, Y)$, each base change functor $\Delta_f : \mathcal{E}_{/Y} \to \mathcal{E}_{/X}$, defined by pullback along f, has a right adjoint Π_f.

Throughout this paper, we work in a locally cartesian-closed category $\mathcal{E}$ with a terminal object $\mathbb{1}_{\mathcal{E}}$ and sums. An object $f : E \to I$ in the slice $\mathcal{E}_{/I}$ can be thought of as an I-indexed set, which we shall informally denote using a set comprehension notation $\{E_i \mid i \in I\}$, where E_i can be understood as the inverse image of f at i, or equivalently a fibre of the discrete fibration f.

By construction, the base change functor has a left adjoint $\Sigma_f = f \circ \_$. We therefore have the adjunctions

$$\Sigma_f \dashv \Delta_f \dashv \Pi_f$$

Using a set theoretic notation, the base change functor writes as reindexing by f:

$$\begin{array}{l} \Delta_f : \mathcal{E}_{/B} \to \mathcal{E}_{/A} \\ \Delta_f \ \{E_b \mid b \in B\} \mapsto \{E_{fa} \mid a \in A\} \end{array}$$

While the left and right adjoints correspond to the following definitions:

$$\begin{array}{l} \Sigma_f : \mathcal{E}_{/A} \to \mathcal{E}_{/B} \\ \Sigma_f \ \{E_a \mid a \in A\} \mapsto \left\{ \sum_{a \in A_b} E_a \mid b \in B \right\} \end{array} \qquad \begin{array}{l} \Pi_f : \mathcal{E}_{/A} \to \mathcal{E}_{/B} \\ \Pi_f \ \{E_a \mid a \in A\} \mapsto \left\{ \prod_{a \in A_b} E_a \mid b \in B \right\} \end{array}$$

Where $\sum$ and $\prod$ respectively represent the (set theoretic) disjoint union and cartesian product. Details of this construction can be found elsewhere [Mac Lane and Moerdijk, 1992].

The internal language of $\mathcal{E}$ corresponds to an extensional type theory denoted Set, up to bureaucracy [Curien, 1993]. This type theory comprises a unit type $\mathbb{1}$, Σ-types, Π-types, and equality is extensional. Syntactically, this type theory is specified by the following judgments:

$$\begin{array}{ccc}
\text{Formation rules:} & \text{Introduction rules:} & \text{Elimination rules:} \\[1em]
\dfrac{}{\Gamma \vdash \mathbb{1} : \textsc{Set}} & \dfrac{}{\Gamma \vdash * : \mathbb{1}} & \\[2em]
\dfrac{\Gamma \vdash A : \textsc{Set} \quad \Gamma \vdash B : \textsc{Set}}{\Gamma \vdash A + B : \textsc{Set}} & \dfrac{\Gamma \vdash a : A}{\Gamma \vdash \mathsf{inj}_l\, a : A} \quad \dfrac{\Gamma \vdash b : B}{\Gamma \vdash \mathsf{inj}_r\, b : B} & \dfrac{\Gamma \vdash f : A \to C \quad \Gamma \vdash g : B \to C \quad \Gamma \vdash x : A + B}{\Gamma \vdash \langle f, g \rangle\, x : C} \\[2em]
\dfrac{\Gamma \vdash S : \textsc{Set} \quad \Gamma ; x : S \vdash T : \textsc{Set}}{\Gamma \vdash (x : S) \times T : \textsc{Set}} & \dfrac{\Gamma \vdash a : S \quad \Gamma \vdash b : T[a/x]}{\Gamma \vdash (a, b) : (x : S) \times T} & \dfrac{\Gamma \vdash p : (x : S) \times T}{\Gamma \vdash \pi_0\, p : S} \quad \dfrac{\Gamma \vdash p : (x : S) \times T}{\Gamma \vdash \pi_1\, p : T[\pi_0\, p/x]} \\[2em]
\dfrac{\Gamma \vdash S : \textsc{Set} \quad \Gamma ; x : S \vdash T : \textsc{Set}}{\Gamma \vdash (x : S) \to T : \textsc{Set}} & \dfrac{\Gamma \vdash S : \textsc{Set} \quad \Gamma ; x : S \vdash t : T}{\Gamma \vdash \lambda_S x.\, t : (x : S) \to T} & \dfrac{\Gamma \vdash f : (x : S) \to T \quad \Gamma \vdash s : S}{\Gamma \vdash f\, s : T[s/x]}
\end{array}$$

We chose to work in an extensional model for simplicity. However, all the constructions presented in this paper have been modelled in Agda, an intuitionistic type theory.

1.2 Polynomials and polynomial functors

Polynomials [Gambino and Hyland, 2004, Gambino and Kock, 2010] provide a categorical model for indexed families [Dybjer, 1991] in a LCCC. Polynomials themselves are small, diagrammatic objects that admit a rich categorical structure. They are then *interpreted* as strong functors – the polynomial functors – between slices of $\mathcal{E}$. In this section, we shall illustrate the categorical definitions with the corresponding notion on (indexed) container [Petersson and Synek, 1989, Hancock and Hyvernat, 2006, Morris, 2007], an incarnation of polynomials in the internal language $\textsc{Set}$.

Definition 2 (Polynomial [Gambino and Kock, 2010, §1.1])**.** A polynomial is the data of 3 morphisms $f : B \to A$, $s : B \to I$, and $t : A \to J$ in $\mathcal{E}$. Conventionally, a polynomial is diagrammatically represented by $I \xleftarrow{s} B \xrightarrow{f} A \xrightarrow{t} J$.

Application 1 (Container)**.** In type theory, it is more convenient to work with (proof relevant) predicates rather than arrows. Hence, inverting the arrow $t : A \to J$, we obtain a predicate $S : J \to \textsc{Set}$ – called the *shapes*. Similarly, inverting $f : B \to A$, we obtain a predicate $P : \forall j.\ S\ j \to \textsc{Set}$ – called the *positions*. The indexing map s remains unchanged but, following conventional notation, we rename it n – the *next index* function. We obtain the following definition:

$$\begin{cases} S : J \to \textsc{Set} \\ P : S\ j \to \textsc{Set} \\ n : P\ sh \to I \end{cases}$$

Note that, to remove clutter, we (implicitly) universally quantify unbound type variables, such as j in the definition of P or sh in the definition of n. The data of S, P, and n is called a *container* and is denoted $S \lhd^n P$. The class of containers indexed by I and J is denoted $\mathit{ICont}\ I\ J$.

Remark 1 (Intuition). Polynomials, and more directly containers, can be understood as multi-sorted signatures. The indices specifies the sorts. The shapes at a given index specify the set of symbols at that sort. The positions specify the arities of each symbol. The next index function specifies, for each symbol, the sort of its arguments.

Definition 3 (Polynomial functor [Gambino and Kock, 2010, §1.4]). We *interpret* a polynomial $F : I \xleftarrow{s} B \xrightarrow{f} A \xrightarrow{t} J$ into a functor, conventionally denoted P_F, between slices of $\mathcal{E}$ with the construction

$$P_F \triangleq \mathcal{E}_{/I} \xrightarrow{\Delta_s} \mathcal{E}_{/B} \xrightarrow{\Pi_f} \mathcal{E}_{/A} \xrightarrow{\Sigma_t} \mathcal{E}_{/J}$$

A functor F is called *polynomial* if it is isomorphic to the interpretation of a polynomial, *i.e.* there exists s, f, and t such that $F \cong \Sigma_t \Pi_f \Delta_s$.

Application 2 (Interpretation of container). Unfolding this definition in the internal language, we interpret a container as, first, a choice (Σ-type) of shape ; then, for each (Π-type) position, a variable X taken at the next index n for that position:

$$\begin{array}{l} [\![(C : \mathit{ICont}\ I\ J)]\!]_{\mathsf{Cont}}\ (X : I \to \textsc{Set})\ :\ J \to \textsc{Set} \\ [\![S \lhd^n P]\!]_{\mathsf{Cont}}\ X\ \mapsto\ \lambda j.\ (sh : S\ j) \times ((pos : P\ sh) \to X\ (n\ pos)) \end{array}$$

hence justifying the name *polynomial functor*: a polynomial interprets into an S-indexed sum of monomials X taken at some exponent $pos : P\ sh$, or put informally:

$$[\![S \lhd^n P]\!]_{\mathsf{Cont}}\ \{X_i \mid i \in I\} \mapsto \left\{ \sum_{sh \in S_j} \prod_{pos \in P_{sh}} X_{n\ pos} \mid j \in J \right\}$$

Example 1 (Container: natural number). Natural numbers are described by the signature functor $X \mapsto 1 + X$. The corresponding container is given in Figure 1a. There are two shapes, one to represent the 0 case, the other to represent the successor case, suc. For the positions, none is offered by the 0 shape, while the suc shape offers one. Note that the signature functor is not indexed: the container is therefore indexed by the unit set and the next index is trivial.

Example 2 (Container: list). The signature functor describing a list of parameter A is $X \mapsto 1 + A \times X$. The container is presented Figure 1b. Note the similarity with natural numbers. There are $1 + A$ shapes, *i.e.* either the empty list nil or the list constructor cons of some $a : A$. There are no subsequent position for the nil shape, while one position is offered by the cons shapes. Indices are trivial, for lists are not indexed.

$$\mathsf{NatCont} \triangleq \begin{cases} \mathsf{S}_{\mathsf{Nat}}\ (*:\mathbb{1}) & : \ \textsc{Set} \\ \mathsf{S}_{\mathsf{Nat}}\quad * & \mapsto \mathbb{1}+\mathbb{1} \\ \mathsf{P}_{\mathsf{Nat}}\ (sh:\mathsf{S}_{\mathsf{Nat}}\ *) & : \ \textsc{Set} \\ \mathsf{P}_{\mathsf{Nat}}\quad (\mathsf{inj}_l\ *) & \mapsto \mathbb{0} \\ \mathsf{P}_{\mathsf{Nat}}\quad (\mathsf{inj}_r\ *) & \mapsto \mathbb{1} \\ \mathsf{n}_{\mathsf{Nat}}\ (pos:\mathsf{P}_{\mathsf{Nat}}\ sh) & : \ \mathbb{1} \\ \mathsf{n}_{\mathsf{Nat}}\quad pos & \mapsto * \end{cases}$$

(a) Natural number

$$\mathsf{ListCont}_A \triangleq \begin{cases} \mathsf{S}_{\mathsf{List}}\ (*:\mathbb{1}) & : \ \textsc{Set} \\ \mathsf{S}_{\mathsf{List}}\quad * & \mapsto \mathbb{1}+A \\ \mathsf{P}_{\mathsf{List}}\ (sh:\mathsf{S}_{\mathsf{List}}\ *) & : \ \textsc{Set} \\ \mathsf{P}_{\mathsf{List}}\quad (\mathsf{inj}_l\ *) & \mapsto \mathbb{0} \\ \mathsf{P}_{\mathsf{List}}\quad (\mathsf{inj}_r\ a) & \mapsto \mathbb{1} \\ \mathsf{n}_{\mathsf{List}}\ (pos:\mathsf{P}_{\mathsf{List}}\ sh) & : \ \mathbb{1} \\ \mathsf{n}_{\mathsf{List}}\quad pos & \mapsto * \end{cases}$$

(b) List

$$\mathsf{VecCont}_A \triangleq \begin{cases} \mathsf{S}_{\mathsf{Vec}}\ (n:\mathsf{Nat}) & : \ \textsc{Set} \\ \mathsf{S}_{\mathsf{Vec}}\quad \mathsf{0} & \mapsto \mathbb{1} \\ \mathsf{S}_{\mathsf{Vec}}\quad (\mathsf{suc}\, n) & \mapsto A \\ \mathsf{P}_{\mathsf{Vec}}\ (n:\mathsf{Nat})\ (sh:\mathsf{S}_{\mathsf{Vec}}\ n) & : \ \textsc{Set} \\ \mathsf{P}_{\mathsf{Vec}}\quad \mathsf{0}\quad * & \mapsto \mathbb{0} \\ \mathsf{P}_{\mathsf{Vec}}\quad (\mathsf{suc}\, n)\quad a & \mapsto \mathbb{1} \\ \mathsf{n}_{\mathsf{Vec}}\ (n:\mathsf{Nat})\ (sh:\mathsf{S}_{\mathsf{Vec}}\ n)\ (pos:\mathsf{P}_{\mathsf{Vec}}\ pos) & : \ \mathsf{Nat} \\ \mathsf{n}_{\mathsf{Vec}}\quad (\mathsf{suc}\, n)\quad a\quad * & \mapsto n \end{cases}$$

(c) Vector

Figure 1: Examples of containers

Example 3 (Container: vector)**.** To give an example of an indexed datatype, we consider vectors, *i.e.* lists indexed by their length. The signature functor of vectors is given by $\{X_n \mid n \in \mathsf{Nat}\} \mapsto \{n = \mathsf{0} \mid n \in \mathsf{Nat}\} + \{A \times X_{n-1} \mid n \in \mathsf{Nat}^*\}$ where the empty vector nil requires the length n to be $\mathsf{0}$, while the vector constructor cons must have a length n of at least one and takes its recursive argument X at index $n-1$. The container representing this signature is given Figure 1c. At index $\mathsf{0}$, only the nil shape is available while index $\mathsf{suc}\, n$ offers a choice of $a : A$ shapes. As for lists, the nil shape has no subsequent position while the cons shapes offer one. It is necessary to compute the next index (*i.e.* the length of the tail) only when the input index is $\mathsf{suc}\, n$, in which case the next index is n.

We leave it to the reader to verify that the interpretation of $\mathsf{NatCont}$ (Example 1), $\mathsf{ListCont}$ (Example 2), and $\mathsf{VecCont}$ (Example 3) are indeed equivalent to the signature functors we aimed at representing. With this exercise, one gains a better intuition of the respective contribution of shapes, positions, and the next index to the encoding of signature functors.

Definition 4 (Polynomial morphism [Gambino and Kock, 2010, §3.8])**.** A morphism from $F : I \xleftarrow{s'} B \xrightarrow{f'} A \xrightarrow{t'} J$ to $G : K \xleftarrow{s} D \xrightarrow{f} C \xrightarrow{t} L$ is uniquely represented – up to the choice of pullback – by the diagram:

$$\begin{tikzcd}
I \arrow[dd, "u"'] & B \arrow[l, "s'"'] \arrow[r, "f'"] & A \arrow[r, "t'"] \arrow[d, equal] & J \arrow[dd, "v"] \\
 & D' \arrow[u, "\omega"] \arrow[r] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & A \arrow[d, "\alpha"] & \\
K & D \arrow[l, "s"] \arrow[r, "f"'] & C \arrow[r, "t"'] & L
\end{tikzcd}$$

Example 4 (Container morphism)**.** Let $u : I \to K$ and $v : J \to L$. A morphism from a container $S' \triangleleft^{n'} P'$ to a container $S \triangleleft^{n} P$ framed by u and v is given by two functions and a coherence condition:

$$\begin{cases} \sigma : S'\ j \rightarrow S\ (v\ j) \\ \rho : P\ (\sigma\ sh') \rightarrow P'\ sh' \\ q : \forall sh' : S'\ j.\ \forall pos : P\ (\sigma\ sh').\ u\ (n'\ (\rho\ pos)) = n\ pos \end{cases}$$

Remark that σ and ρ correspond exactly to their diagrammatic counterparts, respectively α and ω, while the coherence condition q captures the commutativity of the left square. Commutativity of the right square is ensured by construction, since we reindex S by v in the definition of σ.

A container morphism, *i.e.* the data σ, ρ, and q, is denoted $\sigma \blacktriangleleft \rho$ (leaving implicit the coherence condition). The hom-set of morphisms from $S' \lhd^{n'} P'$ to $S \lhd^{n} P$ framed by u and v is denoted $S' \lhd^{n'} P' \overset{u}{\underset{v}{\Longrightarrow}} S \lhd^{n} P$.

In this paper, we are particularly interested in a sub-class of polynomial morphisms: the class of cartesian morphisms. Cartesian morphisms represent only cartesian natural transformations – *i.e.* for which the naturality square forms a pullback.

Definition 5 (Cartesian morphism [Gambino and Kock, 2010, §3.14])**.** A cartesian morphism from $F : I \xleftarrow{s'} B \xrightarrow{f'} A \xrightarrow{t'} J$ to $G : K \xleftarrow{s} D \xrightarrow{f} C \xrightarrow{t} L$ is uniquely represented by the diagram:

$$\begin{tikzcd}
I \arrow[d, "u"'] & B \arrow[l] \arrow[r] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & A \arrow[r] \arrow[d, "\alpha"] & J \arrow[d, "v"] \\
K & D \arrow[l] \arrow[r] & C \arrow[r] & L
\end{tikzcd}$$

Where the α is pulled back along f, as conventionally indicated by the right angle symbol.

Application 3 (Cartesian morphism of containers)**.** In the internal language, a cartesian morphism from $S' \lhd^{n'} P'$ to $S \lhd^{n} P$ framed by u and v corresponds to the triple:

$$\begin{cases} \sigma : S'\ j \rightarrow S\ (v\ j) \\ \rho : \forall sh' : S'\ j.\ P\ (\sigma\ sh') = P'\ sh' \\ q : \forall sh' : S'\ j.\ \forall pos : P\ (\sigma\ sh').\ u\ (n'\ pos) = n\ pos \end{cases}$$

The diagrammatic morphism α translates into an operation on shapes, denoted σ. The pullback condition translates into a proof ρ that the source positions are indeed obtained by pulling back the target positions along σ. As for the indices, the coherence condition q captures the commutativity of the left square. Commutativity of the right square is ensured by construction, since we reindex S by v in the definition of σ.

A cartesian morphism is denoted $\sigma \blacktriangleleft^{c}$, leaving implicit the proof obligations. The hom-set of cartesian morphisms from $S' \lhd^{n'} P'$ to $S \lhd^{n} P$ is denoted $S' \lhd^{n'} P' \overset{u}{\underset{v}{\Longrightarrow}}{}^{c} S \lhd^{n} P$. Because polynomials and containers conventionally use different notations, we sum-up the equivalences in Figure 2.

Polynomial	Container	Obtained by
$t : A \to J$	$S : J \to \textsc{Set}$	Inverse image
$f : B \to A$	$P : S\ j \to \textsc{Set}$	Inverse image
$s : B \to I$	$n : P\ sh \to I$	Identity
$\alpha : A \to C$	$\sigma : S'\ j \to S\ (v\ j)$	Identity

Figure 2: Translation polynomial/container

Example 5 (Cartesian morphism)**.** We build a cartesian morphism from $\mathsf{ListCont}_A$ (Example 2) to $\mathsf{NatCont}$ (Example 1) by mapping shapes of $\mathsf{ListCont}_A$ to shapes of $\mathsf{NatCont}$:

$$\begin{array}{llll}
\sigma\ (sh_l : \mathsf{S}_{\mathsf{List}}\ *) & : & \mathsf{S}_{\mathsf{Nat}}\ * & \\
\sigma \quad (\mathsf{inj}_l\ *) & \mapsto & \mathsf{inj}_l\ * & \text{-- } \textit{nil to 0} \\
\sigma \quad (\mathsf{inj}_r\ a) & \mapsto & \mathsf{inj}_r\ * & \text{-- } \textit{cons}\, a \textit{ to suc}
\end{array}$$

We are then left to check that positions are isomorphic: this is indeed true, since, in the $\mathsf{nil}/\mathsf{0}$ case, there is no position while, in the $\mathsf{cons}/\mathsf{suc}$ case, there is only one position. The coherence condition is trivially satisfied, since both containers are indexed by $\mathbb{1}$. We shall relate this natural transformation to the function computing the length of a list in Example 15.

We have seen that polynomials interpret to (polynomial) functors. We therefore expect morphisms of polynomials to interpret to natural transformations between these functors.

Definition 6 (Interpretation of polynomial morphism [Gambino and Kock, 2010, §2.1 and §2.7])**.** For the morphism of polynomials given in Definition 4, we construct the following natural transformation:

$$\begin{tikzcd}
\mathcal{E}_{/I} \arrow[r, "\Delta_{s'}"] \arrow[d, "\Sigma_u"'] \arrow[dr, phantom, "\cong"] & \mathcal{E}_{/B} \arrow[r, equal] \arrow[d, "\Delta_\omega"'] \arrow[dr, phantom, "\Downarrow \eta"] & \mathcal{E}_{/B} \arrow[r, "\Pi_{f'}"] \arrow[dr, phantom, "\cong"] & \mathcal{E}_{/A} \arrow[rr, "\Sigma_{t'}"] \arrow[d, equal] \arrow[drr, phantom, "\cong"] & & \mathcal{E}_{/J} \arrow[d, equal] \\
\mathcal{E}_{/K} \arrow[r, "\Delta_{s \circ \alpha^\dagger}"] \arrow[d, equal] \arrow[dr, phantom, "\cong"] & \mathcal{E}_{/D'} \arrow[rr, "\Pi_{f^\dagger}"'] \arrow[ur, "\Pi_\omega"'] \arrow[drr, phantom, "\cong"] & & \mathcal{E}_{/A} \arrow[rr, "\Sigma_t"] \arrow[dr, "\Sigma_\alpha"] \arrow[d, phantom, "\Downarrow \epsilon"] & & \mathcal{E}_{/J} \arrow[d, "\Sigma_v"] \arrow[dl, phantom, "\cong"] \\
\mathcal{E}_{/K} \arrow[r, "\Delta_s"'] & \mathcal{E}_{/D} \arrow[rr, "\Pi_f"'] \arrow[u, "\Delta_{\alpha^\dagger}"'] & & \mathcal{E}_{/C} \arrow[r, equal] \arrow[u, "\Delta_\alpha"] & \mathcal{E}_{/C} \arrow[r, "\Sigma_t"'] & \mathcal{E}_{/L}
\end{tikzcd}$$

The diagrammatic construction of the interpretation on morphism is perhaps intimidating. Its actual simplicity is revealed by containers, in the internal language.

Example 6 (Interpretation of container morphism)**.** A morphism from $S' \lhd^{n'} P'$ to $S \lhd^{n} P$ simply maps shapes S' to shapes S covariantly using σ and maps positions P

to positions P' contravariantly using ρ:

$$\begin{aligned}
&[\![(m : S' \lhd^{n'} P' \overset{u}{\underset{v}{\Longrightarrow}} S \lhd^{n} P)]\!]_{\mathsf{Cont}}\ (xs : [\![S' \lhd^{n'} P']\!]_{\mathsf{Cont}}\ X)\ :\ [\![S \lhd^{n} P]\!]_{\mathsf{Cont}}\ X \\
&[\![\sigma \blacktriangleleft \rho]\!]_{\mathsf{Cont}}\ (sh', Xs)\ \mapsto\ (\sigma\ sh', Xs \circ \rho)
\end{aligned}$$

Note that, thinking of X as being parametrically quantified, there is not much choice anyway: the shapes are in a covariant position while positions are on the left on an arrow, *i.e.* contravariant position.

Hancock and Hyvernat [2006] present these morphisms as defining a (constructive) simulation relation: having a morphism from $C' \triangleq S' \lhd^{n'} P'$ to $C \triangleq S \lhd^{n} P$ gives you an effective recipe to *simulate* the behavior of C' using C: a choice of shape sh' in S' is translated to a choice of shape in S through σ while the subsequent response $pos : P(\sigma\, sh')$ is back-translated through ρ to a response in P'.

We shall need the following lemma that creates polynomial functors from a cartesian natural transformations to a polynomial functor:

Lemma 1 (Lemma 2.2 [Gambino and Kock, 2010])**.** *Let $P_F : \mathcal{E}_{/I} \to \mathcal{E}_{/J}$ be a polynomial functor. Let Q a functor from $\mathcal{E}_{/I}$ to $\mathcal{E}_{/J}$.*

If $\phi : Q \dot{\to} P_F$ is a cartesian natural transformation, then Q is also a polynomial functor.

Finally, we shall need the following algebraic characterization of polynomial functors:

Lemma 2 (Corollary 1.14 [Gambino and Kock, 2010])**.** *The class of polynomial functors is the smallest class of functors between slices of $\mathcal{E}$ containing the pullback functors and their adjoints, and closed under composition and natural isomorphism.*

1.3 Framed bicategory

We have resisted the urge of defining a category of polynomials and polynomial functors. Such a category can be defined for a given pair of indices I and J, with objects being polynomials indexed by I and J (Definition 2) and morphisms (Definition 4) specialised to the case where $u = \mathrm{id} : I \to I$ and $v = \mathrm{id} : J \to J$.

From there, we are naturally lead to organise polynomials and their indices in a 2-category. However, this fails to capture the fact that indices have a life of their own: it makes sense to have morphisms between differently indexed functors, *i.e.* between different slices of $\mathcal{E}$. Indeed, morphisms between indices – the objects – *induce* 1-morphisms. The 2-categorical presentation does not capture this extra power. Following the steps of Gambino and Kock [2010], we organize polynomials and their functors in a *framed bicategory* [Shulman, 2009]. We refer the reader to that latter paper for a comprehensive presentation of this concept. A framed bicategory is a double category with some more structure. We therefore recall the definition of a double category and give a few examples.

Definition 7 (Double category)**.** A double category $\mathbb{D}$ consists of a category of objects $\mathbb{D}_0$ and a category of morphisms $\mathbb{D}_1$, together with structure functors:

$$\begin{tikzcd}
\mathbb{D}_0 \arrow[r, "U"] & \mathbb{D}_1 \arrow[l, bend right, "L"'] \arrow[l, bend left, "R"] & \mathbb{D}_1 \times_{\mathbb{D}_0} \mathbb{D}_1 \arrow[l, "\odot"']
\end{tikzcd}$$

Satisfying the usual axioms of categories relating the left frame L, right frame R, identity U, and composition $\odot$.

Example 7 (Double category *PolyFun*$_\mathcal{E}$ [Gambino and Kock, 2010, §3.5])**.** The double category *PolyFun*$_\mathcal{E}$ is defined as follow:

- Objects (*i.e.* objects of $\mathbb{D}_0$): slices $\mathcal{E}_{/I}$
- Vertical arrows (*i.e.* morphism of $\mathbb{D}_0$): colift $\Sigma_u : \mathcal{E}_{/I} \to \mathcal{E}_{/K}$, for $u : I \to K$ in $\mathcal{E}$
- Horizontal arrows (*i.e.* objects of $\mathbb{D}_1$) with left frame $\mathcal{E}_{/I}$ and right frame $\mathcal{E}_{/J}$: polynomial functor $P_F : \mathcal{E}_{/I} \to \mathcal{E}_{/J}$
- Squares (*i.e.* morphisms of $\mathbb{D}_1$) with left frame Σ_u and right frame Σ_v: strong natural transformation ϕ:

$$\begin{tikzcd}
\mathcal{E}_{/I} \arrow[r, "P_F"] \arrow[d, "\Sigma_u"'] \arrow[dr, phantom, "\Downarrow \phi"] & \mathcal{E}_{/J} \arrow[d, "\Sigma_v"] \\
\mathcal{E}_{/K} \arrow[r, "P_G"'] & \mathcal{E}_{/L}
\end{tikzcd}$$

Example 8 (Double category *Poly*$_\mathcal{E}$)**.** The double category *Poly*$_\mathcal{E}$ is defined as follow:

- $\mathbb{D}_0 =$ Set, *i.e.*
  - Objects: index set $I, J, K, L, \ldots$
  - Vertical arrows: $u : I \to K$, $v : J \to L$, …
- Horizontal arrows (*i.e.* objects of $\mathbb{D}_1$) with left frame I and right frame J: polynomial indexed by I and J
- Squares (*i.e.* morphisms of $\mathbb{D}_1$) with left frame Σ_u and right frame Σ_v: polynomial morphism, with u closing the diagram on the left and v closing the diagram on the right.

We are naturally tempted to establish a connection between the double category *Poly*$_\mathcal{E}$ of polynomials and the double category *PolyFun*$_\mathcal{E}$ of polynomial functors. We expect the interpretation of polynomials to play that role, behaving, loosely speaking, as a functor from *Poly*$_\mathcal{E}$ to *PolyFun*$_\mathcal{E}$. To formalize that intuition, we need a notion of functor between double categories:

Definition 8 (Lax double functor [Shulman, 2009, §6.1])**.** A lax double functor $F : \mathbb{C} \to \mathbb{D}$ consists of:

- Two functors $F_0 : \mathbb{C}_0 \to \mathbb{D}_0$ and $F_1 : \mathbb{C}_1 \to \mathbb{D}_1$ such that $L \circ F_1 = F_0 \circ L$ and $R \circ F_1 = F_0 \circ R$.
- Two natural transformations transporting the $\odot$ and U functors in the expected way.

Having presented the double-categorical framework, we now move on to framed bicategories. The key intuition here comes from our earlier observation: morphisms between indices, *i.e.* morphisms in $\mathbb{D}_0$, induce polynomials, *i.e.* objects in $\mathbb{D}_1$. Categorically, this translates into a bifibrational structure on the (L, R) functor:

Definition 9 (Framed bicategory [Shulman, 2009])**.** A framed bicategory is a double category for which the functor

$$(L, R) : \mathbb{D}_1 \to \mathbb{D}_0 \times \mathbb{D}_0$$

is a bifibration.

Example 9 (Framed bicategory $Poly_{\mathcal{E}}$ [Gambino and Kock, 2010, §3.7])**.** The bifibration is therefore the *endpoints* functor [Gambino and Kock, 2010, §3.10] for which the cobase change of polynomial $F : I \xleftarrow{s} B \xrightarrow{f} A \xrightarrow{t} J$ along (u, v), denoted $(u, v)_! F$, is

$$\begin{tikzcd}
I \arrow[d, "u"'] & B \arrow[l, "s"'] \arrow[r, "f"] \arrow[d, equal] & A \arrow[r, "t"] \arrow[d, equal] & J \arrow[d, "v"] \\
K & B \arrow[l, "u \circ s"] \arrow[r, "f"'] & A \arrow[r, "v \circ t"'] & L
\end{tikzcd}$$

While the base change of G along (u, v), denoted $(u, v)^* G$ is defined as:

$$\begin{tikzcd}
I \arrow[d, equal] & & \cdot \arrow[ll] \arrow[r] \arrow[d, "v^{\dagger\dagger}"] \arrow[dr, phantom, "\lrcorner", very near start] & \cdot \arrow[r] \arrow[d, "v^{\dagger}"] \arrow[dr, phantom, "\lrcorner", very near start] & J \arrow[d, "v"] \\
\cdot \arrow[d, equal] & & \cdot \arrow[ll] \arrow[r] \arrow[dl, "\epsilon"] \arrow[dd, "\Delta_f \Pi_f u^{\dagger}"] \arrow[ddr, phantom, "\lrcorner", very near start] & \cdot \arrow[r] \arrow[dd, "\Pi_f u^{\dagger}"] & \cdot \arrow[dd, equal] \\
\cdot \arrow[d, "u"'] & \cdot \arrow[l, "s^{\dagger}"'] \arrow[dr, "u^{\dagger}"] \arrow[dl, phantom, "\llcorner", very near start] & & & \\
K & & D \arrow[ll, "s"] \arrow[r, "f"'] & C \arrow[r, "t"'] & L
\end{tikzcd}$$

As before, these definitions straightforwardly translates to operations on containers.

Example 10 (Framed bicategory *PolyFun*$_{\mathcal{E}}$ [Gambino and Kock, 2010, §3.6])**.** The fibrational structure of the framed bicategory gives rise to a *transporter lift* (the cartesian lifting of the fibration (L, R)) and a *cotransporter lift* (the op-cartesian lifting of the opfibration (L, R)).

The transporter lift of (u, v) to P_G is given by:

$$\begin{tikzcd}
\mathcal{E}_{/I} \arrow[r, "\Sigma_u"] \arrow[d, "\Sigma_u"'] \arrow[dr, phantom, "\cong"] & \mathcal{E}_{/K} \arrow[r, "P_G"] \arrow[d, equal] \arrow[dr, phantom, "\cong"] & \mathcal{E}_{/L} \arrow[r, "\Delta_v"] \arrow[d, equal] \arrow[dr, phantom, "\Downarrow \eta"] & \mathcal{E}_{/K} \arrow[d, "\Sigma_v"] \\
\mathcal{E}_{/K} \arrow[r, equal] & \mathcal{E}_{/K} \arrow[r, "P_G"'] & \mathcal{E}_{/L} \arrow[r, equal] & \mathcal{E}_{/L}
\end{tikzcd}$$

The cotransporter lift of (u, v) to P_F is given by:

$$\begin{tikzcd}
\mathcal{E}_{/I} \arrow[r, equal] \arrow[d, "\Sigma_u"'] \arrow[dr, phantom, "\Downarrow \epsilon"] & \mathcal{E}_{/I} \arrow[r, "P_F"] \arrow[d, equal] \arrow[dr, phantom, "\cong"] & \mathcal{E}_{/J} \arrow[r, equal] \arrow[d, equal] \arrow[dr, phantom, "\cong"] & \mathcal{E}_{/J} \arrow[d, "\Sigma_v"] \\
\mathcal{E}_{/K} \arrow[r, "\Delta_u"'] & \mathcal{E}_{/I} \arrow[r, "P_F"'] & \mathcal{E}_{/J} \arrow[r, "\Sigma_v"'] & \mathcal{E}_{/L}
\end{tikzcd}$$

Following Gambino and Kock [2010], we define the base change of P_G along (u, v) by $(u, v)^* P_G = \Delta_v \circ P_G \circ \Sigma_u$. Dually, we define the cobase change of P_F along (u, v) by $(u, v)_! P_F = \Sigma_v \circ P_F \circ \Delta_u$.

At this stage, it should be clear that the interpretation of polynomials is more than a mere functor from *Poly*$_{\mathcal{E}}$ to *PolyFun*$_{\mathcal{E}}$: loosely speaking, it establishes an equivalence of categories. Equivalence of framed bicategory is formally defined as follow:

Definition 10 (Framed biequivalence [Shulman, 2009, §7.1])**.** A framed equivalence between the framed bicategory $\mathbb{C}$ and $\mathbb{D}$ consists of:

- Two lax double functors $F : \mathbb{C} \to \mathbb{D}$ and $G : \mathbb{D} \to \mathbb{C}$, and
- Two framed natural isomorphism $\eta : 1 \cong G \circ F$ and $\epsilon : F \circ G \cong 1$

We then recall this result of Gambino and Kock [2010]:

Theorem 1 (Theorem 3.8 [Gambino and Kock, 2010])**.** *Squares of PolyFun*$_{\mathcal{E}}$ *are uniquely represented (up to a choice of pullback) by a square of Poly*$_{\mathcal{E}}$*. Consequently, the interpretation of polynomials is a framed biequivalence.*

The interpretation functor is an equivalence of framed bicategory between *Poly*$_{\mathcal{E}}$ and the framed bicategory *PolyFun*$_{\mathcal{E}}$. We thus conflate the category of polynomials *Poly*$_{\mathcal{E}}$ and the category of polynomial functors $\mathit{PolyFun}_{\mathcal{E}}$. Polynomials are a “small” presentation of the larger functorial objects. Since both categories are equivalent, we do not lose expressive power by working in the small language.

Earlier, we have isolated a class of cartesian morphisms. This defines a sub-category of polynomials, which will be of prime interest in this paper. For clarity, we expound its definition:

Example 11 (Framed bicategory $\mathit{Poly}^c_{\mathcal{E}}$ [Gambino and Kock, 2010, §3.13])**.** The framed bicategory $\mathit{Poly}^c_{\mathcal{E}}$ is defined by:

- Objects: indices, *i.e.* objects of $\mathcal{E}$
- Vertical arrows: index morphisms, *i.e.* morphisms of $\mathcal{E}$
- Horizontal arrows: polynomial indexed by I and J, respectively left and right frames
- Squares: cartesian morphism of polynomial reindexed by u and v, respectively left and right frames:

$$\begin{tikzcd}
I \arrow[d, "s"'] & B \arrow[l] \arrow[r] \arrow[d] \arrow[dr, phantom, very near start, "\lrcorner"] & A \arrow[r] \arrow[d, "\alpha"] & J \arrow[d, "t"] \\
K & D \arrow[l] \arrow[r] & C \arrow[r] & L
\end{tikzcd}$$

That is, we take $\mathbb{D}_0 \triangleq \mathcal{E}$ and $\mathbb{D}_1 \triangleq \biguplus_{I,J} \mathit{Poly}^c_{\mathcal{E}}(I, J)$ for which we define:

- The identity functor U that maps an index to the identity polynomial at that index ;
- The left frame L that projects the source index I ;
- The right frame R that projects the target index J ;
- The composition $\odot$ of polynomial functors.

The *frames* defined by L and R thus correspond to, respectively, the left-hand side and right-hand side of polynomials and polynomial morphisms. As for the bifibration structure, consider a pair of morphism $u, v : K \to I, L \to J$ in the base category $\mathcal{E} \times \mathcal{E}$, we have:

- A cobase-change functor reindexing a polynomial $P : \mathit{Poly}^c_{\mathcal{E}}(I, J)$ to a polynomial $(u, v)_! P : \mathit{Poly}^c_{\mathcal{E}}(K, L)$;
- A base-change functor reindexing a polynomial $P : \mathit{Poly}^c_{\mathcal{E}}(K, L)$ to a polynomial $(u, v)^* P : \mathit{Poly}^c_{\mathcal{E}}(I, J)$.

This extra-structure lets us transport polynomials across frames: given a polynomial, we can reindex or op-reindex it to any frame along a pair of index morphisms.

The interpretation functor is again an equivalence of framed bicategory between $\mathit{Poly}^{c}_{\mathcal{E}}$ and the framed bicategory $\mathit{PolyFun}^{c}_{\mathcal{E}}$ which morphisms of functors consists of cartesian natural transformations. We therefore have the following result:

Theorem 2 (Theorem 3.13 [Gambino and Kock, 2010])**.** *The subcategory $\mathit{PolyFun}^{c}_{\mathcal{E}}$ is framed biequivalent to $\mathit{Poly}^{c}_{\mathcal{E}}$. In particular, squares of $\mathit{PolyFun}^{c}_{\mathcal{E}}$ are uniquely represented by their diagrammatic counterpart in $\mathit{Poly}^{c}_{\mathcal{E}}$.*

The interpretation functor is an equivalence of framed bicategory between $\mathit{Poly}^{c}_{\mathcal{E}}$ and the framed bicategory $\mathit{PolyFun}^{c}_{\mathcal{E}}$ [Gambino and Kock, 2010, Theorem 3.13]. We can therefore conflate, once and for all, the category of polynomials $\mathit{Poly}^{c}_{\mathcal{E}}$ and the category of polynomial functors $\mathit{PolyFun}^{c}_{\mathcal{E}}$. In a sense, polynomials are a "small" presentation of the larger functorial objects. However, we do not lose expressive power by working in the small language, since both categories are equivalent.

2 Inductive Families in Type Theory

In this section, we set out to establish a formal connection between a presentation of inductive families in type theory and the categorical model of polynomial functors. On the type theoretical side, we adopt the universe-based presentation introduced by Chapman et al. [2010]. Working on a universe gives us a syntactic internalisation of inductive families within type theory. Hence, we can manipulate and reason about inductive families from within the type theory itself.

The original motivation for this design is *generic programming*: the programmer can compute over the structure of datatypes, or even compute new datatypes from old. In mathematical term, "generic programming" reads as *reflection*: we can reflect the meta-theory of inductive types within the type theory. For systems like Agda or Coq, we can imagine reducing their syntactic definition to such a universe by *elaboration* [Dagand and McBride, 2013].

We recall the definition of the universe in Figure 3. A Desc code is a syntactic object *desc*ribing a functor from Set^{I} to Set. To obtain this functor, we have to interpret the code using $[\![\_]\!]$. The reader will gain intuition for the codes by looking at their interpretation, *i.e.* their semantics. To describe functors from Set^{I} to Set^{J}, we use the isomorphism $[\mathrm{Set}^{I}, \mathrm{Set}]^{J} \cong [\mathrm{Set}^{I}, \mathrm{Set}^{J}]$. Hence, in idesc, we pull the J-index to the front and thus capture functors on slices of Set. The interpretation $[\![\_]\!]$ extends pointwise to idesc. Inhabitants of the idesc type are called *descriptions*. By construction, the interpretation of a description is a strictly positive functor: for a description D, the initial algebra always exists and is denoted $(\mu D, \mathit{in} : [\![D]\!]\ \mu D \to \mu D)$.

Definition 11 (Described functor)**.** A functor is *described* if it is isomorphic to the interpretation of a description.

$$\begin{array}{l}
\textbf{data}\ \mathsf{Desc}\ [I : \textsc{Set}] : \textsc{Set}^1\ \textbf{where} \\
\quad \mathsf{Desc}\ I \ni \text{'}\mathsf{var}\ (i : I) \\
\qquad\qquad \mid\ \text{'}1 \\
\qquad\qquad \mid\ \text{'}\Pi\ (S : \textsc{Set})\ (T : S \to \mathsf{Desc}\ I) \\
\qquad\qquad \mid\ \text{'}\Sigma\ (S : \textsc{Set})\ (T : S \to \mathsf{Desc}\ I)
\end{array}
\qquad
\begin{array}{llcl}
[\![(D : \mathsf{Desc}\ I)]\!] & (X : I \to \textsc{Set}) & : & \textsc{Set} \\
[\![\text{'}\mathsf{var}\ i]\!] & X & \mapsto & X\ i \\
[\![\text{'}1]\!] & X & \mapsto & \mathbb{1} \\
[\![\text{'}\Pi\ S\ T]\!] & X & \mapsto & (s : S) \to [\![T\ s]\!]\ X \\
[\![\text{'}\Sigma\ S\ T]\!] & X & \mapsto & (s : S) \times [\![T\ s]\!]\ X
\end{array}$$

$$\begin{array}{lllcl}
\mathsf{idesc} & (I : \textsc{Set}) & (J : \textsc{Set}) & : & \textsc{Set}^1 \\
\mathsf{idesc} & I & J & \mapsto & J \to \mathsf{Desc}\ I
\end{array}
\qquad
\begin{array}{llcl}
[\![(D : \mathsf{idesc}\ I\ J)]\!] & (X : I \to \textsc{Set}) & : & J \to \textsc{Set} \\
[\![D]\!] & X & \mapsto & \lambda j.\ [\![D\ j]\!]\ X
\end{array}$$

Figure 3: Universe of inductive families

Example 12 (Natural numbers)**.** The signature functor of natural numbers is described by:

$$\begin{array}{lcl}
\mathsf{NatD} & : & \mathsf{idesc}\ \mathbb{1}\ \mathbb{1} \\
\mathsf{NatD} & \mapsto & \lambda *.\ \text{'}\Sigma\ (\mathbb{1} + \mathbb{1})\ \lambda \begin{cases} \mathsf{inj}_l * \mapsto \text{'}1 \\ \mathsf{inj}_r * \mapsto \text{'}\mathsf{var}\ * \end{cases}
\end{array}$$

The reader will check that the interpretation $[\![\_]\!]$ of this code gives a functor isomorphic to the expected $X \mapsto 1 + X$.

2.1 Descriptions are equivalent to polynomials

We can now prove the equivalence between described functors and polynomial functors. The first step is to prove that described functors are polynomial:

Lemma 3. *The class of described functors is included in the class of polynomial functors.*

Proof. Let $F : \textsc{Set}_{/I} \to \textsc{Set}_{/J}$ a described functor.

By definition of the class of described functor, F is naturally isomorphic to the interpretation of a description. That is, for any $j : J$, there is a $D : \mathsf{Desc}\ I$ such that:

$$F\ j \cong [\![D]\!]$$

By induction over D, we show that $[\![D]\!]$ is naturally isomorphic to a polynomial:

Case $D = \text{'}1$**:** We have $[\![\text{'}1]\!]\ X \cong \mathbb{1} \times X^{\mathbb{0}}$, which is clearly polynomial

Case $D = \text{'}\mathsf{var}\ i$**:** We have $[\![\text{'}\mathsf{var}\ i]\!]\ X \cong \mathbb{1} \times (X\ i)^{\mathbb{1}}$, which is clearly polynomial

Case $D = \text{'}\Sigma\ S\ T$**:** We have $[\![\text{'}\Sigma\ S\ T]\!]\ X = (s : S) \times [\![T\ s]\!]\ X$. By induction hypothesis, $[\![T\ s]\!]\ X \cong (x : S_{T\ s}) \times (p : P_{T\ s}\ x) \to X\ (n_{T\ s}\ p)$. Therefore, we obtain that:

$$\begin{aligned}
[\![\text{'}\Sigma\ S\ T]\!]\ X &\cong (s : S) \times (x : S_{T\ s}) \times (p : P_{T\ s}) \to X\ (n_{T\ s}\ p) \\
&\cong (sx : (s : S) \times S_{T\ s}) \times (p : P_{T\ (\pi_0\ sx)} (\pi_1\ sx)) \to X\ (n_{T\ (\pi_0\ sx)}\ p)
\end{aligned}$$

This last functor being clearly polynomial.

Case $D = \text{'}\Pi\, S\, T$**:** We have $[\![\text{'}\Pi\, S\, T]\!]\ X = (s : S) \to [\![T\ s]\!]\ X$. By induction hypothesis, $[\![T\ s]\!]\ X \cong (x : S_{T\ s}) \times (p : P_{T\ s}\ x) \to X\ (n_{T\ s}\ p)$. Therefore, we obtain that:

$$\begin{aligned}
[\![\text{'}\Pi\, S\, T]\!]\ X &\cong (s : S) \to (x : S_{T\ s}) \times (p : P_{T\ s}\ x) \to X\ (n_{T\ s}\ p) \\
&\cong (f : (s : S) \to S_{T\ s}) \times (s : S)(p : P_{T\ s}\ (f\ s)) \to X\ (n_{T\ s}\ p) \\
&\cong (f : (s : S) \to S_{T\ s}) \times (sp : (s : S) \times P_{T\ s}\ (f\ s)) \to X\ (n_{T\ (\pi_0\ sp)}\ (\pi_1\ sp))
\end{aligned}$$

This last functor being clearly polynomial.

□

To prove the other inclusion – that polynomials functors on Set are a subset of described functors – we rely on Lemma 2. To this end, we must prove some algebraic properties of the class of described functors, namely that they are closed under reindexing, its adjoints, and composition. To do so, the methodology is simply to code these operations in idesc.

Lemma 4. *Described functors are closed under reindexing and its adjoints.*

Proof. We describe the pullback functors and their adjoints by:

$$\begin{array}{lcl}
D\Delta_{(f : A \to B)} & : & \mathsf{idesc}\ B\ A \\
D\Delta_f & \mapsto & \lambda a.\ \text{'}\mathsf{var}\ (f\ a)
\end{array}$$

$$\begin{array}{lcl}
D\Sigma_{(f : A \to B)} & : & \mathsf{idesc}\ A\ B \\
D\Sigma_f & \mapsto & \lambda b.\ \text{'}\Sigma\ f^{-1}\ b\ \lambda a.\ \text{'}\mathsf{var}\ a
\end{array}
\qquad
\begin{array}{lcl}
D\Pi_{(f : A \to B)} & : & \mathsf{idesc}\ A\ B \\
D\Pi_f & \mapsto & \lambda b.\ \text{'}\Pi\ f^{-1}\ b\ \lambda a.\ \text{'}\mathsf{var}\ a
\end{array}$$

Where the inverse of a function f is represented by the following inductive type:

$$\begin{array}{l}
\textbf{data}\ [f : A \to B]^{-1}\ (b : B) : \text{Set}\ \textbf{where} \\
\quad f^{-1}\ (b = f\ a) \ni \mathsf{inv}\ (a : A)
\end{array}$$

It is straightforward to check that these descriptions interpret to the expected operation on slices of Set, *i.e.* that we have:

$$[\![D\Delta_f]\!] \cong \Delta_f \qquad [\![D\Sigma_f]\!] \cong \Sigma_f \qquad [\![D\Pi_f]\!] \cong \Pi_f$$

□

Lemma 5. *Described functors are closed under composition.*

Proof. We define composition of descriptions by:

$$\begin{array}{lllcl}
(D : \mathsf{idesc}\ B\ C) & \circ_D & (E : \mathsf{idesc}\ A\ B) & : & \mathsf{idesc}\ A\ C \\
D & \circ_D & E & \mapsto & \lambda c.\ \mathsf{compose}\ (D\ c)\ E
\end{array}$$

$$\begin{array}{l}
\textbf{where} \\
\quad \begin{array}{llllcl}
\mathsf{compose} & (D : \mathsf{Desc}\ B) & (E : \mathsf{idesc}\ B\ A) & : & \mathsf{Desc}\ A \\
\mathsf{compose} & (\text{'}\mathsf{var}\ b) & E & \mapsto & E\ b \\
\mathsf{compose} & \text{'}1 & E & \mapsto & \text{'}1 \\
\mathsf{compose} & (\text{'}\Pi\ S\ T) & E & \mapsto & \text{'}\Pi\ S\ \lambda s.\ \mathsf{compose}\ (T\ s)\ E \\
\mathsf{compose} & (\text{'}\Sigma\ S\ T) & E & \mapsto & \text{'}\Sigma\ S\ \lambda s.\ \mathsf{compose}\ (T\ s)\ E
\end{array}
\end{array}$$

It is then straightforward to check that this is indeed computing the composition of the functors, *i.e.* that we have:

$$[\![D \circ_D E]\!] \cong [\![D]\!] \circ [\![E]\!]$$

□

Lemma 6. *The class of polynomial functors is included in the class of described functors.*

Proof. Described functors are closed under reindexing, its left and right adjoint (Lemma 4), are closed under composition (Lemma 5) and are defined up to natural isomorphism. By Lemma 2, the class of polynomial functors is the least such set. Therefore, the class of polynomial functor is included in the class of described functors.

□

We conclude with the desired equivalence:

Proposition 1. *The class of described functors corresponds exactly to the class of polynomial functors.*

Proof. By Lemma (3) and Lemma (6), we have both inclusions.

□

The benefit of this algebraic approach is its flexibility with respect to the universe definition: for practical purposes, we are likely to introduce new Desc codes. However, the implementation of reindexing and its adjoints will remain unchanged. Only composition would need to be verified. Besides, these operations are useful in practice, so we are bound to implement them anyway. In the rest of this paper, we shall conflate descriptions, polynomials, and polynomial functors, silently switching from one to another as we see fit.

2.2 An alternative proof

An alternative approach, followed by Morris [2007] for example, consists in reducing these codes to containers. We thus obtain the equivalence to polynomial functors, relying on the fact that containers are an incarnation of polynomial functors in the internal language Gambino and Kock [2010, §2.18]. This less algebraic approach is more constructive. However, to be absolutely formal, it calls for proving some rather painful (extensional) equalities. If the proofs are laborious, the translation itself is not devoid of interest. In particular, it gives an intuition of descriptions in terms of shape, position and indices. This slightly more abstract understanding of our universe will be useful in this paper, and is useful in general when reasoning about datatypes.

We formalise the translation in Figure 4, mapping descriptions to containers. The message to take away from that translation is which code contributes to which part of the container, *i.e.* shape, position, and/or index. Crucially, the '1 and 'Σ codes contribute only to the shapes. The 'var and 'Π codes, on the other hand, contribute to the positions. Finally, the 'var code is singly defining the next index.

$$
\begin{array}{l}
\langle(D : \mathsf{idesc}\ I\ J)\rangle\ :\ \mathit{ICont}\ I\ J \\
\langle D\rangle \mapsto \lambda j.\,\mathsf{Shape}\,(D\ j) \lhd^{\lambda j.\,\mathsf{Index}\,(D\ j)} \lambda j.\,\mathsf{Pos}\,(D\ j) \qquad \textbf{where} \\
\quad \begin{array}{llll}
\mathsf{Shape}\,(D : \mathsf{Desc}\,I) & & : & \mathrm{SET} \\
\mathsf{Shape} & \text{'var}\ i & \mapsto & \mathbb{1} \\
\mathsf{Shape} & \text{'1} & \mapsto & \mathbb{1} \\
\mathsf{Shape} & \text{'}\Pi\, S\ T & \mapsto & (s : S) \to \mathsf{Shape}\,(T\ s) \\
\mathsf{Shape} & \text{'}\Sigma\, S\ T & \mapsto & (s : S) \times \mathsf{Shape}\,(T\ s)
\end{array} \\
\\
\quad \begin{array}{lllll}
\mathsf{Pos}\,(D : \mathsf{Desc}\,I)\,(sh : \mathsf{Shape}\ D) & & & : & \mathrm{SET} \\
\mathsf{Pos} & \text{'var}\ i & * & \mapsto & \mathbb{1} \\
\mathsf{Pos} & \text{'1} & * & \mapsto & \mathbb{0} \\
\mathsf{Pos} & \text{'}\Pi\, S\ T & f & \mapsto & (s : S) \times \mathsf{Pos}\,(T\ s)\,(f\ s) \\
\mathsf{Pos} & \text{'}\Sigma\, S\ T & (s, t) & \mapsto & \mathsf{Pos}\,(T\ s)\ t
\end{array} \\
\\
\quad \begin{array}{lllll}
\mathsf{Index}\,(D : \mathsf{Desc}\,I)\,(pos : \mathsf{Pos}\ D\ sh) & & & : & I \\
\mathsf{Index} & \text{'var}\ i & * & \mapsto & i \\
\mathsf{Index} & \text{'}\Pi\, S\ T & (s, pos) & \mapsto & \mathsf{Index}\,(T\ s)\ pos \\
\mathsf{Index} & \text{'}\Sigma\, S\ T & pos & \mapsto & \mathsf{Index}\,(T\,(\pi_0\ sh))\ pos
\end{array}
\end{array}
$$

Figure 4: From descriptions to containers

The inverse translation is otherwise trivial and given here for the sake of completeness:

$$
\begin{array}{lll}
\langle(C : \mathit{ICont}\ I\ J)\rangle^{-1} & : & \mathsf{idesc}\ I\ J \\
\langle S \lhd^n P\rangle^{-1} & \mapsto & \text{'}\Sigma\ S\ \lambda sh.\ \text{'}\Pi\,(P\ sh)\ \lambda pos.\ \text{'var}\,(n\ pos)
\end{array}
$$

We are left to prove that these translations are indeed inverse of each other: while this proof is extremely tedious to carry formally, it should be intuitively straightforward. We therefore assume the following lemma:

Lemma 7 (Described functors to polynomials, alternatively)**.** $\langle \_ \rangle$ *is essentially surjective.*

2.3 Discussion

Let us reflect on the results obtained in this section. By establishing an equivalence between descriptions – a programming artefact – and polynomial functors – a mathematical object – we connect software to mathematics, and conversely. On the one hand, descriptions are suitable for practical purposes: they are a syntactic object, fairly intensional, and can therefore be conveniently manipulated by a computer. Polynomial functors, on the other hand, are fit for theoretical work: they admit a diagrammatic representation and are defined extensionally, up to natural isomorphism.

Better still, we have introduced containers as a middle ground between these two presentations. Containers are an incarnation of polynomials in the internal language. Reasoning extensionally about them is equivalent to reasoning about polynomials. Nonetheless, they are also rather effective type theoretic procedures: we can implement them in Agda[1].

[1]Such an implementation is available on the the first author's website.

We shall traverse this bridge between software and mathematics in both directions. Going from software to mathematics, we hope to gain a deeper understanding of our constructions. Case in point is generic programming in type theory: we develop many constructions over datatypes, such as ornaments, but the justification for these is often extremely operational, one might even say "ad-hoc". By putting our polynomial glasses on, we can finally see through the syntax and understand the structure behind these definitions. Conversely, going from mathematics to software, we translate mathematical structures to new software constructions. The theory of polynomial functors is indeed well developed. Most programming examples presented in this paper – such as derivatives or ornaments – were first presented in the polynomial functor literature. Besides, by exploring the structure of polynomial functors, we discover new and interesting programming idioms – such as the pullback and composition of ornaments.

The categorically minded reader might be tempted to look for an equivalence of category. However, we have not yet introduced any notion of morphism between descriptions. What we have established is a lowly "set theoretic" equivalence between the class of descriptions and the class of polynomial functors. In terms of equivalence of categories, we have established that the object part of a functor, yet to be determined, maps descriptions to polynomial functors in an essentially surjective way. We shall complete this construction in the following section. We will set up descriptions in a double category with ornaments as morphisms. The translation $\langle \_ \rangle$ will then functorially map it to the double category $\mathit{PolyFun}^{c}_{\mathcal{E}}$.

3 A Categorical Treatment of Ornaments

The motivation for ornaments comes from the frequent need, when using dependent types, to relate datatypes that share the same structure. In this setting, ornaments play the role of an organisation principle. Intuitively, an ornament is the combination of two datatype transformations: we may *extend* the constructors, and/or *refine* the indices. Ornaments preserve the underlying data-structure by enforcing that an extension respects the arity of the original constructors. By extending a datatype, we introduce more information, thus enriching its logical content. A typical example of such an ornament is the one taking natural numbers to lists:

$$\begin{array}{l} \textbf{data}\ \mathsf{Nat} : \textsc{Set}\ \textbf{where} \\ \quad \mathsf{Nat}\ \ni \mathsf{0} \\ \qquad\quad \mid \mathsf{suc}\,(n : \mathsf{Nat}) \end{array} \quad \overset{\mathsf{List\text{-}Orn}\ A}{\Rightarrow} \quad \begin{array}{l} \textbf{data}\ \mathsf{List}\ [A : \textsc{Set}] : \textsc{Set}\ \textbf{where} \\ \quad \mathsf{List}\ A\ \ni \mathsf{nil} \\ \qquad\qquad \mid \mathsf{cons}\,(a : A)(as : \mathsf{List}\ A) \end{array}$$

By refining the indices of a datatype, we make it logically more discriminating. For example, we can ornament natural numbers to finite sets:

$$\begin{array}{l} \textbf{data}\ \mathsf{Nat} : \textsc{Set}\ \textbf{where} \\ \quad \mathsf{Nat}\ \ni \mathsf{0} \\ \qquad\quad \mid \mathsf{suc}\,(n : \mathsf{Nat}) \end{array} \quad \overset{\mathsf{Fin\text{-}Orn}}{\Rightarrow} \quad \begin{array}{l} \textbf{data}\ \mathsf{Fin}\ (n : \mathsf{Nat}) : \textsc{Set}\ \textbf{where} \\ \quad \mathsf{Fin}\ (n = \mathsf{suc}\, n')\ \ni \mathsf{f0}\ (n' : \mathsf{Nat}) \\ \qquad\qquad\qquad\quad \mid \mathsf{fsuc}\,(n' : \mathsf{Nat})(k : \mathsf{Fin}\, n') \end{array}$$

$$
\begin{array}{llll}
\multicolumn{4}{l}{\textbf{data}\ \mathsf{Orn}\ (D : \mathsf{Desc}\ K)[u : I \to K] : \textsc{Set}^1\ \textbf{where}} \\
\multicolumn{4}{l}{\quad -\ \textit{Extend with } S\textit{:}} \\
\mathsf{Orn}\ \ D\ \ u & \ni & \mathsf{insert}\ (S : \textsc{Set})(D^+ : S \to \mathsf{Orn}\ D\ u) \\
\multicolumn{4}{l}{\quad -\ \textit{Refine index:}} \\
\mathsf{Orn}\ \ (\text{'}\mathsf{var}\ k)\ \ u & \ni & \text{'}\mathsf{var}\ (i : u^{-1}\ k) \\
\multicolumn{4}{l}{\quad -\ \textit{Copy the original:}} \\
\mathsf{Orn}\ \ \text{'}1\ \ u & \ni & \text{'}1 \\
\mathsf{Orn}\ (\text{'}\Pi\ S\ T)\ u & \ni & \text{'}\Pi\ (T^+ : (s : S) \to \mathsf{Orn}\ (T\ s)\ u) \\
\mathsf{Orn}\ (\text{'}\Sigma\ S\ T)\ u & \ni & \text{'}\Sigma\ (T^+ : (s : S) \to \mathsf{Orn}\ (T\ s)\ u) \\
\multicolumn{4}{l}{\quad -\ \textit{Delete } \text{'}\Sigma\ S\textit{:}} \\
 & \mid & \mathsf{delete}\ (s : S)(T^+ : \mathsf{Orn}\ (T\ s)\ u)
\end{array}
$$

(a) Code

$$
\begin{array}{lcl}
[\![(O : \mathsf{Orn}\ D\ u)]\!]_{\mathsf{orn}} & : & \mathsf{Desc}\ I \\
[\![\mathsf{insert}\ S\ D^+]\!]_{\mathsf{orn}} & \mapsto & \text{'}\Sigma\ S\ \lambda s.\ [\![D^+\ s]\!]_{\mathsf{orn}} \\
[\![\text{'}\mathsf{var}\ (\mathsf{inv}\ i)]\!]_{\mathsf{orn}} & \mapsto & \text{'}\mathsf{var}\ i \\
[\![\text{'}1]\!]_{\mathsf{orn}} & \mapsto & \text{'}1 \\
[\![\text{'}\Pi\ T^+]\!]_{\mathsf{orn}} & \mapsto & \text{'}\Pi\ S\ \lambda s.\ [\![T^+\ s]\!]_{\mathsf{orn}} \\
[\![\text{'}\Sigma\ T^+]\!]_{\mathsf{orn}} & \mapsto & \text{'}\Sigma\ S\ \lambda s.\ [\![T^+\ s]\!]_{\mathsf{orn}} \\
[\![\mathsf{delete}\ s\ T^+]\!]_{\mathsf{orn}} & \mapsto & [\![T^+\ s]\!]_{\mathsf{orn}}
\end{array}
$$

(b) Interpretation

Figure 5: Universe of ornaments

3.1 Ornaments

We recall the definition of the universe of ornaments in Figure 5. Besides our ability to copy the original description (with the codes '1, 'Σ, and 'Π), we can insert new Σ-types, delete Σ-types by providing a witness, and use a more precise index in the 'var codes. While this universe is defined on $\mathsf{Desc}\ K$, *i.e.* functors from $\textsc{Set}_{/K}$ to $\textsc{Set}$, it readily lifts to endofunctors on slices, *i.e.* on descriptions $\mathsf{idesc}\ K\ L$:

$$
\begin{array}{lllll}
\mathsf{orn}\ (re_I : J \to I) & (re_O : P \to O) & (D : O \to \mathsf{Desc}\ I) & : & \textsc{Set}^1 \\
\mathsf{orn}\ \quad re_I & \quad re_O & \quad D & \mapsto & (p : P) \to \mathsf{Orn}\ re_I\ (D\ (re_O\ p))
\end{array}
$$

$$
\begin{array}{lcl}
[\![(o : \mathsf{orn}\ re_I\ re_O\ D)]\!]_{\mathsf{orn}}\ (p : P) & : & \mathsf{Desc}\ J \\
[\![o]\!]_{\mathsf{orn}} \qquad\qquad\qquad p & \mapsto & \mathsf{intOrn}\ (D\ (re_O\ p))\ (o\ p)
\end{array}
$$

Example 13 (Ornamenting natural numbers to list)**.** We obtain list from natural numbers with the following ornament:

$$
\begin{array}{lcl}
\mathsf{List\text{-}Orn}\ (A : \textsc{Set}) & : & \mathsf{orn}\ \mathsf{NatD}\ \mathsf{id}\ \mathsf{id} \\
\mathsf{List\text{-}Orn}\ \quad A & \mapsto & \lambda *.\ \text{'}\Sigma\ \lambda \begin{cases} \mathsf{inj}_l\ * \mapsto \text{'}1 \\ \mathsf{inj}_r\ * \mapsto \mathsf{insert}\ A\ \lambda\_.\ \text{'}\mathsf{var}\ * \end{cases}
\end{array}
$$

The reader will check that the interpretation ($[\![\_]\!]_{\mathsf{orn}}$) of this ornament followed by the interpretation ($[\![\_]\!]$) of the resulting description yields the signature functor of list $X \mapsto 1 + A \times X$.

Example 14 (Ornamenting natural numbers to finite sets)**.** We obtain finite sets by inserting a number $n' : \mathsf{Nat}$, constraining the index n to $\mathsf{suc}\ n'$, and – in the recursive case – indexing at n':

$$
\begin{array}{lcl}
\mathsf{Fin\text{-}Orn} & : & \mathsf{orn}\ \mathsf{NatD}\ (\lambda n.\ *)\ (\lambda n.\ *) \\
\mathsf{Fin\text{-}Orn} & \mapsto & \lambda n.\ \mathsf{insert}\ \mathsf{Nat}\ \lambda n'.\ \mathsf{insert}\ (n = \mathsf{suc}\ n')\ \lambda\_. \\
 & & \qquad \text{'}\Sigma\ \lambda \begin{cases} \mathsf{inj}_l\ * \mapsto \text{'}1 \\ \mathsf{inj}_r\ * \mapsto \text{'}\mathsf{var}\ n' \end{cases}
\end{array}
$$

Again, the reader will verify that this is indeed describing the signature of finite sets.

A detailed account of ornaments from a programmer's perspective will be found elsewhere [McBride, 2013, Dagand and McBride, 2012, Ko and Gibbons, 2011]. For the purpose of this paper, these definitions are enough. We shall refer to the aforementioned papers when programming concepts reappear in our categorical framework.

3.2 Ornaments are cartesian morphisms

Relating the definition of ornaments with our polynomial reading of descriptions, we make the following remarks. Firstly, the ornament code lets us only insert – with the insert code – or delete – with the delete code – 'Σ codes while forbidding deletion or insertion of either 'Π or 'var codes. In terms of container, this translates to: shapes can be extended, while positions must be isomorphic. Secondly, on the 'var code, the ornament code lets us pick any index in the inverse image of u. In terms of container, this corresponds to the coherence condition: the initial indexing must commute with applying the ornamented indexing followed by u. Concretely, for a container $S \lhd^n P$, an ornament can be modelled as an extension *ext*, a refined indexing n^+ subject to coherence condition q with respect to the original indexing:

$$\begin{cases} ext : S\ (v\ l) \to \textsc{Set} \\ n^+ : ext\ sh \to P\ sh \to K \\ q : \forall pos : P\ sh.\ u\ (n^+\ e\ pos) = n\ pos \end{cases}$$

Equivalently, the family of set *ext* can be understood as the inverse image of a function $\sigma : S^+\ l \to S\ (v\ l)$. The function n^+ is then the next index function of a container with shapes S^+ and positions $P \circ \sigma$. Put otherwise, the morphism on shapes σ together with the coherence condition q form a cartesian morphism from $S^+ \lhd^{n^+} P \circ \sigma$ to $S \lhd^n P$! To gain some intuition, the reader can revisit the cartesian morphism of Example 5 as an ornament of container – by simply inverting the morphism on shapes – and as an ornament of description – by relating it with the ornament List-Orn (Example 13).

We shall now formalise this intuition by proving the following isomorphism:

Lemma 8. *Ornaments describe cartesian morphisms between polynomial functors,* i.e. *we have the isomorphism*

$$\mathsf{orn}\, D\ u\ v \cong Poly^c_{\mathcal{E}}(\_, D)_{u,v}$$

In terms of cartesian morphism of polynomials, extending the shape corresponds to the morphism α. Enforcing that the positions, *i.e.* the structure, of the datatype remain the same corresponds to the pullback along α. The refinement of indices corresponds to the frame morphisms commuting.

Proof. We develop the proof on the container presentation: this lets us work in type theory, where is anchored the definition of ornaments. It is a necessary hardship, since no other decent model of ornaments is available to us. After this bootstrapping process, we shall have the abstract tools necessary to lift off type theory.

The first half of the isomorphism consists of mapping an ornament o of a description D to a cartesian morphism from the container described by $[\![o]\!]_{\mathsf{orn}}$ to the container described by D. By definition of cartesian morphisms, we simply have to give a map from the shape of $[\![o]\!]_{\mathsf{orn}}$ to the shape of D:

$$
\begin{array}{lll}
\phi\,(o : \mathsf{orn}\,D\,u\,v) & : & \langle [\![o]\!]_{\mathsf{orn}} \rangle \overset{u}{\underset{v}{\Longrightarrow}}{}^{c} \langle D \rangle \\
\phi\quad o & \mapsto & (\lambda i.\,\mathsf{forget}\,o\,(u\,i)) \blacktriangleleft^{c}\ \ \textbf{where}
\end{array}
$$

$$
\begin{array}{llll}
\mathsf{forget}\,(O : \mathsf{Orn}\,D\,u)\,(sh : \mathsf{Shape}\,[\![O]\!]_{\mathsf{orn}}) & & : & \mathsf{Shape}\,D \\
\mathsf{forget}\quad \text{'}1 & * & \mapsto & * \\
\mathsf{forget}\quad (\text{'}\Pi\,T^{+}) & f & \mapsto & \lambda a.\,\mathsf{forget}\,(T^{+}\,a)\,(f\,a) \\
\mathsf{forget}\quad (\text{'}\Sigma\,T^{+}) & (a, sh) & \mapsto & (a, \mathsf{forget}\,(T^{+}\,a)\,sh) \\
\mathsf{forget}\ (\text{'}\mathsf{var}\,(\mathsf{inv}\,j)) & * & \mapsto & * \\
\mathsf{forget}\ (\mathsf{insert}\,a\,D^{+}) & (a, sh) & \mapsto & \mathsf{forget}\,(D^{+}\,a)\,sh \\
\mathsf{forget}\ (\mathsf{delete}\,s\,O) & sh & \mapsto & (s, \mathsf{forget}\,O\,sh)
\end{array}
$$

We are then left to check that (extensionally) the positions are constructed by pullback and the indexing is coherent. This is indeed the case, even though proving it in type theory is cumbersome. On positions, the ornament does not introduce or delete any new $\text{'}\Pi$ or $\text{'}\mathsf{var}$: hence the positions are left unchanged. On indices, we rely on $u^{-1}\,k$ to ensure that the more precise indexing is coherent by construction.

In the other direction, we are given a cartesian morphism from F to G. We return an ornament of the description of G. For the isomorphism to hold, this ornament must interpret to the description of F:

$$
\begin{array}{lll}
\psi\,(m : F \overset{u}{\underset{v}{\Longrightarrow}}{}^{c} G) & : & \mathsf{orn}\,\langle G \rangle^{-1}\,u\,v \\
\psi\quad (\mathsf{forget} \blacktriangleleft^{c}) & \mapsto & \lambda j.\,\text{'}\Sigma\,\lambda sh.\,\mathsf{insert}\,(\mathsf{forget}^{-1}\,sh)\,\lambda ext.\,\text{'}\Pi\,\lambda ps.\,\text{'}\mathsf{var}\,(\mathsf{inv}\,(n_F\,ps))
\end{array}
$$

Indeed, the description of G is a $\text{'}\Sigma$ of its shape, followed by a $\text{'}\Pi$ of its positions, terminated by a $\text{'}\mathsf{var}$ at the next index. To ornament G to F, we simply have to insert the inverse image of forget, *i.e.* the information that extends G to F. As for the next index, we can legitimately use F's indexing function: the coherence condition of the cartesian morphism ensures that it is indeed in the inverse image of the reindexing function.

Having carefully crafted the definition of ϕ and ψ, it should be obvious that these functions are inverse of each other. It is sadly not that obvious to an (intensional) theorem prover. Hence, we will not attempt to prove it in type theory here.

□

Relation with ornamental algebras [McBride, 2013, §4] To introduce the notion of "ornamental algebra", the second author implemented the erase helper function taking an ornamented type to its unornamented form. This actually corresponds to our transformation ϕ, followed by the interpretation of the resulting cartesian morphism. The erase function given in the original presentation is indeed natural and cartesian.

In the previous section, we have established a connection between descriptions and polynomials. We have now established a connection between ornaments and cartesian morphisms of polynomials. It thus makes sense to organise descriptions in a framed bicategory $IDesc^{c}$:

Definition 12 (Framed bicategory $IDesc^c$)**.** The framed bicategory $IDesc^c$ is defined by:

- Objects: sets
- Vertical morphisms: set morphisms
- Horizontal morphisms: descriptions, framed by I and J
- Squares: a square from F to G framed by u and v is an ornament $o : \mathsf{orn}\, G\; u\; v$ of G that interprets to (a code isomorphic to) F

Where, as for $Poly^c_{\mathcal{E}}$ (Example 11), the frame structure consists in reindexing a description along a pair of functions.

3.3 A framed biequivalence

We are now ready to establish an equivalence of category between $IDesc^c$ and $PolyFun^c_{\mathcal{E}}$, thus completing our journey from the type theoretical definition of ornaments to its model as cartesian morphisms.

Proposition 2. *The double category $IDesc^c$ is framed biequivalent to $PolyFun^c_{\mathcal{E}}$.*

Proof. As for the proof of Lemma 8, we work from $IDesc^c$ to $ICont$ to prove this theorem. Since $ICont$ is equivalent to $PolyFun^c_{\mathcal{E}}$, this gives the desired result. To prove a framed biequivalence, we need a functor on the base category and another on the total category. In this particular case, both base categories are SET: we shall therefore take the identity functor, hence trivialising the natural isomorphisms on composition, identity, and frames.

On the total category, we prove the equivalence by exhibiting a full and faithful functor from Desc to $ICont$ that is essentially surjective on objects. Unsurprisingly, this functor is defined on objects by $\langle \_ \rangle$, which is indeed essentially surjective by Lemma 7. The morphism part is defined by ϕ, which is full and faithful by Lemma 8. □

We have therefore established the following equivalences of framed bicategories:

$$\begin{tikzcd}
IDesc^c \arrow[rr, leftrightarrow, dashed] \arrow[dr, leftrightarrow] & & Poly^c_{\mathcal{E}} \arrow[r, leftrightarrow] & PolyFun^c_{\mathcal{E}} \\
& ICont \arrow[ur, leftrightarrow] & &
\end{tikzcd}$$

We may now conflate the notions of ornament, cartesian morphism, and cartesian natural transformation. In particular, we shall say that "F ornaments G" when we have a cartesian morphism from F to G. Let us now raid the polynomial toolbox for the purpose of programming with ornaments. The next section shows the beginning of what is possible.

4 Tapping into the categorical structure

In the previous section, we have characterised the notion of ornament in terms of cartesian morphism. We now turn to the original ornamental constructions [McBride, 2013] – such as the ornamental algebra and the algebraic ornament – and rephrase them in our categorical framework. Doing so, we extract the structure governing their type theoretic definition.

Next, we study the categorical structure of cartesian morphisms and uncover novel and interesting ornamental constructions. We shall see how the identity, composition, and frame reindexing translate into ornaments. We shall also be interested in pullbacks in the category $PolyFun^{c}_{\mathcal{E}}$ and the functoriality of the derivative in that category.

4.1 Ornamental algebra

Ornamenting a datatype is an effective recipe to augment it with new information. We thus expects that, given an ornamented object, we can *forget* its extra information and regain a raw object. This projection is actually a generic operation, provided by the *ornamental algebra*. It is a corollary of the very definition of ornaments as cartesian morphisms.

Corollary 1 (Ornamental algebra)**.** *From an ornament* $o : F \overset{u}{\underset{v}{\Longrightarrow}}{}^{c}\ G$*, we obtain the* ornamental algebra $o\text{-forgetAlg} : F\ (\mu G \circ v) \to \mu G \circ u$.

Proof. We apply the natural transformation o at μG and post-compose by in:

$$o\text{-forgetAlg} : F\ (\mu G \circ v) \xrightarrow{o_{\mu G}} (G\ \mu G) \circ u \xrightarrow{in} \mu G \circ u$$

□

Folding the ornamental algebra, we obtain a map from the ornamented type μF to its unornamented version μG. In effect, the ornamental algebra describes how to *forget* the extra-information introduced by the ornament.

Example 15 (Ornamental algebra of the List ornament)**.** The cartesian morphism from list to natural numbers (Example 5) maps the nil constructor to 0, while the cons constructor is mapped to suc. Post-composing by *in*, we obtain a natural number. This is the algebra computing the length of a list.

4.2 Algebraic ornaments

The notion of algebraic ornament was initially introduced by the second author [McBride, 2013]. A similar categorical construction, defined for any functor, was also presented by Atkey et al. [2012]. In this section, we reconcile these two works and show that, for a polynomial functor, the refinement functor can itself be internalised as a polynomial functor.

Definition 13 (Refinement functor [Atkey et al., 2012, §4.3])**.** Let F an endofunctor on $\mathcal{E}_{/I}$. Let $(X : \mathcal{E}_{/I}, \alpha : F\ X \to X)$ an algebra over F.

The *refinement functor* is defined by:

$$F^{\alpha} \triangleq \Sigma_{\alpha} \circ \hat{F} : (\mathcal{E}_{/I})_{/X} \to (\mathcal{E}_{/I})_{/X}$$

Where $\hat{F}$ – the lifting of F [Hermida and Jacobs, 1998, Fumex, 2012] – is taken, in an LCCC, to be the morphism part of the functor F.

The idea, drawn from refinement types [Freeman and Pfenning, 1991], is that a function $(\!|\alpha|\!) : \mu F \to X$ can be thought of as a predicate over μF. By *integrating* the algebra α *into* the signature F, we obtain a signature F^{α} indexed by X that describes the F-objects satisfying, by construction, the predicate $(\!|\alpha|\!)$. Categorically, this translates to:

Theorem 3 (Coherence property of algebraic ornament)**.** *The fixpoint of the algebraic ornament of P_F by α satisfies the isomorphism $\mu {P_F}^{\alpha} \cong \Sigma_{(\!|\alpha|\!)} \mathbf{1}\, \mu F$ where $\mathbf{1} : \mathcal{E}_{/I} \to [\mathcal{E}_{/I}, \mathcal{E}_{/I}]$, the terminal object functor, maps objects X to id_X.*

Proof. This is an application of Theorem 4.6 [Atkey et al., 2012], specialised to the codomain fibration (*i.e.* an LCCC). □

Informally, using a set theoretic notation, this isomorphisms reads as $\mu F^{\alpha}\ i\ x \cong \{t : \mu F\ i \mid (\!|\alpha|\!)\ t = x\}$. That is, the algebraic ornament μF^{α} at index i and x corresponds *exactly* to the pair of a witness t of $\mu F\ i$ and a proof that this witness satisfies the indexing equation $(\!|\alpha|\!)\ t = x$. In effect, from an algebraic predicate over an inductive type, we have an effective procedure reifying this predicate as an inductive family. This theorem also has an interesting computational interpretation. Crossing the isomorphism from left to right, we obtain the Recomputation theorem[McBride, 2013, §8]: from any $t^{+} : \mu F^{\alpha}\ i\ x$, we can extract a $t : \mu F\ i$ together with a proof that $(\!|\alpha|\!)\ t$ equals x. From right to left, we obtain the remember function [McBride, 2013, §7]: from any $t : \mu F\ i$, we can lift it to its ornamented form with remember $t : \mu F^{\alpha}\ i\ ((\!|\alpha|\!)\ t)$.

When F is a polynomial functor, we show that the refinement functor can be internalised and presented as an ornament of F. In practice, this means that from a description D and an algebra α, we can *compute* an ornament code that describes the functor D^{α}. This should not come as a surprise: algebraic ornaments were originally presented as ornamentations of the initial description [McBride, 2013, §5]. The following theorem abstracts the original definition:

Proposition 3. *Let F a polynomial endofunctor on $\mathcal{E}_{/I}$. Let (X, α) an algebra over P_F,* i.e. *$\alpha : P_F X \to X$. The refinement functor ${P_F}^{\alpha}$ is polynomial and ornaments F.*

Proof. To show that ${P_F}^{\alpha}$ is a polymonial ornamenting F, we exhibit a cartesian natural transformation from ${P_F}^{\alpha}$ to P_F. Since P_F is polynomial, we obtain that ${P_F}^{\alpha}$ is polynomial by Lemma 1.

First, there is a cartesian natural transformation from the lifting $\hat{P_F}$ to P_F. Indeed, for an LCCC, the lifting consists of the morphism part of P_F, denoted ${P_F}^{\to}$ [Fumex, 2012]. We therefore have the following isomorphism, hence cartesian natural transformation:

$$\begin{tikzcd}
(\mathcal{E}_{/I})_{/X} \arrow[rrrr, "{P_F}^{\to}"] \arrow[d, "\Sigma_{!_X}"'] & & & & (\mathcal{E}_{/I})_{/P_F\, X} \arrow[dll, "\Sigma_{P_F\, !_X}"'] \arrow[d, "\Sigma_{!_{P_F\, X}}"] \\
\mathcal{E}_{/I} \cong (\mathcal{E}_{/I})_{/\mathbb{1}} \arrow[rr, "{P_F}^{\to}"'] \arrow[rrrr, bend right, "P_F"'] & & (\mathcal{E}_{/I})_{/P_F\, \mathbb{1}} \arrow[rr, "\Sigma_{!_{P_F\, \mathbb{1}}}"'] & & (\mathcal{E}_{/I})_{/\mathbb{1}} \cong \mathcal{E}_{/I}
\end{tikzcd}$$

Indeed, unfolding the definition of $\Sigma_f \triangleq f \circ \_$, the left square reduces to the functoriality of P_F. The right triangle is simply the op-cartesian lifting of

$$\begin{tikzcd}
& P_F\, X \arrow[dl, "P_F\, !_X"'] \arrow[d, "!_{P_F X}"] \\
P_F\, \mathbb{1} \arrow[r, "!_{P_F \mathbb{1}}"'] & \mathbb{1}
\end{tikzcd} .$$

The bottom triangle commutes by the isomorphism relating the slice over the terminal and the total category, *i.e.* $(\mathcal{E}_{/I})_{/\mathbb{1}} \cong \mathcal{E}_{/I}$.

There is also a cartesian natural transformation from Σ_α to the identity polynomial indexed by J:

$$\begin{tikzcd}
\mathcal{E}_{/\Sigma_! P_F\, X} \cong (\mathcal{E}_{/I})_{/P_F\, X} \arrow[r, "\Sigma_\alpha"] \arrow[d, "\Sigma_{!_{P_F X}}"'] \arrow[dr, phantom, "\cong"] & (\mathcal{E}_{/I})_{/X} \cong \mathcal{E}_{/\Sigma_! X} \arrow[d, "\Sigma_{!_X}"] \\
\mathcal{E}_{/I} \cong (\mathcal{E}_{/I})_{/\mathbb{1}} \arrow[r, equal] & (\mathcal{E}_{/I})_{/\mathbb{1}} \cong \mathcal{E}_{/I}
\end{tikzcd}$$

By horizontal composition of these two cartesian natural transformations, we obtain a cartesian natural transformation from ${P_F}^{\alpha}$ to $\mathrm{id} \circ P_F \cong P_F$.

□

Remark 2. This proof is not entirely satisfactory: it is specialised to the predicate lifting in the codomain fibration. The construction of the cartesian natural transformation from the lifting to the functor is therefore a rather pedantic construction. Hopefully, a more abstract proof could be found.

4.3 Categorical structures

Identity A trivial ornamental construction is the *identity* ornament. Indeed, for every polynomial, there is a cartesian morphism from and to itself, introducing no extension and no refinement. In terms of Orn code, this construction simply consists in copying the code of the description: this is a generic program, taking a description as input and returning the identity ornament.

Vertical composition The next structure of interest is composition. Recall that an ornament corresponds to a (cartesian) natural transformation. There are therefore two notions of composition. First, vertical composition lets us collapse chains of ornaments:

$$\begin{tikzcd}
\mathcal{E}_{/I} \arrow[r, bend left=60, "F"] \arrow[r, "G" description] \arrow[r, bend right=60, "H"'] \arrow[r, phantom, shift left=3ex, "\Downarrow o_1"] \arrow[r, phantom, shift right=3ex, "\Downarrow o_2"] & \mathcal{E}_{/J} & \cong & \mathcal{E}_{/I} \arrow[r, bend left=60, "F"] \arrow[r, bend right=60, "H"'] \arrow[r, phantom, "\overset{o_2 \bullet o_1}{\Downarrow}"] & \mathcal{E}_{/J}
\end{tikzcd}$$

Example 16 (Vertical composition of ornaments)**.** We have seen that List ornaments Nat. We also know that Vec ornaments List. By vertical composition, we thus obtain that Vec ornaments Nat.

Horizontal composition Turning to horizontal composition, we have the following identity:

$$\begin{tikzcd}
\mathcal{E}_{/I} \arrow[r, bend left=30, "F_1"] \arrow[r, bend right=30, "G_1"'] \arrow[r, phantom, "\Downarrow o_1"] & \mathcal{E}_{/J} \arrow[r, bend left=30, "F_2"] \arrow[r, bend right=30, "G_2"'] \arrow[r, phantom, "\Downarrow o_2"] & \mathcal{E}_{/K} \\
\cong & & \\
\mathcal{E}_{/I} \arrow[rr, bend left=20, "F_2 \circ F_1"] \arrow[rr, bend right=20, "G_2 \circ G_1"'] \arrow[rr, phantom, "\overset{o_2 \circ o_1}{\Downarrow}"] & & \mathcal{E}_{/K}
\end{tikzcd}$$

Example 17 (Horizontal composition of ornaments)**.** Let us consider the following polynomials:

$$\begin{aligned}
&\mathsf{Square}\ X \mapsto X \times X : \textsc{Set}_{/\mathbb{1}} \to \textsc{Set}_{/\mathbb{1}} \\
&\mathsf{Height}\ \{X_n \mid n \in \mathsf{Nat}\} \mapsto \{X_n \times X_{n+1} \mid n \in \mathsf{Nat}\} \\
&\qquad\qquad + \{X_n \times X_n \mid n \in \mathsf{Nat}\} : \textsc{Set}_{/\mathsf{Nat}} \to \textsc{Set}_{/\mathsf{Nat}}
\end{aligned}$$

It is easy to check that $\mathsf{VecCont}$ ornaments $\mathsf{ListCont}$ and Height ornaments Square. By horizontal composition of these ornaments, we obtain that $\mathsf{VecCont} \circ \mathsf{Height}$ – describing a balanced binary tree – is an ornament of $\mathsf{ListCont} \circ \mathsf{Square}$ – describing a binary tree. Thus, we obtain that balanced binary trees ornament binary trees.

Frame structure Finally, the frame structure of the bicategory lets us lift morphisms on indices to polynomials.

Example 18 (Reindexing ornament)**.** Let twice : Nat → Even, the function that multiplies its input by 2. The Vec polynomial is indexed by Nat: we can therefore reindex it with twice. We automatically obtain an ornament of Vec that is indexed by Even. Needless to say, this construction is not very interesting on its own. However, in a larger development, we can imagine retrofitting an indexed datatype to use another index, making it usable by a library function.

The identity, vertical, and horizontal compositions illustrate the algebraic properties of ornaments. The categorical simplicity of cartesian morphisms gives us a finer understanding of datatypes and their relation to each other, as illustrated by Example 17.

4.4 Pullback of ornaments

So far, we have merely exploited the fact that $\mathit{PolyFun}^c_{\mathcal{E}}$ is a framed bicategory. However, it has a much richer structure. That extra structure can in turn be translated into ornamental constructions. We shall focus on pullbacks, but we expect other categorical notions to be of programming interest.

Proposition 4. *The category $\mathit{PolyFun}^c_{\mathcal{E}}$ has all pullbacks.*

Proof. First, let us recall that the notion of *cartesian* morphism arises from the fact that the following functor is a fibration:

$$\begin{tikzcd}
{[\mathcal{E}_{/I}, \mathcal{E}_{/J}]} \arrow[d, "\_\,\mathbb{1}"] \\
\mathcal{E}_{/J}
\end{tikzcd}$$

Where cartesian natural transformation corresponds to the cartesian morphisms of that fibration.

Let $\phi : F \dot{\to}^c H$ and $\psi : G \dot{\to}^c H$ two cartesian natural transformation. They are projected to $\phi_{\mathbb{1}}$ and $\psi_{\mathbb{1}}$ in the base category. Since $\mathcal{E}_{/J}$ is pullback complete, we can construct the pullback of $\phi_{\mathbb{1}}$ and $\psi_{\mathbb{1}}$, thus obtaining the following pullback square:

$$\begin{tikzcd}
\cdot \arrow[r, "\psi^{\dagger}_{\mathbb{1}}"] \arrow[d, "\phi^{\dagger}_{\mathbb{1}}"'] \arrow[dr, phantom, "\lrcorner", very near start] & F\mathbb{1} \arrow[d, "\phi_{\mathbb{1}}"] \\
G\mathbb{1} \arrow[r, "\psi_{\mathbb{1}}"'] & H\mathbb{1}
\end{tikzcd}$$

By reindexing, we thus obtain the following square in the total category:

$$\begin{tikzcd}
\cdot \arrow[r, "{\psi^\dagger}_X"] \arrow[d, "{\phi^\dagger}_X"'] & FX \arrow[d, "\phi_X"] \\
GX \arrow[r, "\psi_X"'] & HX
\end{tikzcd}$$

By Exercise 1.4.4 [Jacobs, 2001], we have that this square is actually a pullback. In a nutshell, we rely on the unicity of cartesian morphisms in the total category to prove the universal property of pullbacks for that square.

□

Example 19 (Pullback of ornament)**.** Natural numbers can be ornamented to lists (Example 13) as well as finite sets (Example 14). Taking the pullback of these two ornaments, we obtain bounded lists that correspond to lists of bounded length, with the bound given by an index $n : \mathsf{Nat}$. Put explicitly, the object thus computed is the following datatype:

```
data BoundedList [A : SET](n : Nat) : SET where
  BoundedList_A (n = suc n')  ∋ nil (n' : Nat)
                              | cons (n' : Nat)(a : A)(as : BoundedList_A n')
```

The pullback construction is another algebraic property of ornaments: given two ornaments, both describing an extension of the same datatype (*e.g.* extending natural numbers to lists and extending natural numbers to finite sets), we can "merge" them into one having both characteristics (*i.e.* bounded lists). In type theory, Ko and Gibbons [2011] have experimented with a similar construction for composing indexing disciplines.

4.5 Derivative of ornament

Abbott et al. [2005b] have shown that the Zipper [Huet, 1997] data-structure can be computed from the derivative of signature functors. Interestingly, the derivative is characterised by the existence of a universal arrow in the category $Poly^{c}_{\mathcal{E}}$:

Definition 14 (Differentiability [Abbott et al., 2005b])**.** A polynomial F is differentiable in i if and only if, for any polynomial G, we have the following bijection of morphisms:

$$\frac{Poly^{c}_{\mathcal{E}}(G \times \pi_i, F)}{Poly^{c}_{\mathcal{E}}(G, \partial_i F)}$$

Where $\pi_i \triangleq I \xleftarrow{k_i} I \xrightarrow{\mathrm{id}} I \xrightarrow{\mathrm{id}} I$. We denote $Poly^{\partial_i}_{\mathcal{E}}$ the class of polynomials differentiable in i.

Proposition 5. *Let F and G two polynomials in $Poly^{\partial_i}_{\mathcal{E}}$.*
If F ornaments G, then $\partial_i F$ ornaments $\partial_i G$.

Proof. The proof simply follows from the functoriality of ∂_i over $Poly_{\mathcal{E}}^{\partial_i}$ [Abbott, 2003, Section 6.4]. In a nutshell, this follows from the existence of the following cartesian morphism:

$$\partial_i F \times \pi_i \to^c F \to^c G$$

where the first component is the unit of the universal arrow while the second component is the ornament from F to G. By definition of differentiability, we therefore have the desired cartesian morphism:

$$\partial_i F \to^c \partial_i G$$

□

Example 20 (Ornamentation of derivative)**.** Let us consider binary trees, with signature functor $1 + A \times X^2$. Balanced binary trees are an ornamentation of binary trees (Example 17). By the theorem above, we have that the derivative of balanced binary trees is an ornament of the derivative of binary trees.

The derivative is thus an example of an operation on datatypes that preserves ornamentation. Knowing that the derivative of an ornamented datatype is an ornamentation of the derivative of the original datatype, we get that the order in which we ornament or derive a datatype does not matter. This let us relate datatypes across such transformations, thus preserving the structural link between them.

5 Related work

Ornaments were initially introduced by the second author [McBride, 2013] as a programming artefact. They were presented in type theory, with a strong emphasis on their computational contribution. Ornaments were thus introduced through a universe. Constructions on ornaments – such as the ornamental algebra, algebraic ornament, and reornament – were introduced as programs in this type theory, relying crucially on the concreteness of the universe-based presentation.

While this approach has many pedagogical benefits, it was also clear that more abstract principles were at play. For example, in a subsequent paper [Dagand and McBride, 2012], the authors successfully adapted the notion of ornaments to another universe of inductive families, whilst Ko and Gibbons [2011] explore datatype engineering with ornaments in yet a third. The present paper gives such an abstract treatment. This focus on the theory behind ornaments thus complements the original, computational treatment.

Building upon that original paper, our colleagues Ko and Gibbons [2011] also identify the pullback structure – called "composition" in their paper – as significant, giving a treatment for a concrete universe of ornaments and compelling examples of its effectiveness for combining indexing disciplines. The conceptual simplicity of our approach lets us subsume their type theoretic construction as a mere pullback.

The notion of algebraic ornament was also treated categorically by Atkey et al. [2012]: instead of focusing on a restricted class of functors, the authors described the refinement of any functor by any algebra. The constructions are presented in the generic framework of fibrations. The refinement construction described in this paper, once specialised to polynomial functors, corresponds exactly to the notion of algebraic ornament, as we have shown.

Hamana and Fiore [2011] also give a model of inductive families in terms of polynomial functors. To do so, they give a translation of inductive definitions down to polynomials. By working on the syntactic representation of datatypes, their semantics is *defined by* this translation. In our system, we can actually prove that descriptions – our language of datatypes – are equivalent to polynomial functors.

Finally, it is an interesting coincidence that cartesian morphisms should play such an important role in structuring ornaments. Indeed, containers stem from the work on shapely types[Jay and Cockett, 1994]. In the shape framework, a few base datatypes were provided (such as natural numbers) and all the other datatypes were grown from these basic blocks by a pullback construction, *i.e.* an ornament. However, this framework was simply typed, hence no indexing was at play.

6 Conclusion

Our study of ornaments began with the equivalence between our universe of descriptions and polynomial functors. This result lets us step away from type theory, and gives access to the abstract machinery provided by polynomials. For practical reasons, the type theoretic definition of our universe is very likely to change. However, whichever concrete definition we choose will always be a syntax for polynomial functors. We thus get access to a stable source of mathematical results that informs our software constructions.

We then gave a categorical presentation of ornaments. Doing so, we get to the essence of ornaments: ornamenting a datatype consists in extending it with new information, and refining its indices. Formally, this characterisation turns into a presentation of ornaments as cartesian morphisms of polynomials.

Finally, we reported some initial results based on our explorations of this categorical structure. We have translated the type theoretic ornamental toolkit to the categorical framework. Doing so, we have gained a deeper understanding of the original definitions. Then, we have expressed the categorical definition of $\mathit{Poly}^{c}_{\mathcal{E}}$ in terms of ornaments, discovering new constructions – identity, vertical, and horizontal composition – in the process. Also, we have studied the structure of $\mathit{Poly}^{c}_{\mathcal{E}}$, obtaining the notion of pullback of ornaments.

Future work We have barely scratched the surface of $\mathit{Poly}^{c}_{\mathcal{E}}$: a lot remain unexplored. Pursuing this exploration might lead to novel and interesting ornamental constructions. Also, our definition of ornaments in terms of polynomials might be limiting. One can wonder if a more abstract criterion could be found for a larger class of functors. For instance, the functor $\_\ \mathbb{1}_{\mathbb{C}} : [\mathbb{C}, \mathbb{D}] \to \mathbb{D}$ is a fibration for $\mathbb{D}$ pullback complete and $\mathbb{C}$ equipped with a terminal object $\mathbb{1}_{\mathbb{C}}$. Specialised to the categories of slices of $\mathcal{E}$, the cartesian morphisms are exactly our ornaments. What about the general case?

Finally, there has been much work recently on homotopy inductive types [Awodey et al., 2012]. Coincidentally, the formalism used in these works is based on W-types, *i.e.* the type theoretic incarnation of polynomial functors. It would be there be interesting to study what ornaments could express in this framework.

Acknowledgements We would like to thank Gabor Greif and José Pedro Magalhães for their input on this paper. We also thank our colleagues Clément Fumex and Lorenzo Malatesta for their feedback on our proofs. The authors are supported by the Engineering and Physical Sciences Research Council, Grant EP/G034699/1.

References

M. Abbott. *Categories of Containers.* PhD thesis, University of Leicester, 2003.

M. Abbott, T. Altenkirch, and N. Ghani. Containers: Constructing strictly positive types. *TCS*, 342(1):3–27, 2005a.

M. Abbott, T. Altenkirch, C. McBride, and N. Ghani. ∂ for data: Differentiating data structures. *Fundam. Inform.*, 65(1-2):1–28, 2005b.

R. Atkey, P. Johann, and N. Ghani. Refining inductive types. *Logical Methods in Computer Science*, 2012.

S. Awodey, N. Gambino, and K. Sojakova. Inductive types in homotopy type theory. Jan. 2012.

J. Chapman, P.-E. Dagand, C. McBride, and P. Morris. The gentle art of levitation. In *ICFP*, pages 3–14, 2010.

P.-L. Curien. Substitution up to isomorphism. *Fundam. Inf.*, 19(1-2):51–85, Sept. 1993. ISSN 0169-2968.

P.-E. Dagand and C. McBride. Transporting functions across ornaments. In *ICFP*, pages 103–114, 2012.

P.-E. Dagand and C. McBride. Elaborating inductive definitions. In *JFLA*, February 2013.

P. Dybjer. Inductive sets and families in Martin-Löf's type theory. In *Logical Frameworks.* 1991.

T. Freeman and F. Pfenning. Refinement types for ML. *SIGPLAN Not.*, 26:268–277, May 1991.

C. Fumex. *Induction and coinduction schemes in category theory.* PhD thesis, University of Strathclyde, 2012.

N. Gambino and M. Hyland. Wellfounded trees and dependent polynomial functors. In *TYPES*, volume 3085 of *LNCS*, pages 210–225. 2004.

N. Gambino and J. Kock. Polynomial functors and polynomial monads. *CoRR*, abs/0906.4931, Mar. 2010.

M. Hamana and M. Fiore. A foundation for GADTs and inductive families: dependent polynomial functor approach. In *WGP'11*, pages 59–70. ACM, 2011.

P. Hancock and P. Hyvernat. Programming interfaces and basic topology. *Annals of Pure and Applied Logic*, 137(1-3):189–239, Jan. 2006.

C. Hermida and B. Jacobs. Structural induction and coinduction in a fibrational setting. *Inf. Comput.*, 145(2):107–152, 1998.

G. Huet. The zipper. *Journal of Functional Programming*, 7(05):549–554, Sept. 1997.

B. Jacobs. *Categorical Logic and Type Theory, Volume 141 (Studies in Logic and the Foundations of Mathematics)*. Elsevier Science, May 2001.

C. Jay and J. Cockett. Shapely types and shape polymorphism. In *ESOP '94*, volume 788 of *LNCS*, pages 302–316. 1994.

H.-S. Ko and J. Gibbons. Modularising inductive families. In *Workshop on Generic Programming*, pages 13–24, 2011.

S. Mac Lane and I. Moerdijk. *Sheaves in Geometry and Logic: A First Introduction to Topos Theory (Universitext)*. Springer, corrected edition, May 1992.

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

C. McBride. Ornamental algebras, algebraic ornaments. *Journal of Functional Programming, to appear*, 2013.

I. Moerdijk and E. Palmgren. Wellfounded trees in categories. *Annals of Pure and Applied Logic*, 104:189–218, July 2000.

P. Morris. *Constructing Universes for Generic Programming*. PhD thesis, University of Nottingham, 2007.

U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

K. Petersson and D. Synek. A set constructor for inductive sets in martin-löf's type theory. In *CTCS*, pages 128–140, 1989.

R. A. G. Seely. Locally cartesian closed categories and type theory. *Mathematical Proceedings of the Cambridge Philosophical Society*, 95, 1983.

M. A. Shulman. Framed bicategories and monoidal fibrations. *CoRR*, abs/0706.1286, Jan. 2009.

The Coq Development Team. *The Coq Proof Assistant Reference Manual*.