*Under consideration for publication in J. Functional Programming* 

# *The Essence of Ornaments*

Pierre-Evariste Dagand
Sorbonne Universités, UPMC Univ Paris 06, CNRS, Inria, LIP6 UMR 7606
(*e-mail:* pierre-evariste.dagand@lip6.fr)

**Abstract**

Functional programmers from all horizons strive to use, and sometimes abuse, their favorite type system in order to capture the invariants of their programs. A widely-used tool in that trade consists in defining finely-indexed datatypes. Operationally, these types *classify* the programmer's data, following the ML tradition. Logically, these types *enforce* the program invariants in a novel manner. This new programming pattern, by which one programs *over* inductive definitions to account for some invariants, lead to the development of a theory of ornaments (McBride, 2011).

However, ornaments originate as a dependently-typed object and may thus appear rather daunting to a functional programmer of the non-dependent kind. This article aims at presenting ornaments *from first-principles* and, in particular, to declutter their presentation from syntactic considerations. To do so, we shall give a sufficiently abstract model of indexed datatypes by means of many-sorted signatures. In this process, we formalize our intuition that an indexed datatype is the combination of a data-*structure* and a data-*logic*.

Over this abstraction of datatypes, we shall recast the definition of ornaments, effectively giving a model of ornaments. Benefiting both from the operational *and* abstract nature of many-sorted signatures, ornaments should appear applicable and, one hopes, of interest beyond the type-theoretic circles, case in point being languages with generalized abstract datatypes or refinement types.

## 1 Introduction

In modern programming languages, datatypes are more than mere data-*structures*. With the advent of indexed datatypes in mainstream languages (Cheney & Hinze, 2003; Sheard & Linger, 2007; Freeman & Pfenning, 1991; Swamy *et al.*, 2011), programmers have gained the ability to precisely capture the *logical* invariants of their programs. A typical example of a definition combining structure and logic is the datatype of finite sets

```
data Fin (n : Nat) : SET where
  Fin    0    ∋
  Fin (suc n) ∋ f0
              | fsuc (k : Fin n)
```

that, in effect, represents a number between 0 and $n-1$. As a data-*structure*, a finite set is nothing but a (unary) number. As a data-*logic*, it captures the invariant that this number is bounded by $n$. Another example is the datatype of vectors

```
data Vec [A : SET](n : Nat) : SET where
  Vec A    0    ∋ nil
  Vec A (suc n) ∋ cons (a : A)(vs : Vec A n)
```

that represents lists of a fixed length $n$. Structurally, vectors are nothing but lists. However, logically, we have a static guarantee on their length. Using this logical information, we can for example write a type-safe lookup function for vectors:

$$\begin{array}{lll} \mathsf{vlookup}\ (m : \mathsf{Fin}\ n)\ (vs : \mathsf{Vec}\ A\ n) & : & A \\ \mathsf{vlookup}\quad \mathsf{f0}\quad\ (\mathsf{cons}\ a\ xs) & \mapsto & a \\ \mathsf{vlookup}\ \ (\mathsf{fsuc}\ n)\ \ (\mathsf{cons}\ a\ xs) & \mapsto & \mathsf{vlookup}\ n\ xs \end{array}$$

Using finely indexed datatypes, the programmer is thus able to express her invariants, relying on the type checker to enforce them. By following this approach, one gets a step closer to *correct-by-construction* software. In our vlookup example above, we are able to sidestep the logically meaningless case where the index is beyond the end of the list. In a non-indexed setting, we have to handle these cases at run-time:

$$\begin{array}{lll} \mathsf{lookup}\ (m : \mathsf{Nat})\ (xs : \mathsf{List}\ A) & : & \mathsf{Maybe}\ A \\ \mathsf{lookup}\quad m\quad\ \ \mathsf{nil} & \mapsto & \mathsf{nothing} \\ \mathsf{lookup}\quad 0\quad\ (\mathsf{cons}\ a\ xs) & \mapsto & \mathsf{just}\ a \\ \mathsf{lookup}\ \ (\mathsf{suc}\ n)\ \ (\mathsf{cons}\ a\ xs) & \mapsto & \mathsf{lookup}\ n\ xs \end{array}$$

Indexed datatypes exist in many forms. Originally, they were studied in (dependent) type theory under the name of *inductive families* (Dybjer, 1994). They are at the heart of proof assistants such as Coq (Coq development team, 2015), and programming languages such as Agda (Norell, 2007) or Idris (Brady, 2013). The examples we gave above use the full-power of dependent types, using a stylized syntax.

Inductive families have percolated in the ML family under the guise of Generalized Algebraic Datatypes (GADTs). Initially implemented in Omega (Sheard & Linger, 2007), GADTs are now a basic convenience of Haskell (Schrijvers *et al.*, 2009) and OCaml (Pottier & Régis-Gianas, 2006). For example, the type of finite sets is defined as follows in the dialect of Haskell supported by GHC:

```
{-# LANGUAGE GADTs, DataKinds, KindSignatures #-}
data Nat where
  Ze :: Nat
  Suc :: Nat -> Nat

data Fin :: Nat -> * where
  FZe :: Fin (Suc n)
  FSuc :: Fin n -> Fin (Suc n)
```

*Refinement types* are another form of indexing, which serves as the basis of languages such as RefinementML (Freeman & Pfenning, 1991) or F$^\star$ (Swamy *et al.*, 2011). A refinement type consists of a bare datatype paired with a refinement predicate that asserts a (logical) property of the underlying type. In F$^\star$, the type of finite sets is thus expressed as a natural number $k$ that is less than the index $n$:

```
type fin (n:nat) = k:nat{k < n}
let fze : n:nat -> fin (n + 1) = fun _ -> 0
let fsuc : n:nat -> fin n -> fin (n+1) = fun _ k -> k + 1
```

While initially developed in dependent type theory, the concept of ornament makes sense in these settings too. For ornaments to be widely applicable, we want to cast them in a single abstract framework, absorbing this diversity once and for all. As it turns out, the model of indexed datatypes as least fixpoint of *many-sorted signatures* (Petersson & Synek, 1989; Morris *et al.*, 2009; Gambino & Kock, 2013). fits perfectly our purposes. In type theoretic circles, it is known as *indexed containers* (Morris & Altenkirch, 2009; Abbott, 2003), or *containers* for short. In categorical circles, it is known as *polynomial functors* (Gambino & Hyland, 2004; Gambino & Kock, 2013) over locally Cartesian-closed categories (LCCC). Polynomial functors are known to be equivalent to containers (Gambino & Kock, 2013).

In this setting, we understand an indexed datatype as the fixpoint of a signature. The operations of the signature and their respective arities correspond to the datatype constructors. The sorts of the signature correspond to the indices of the datatype. The typing captures the indexing discipline of the datatype. For example, finite sets are understood as the fixpoint of a signature with two operations corresponding, respectively, to the zero constructor and the successor constructor. The former takes no argument, its arity is thus 0, while the latter takes one recursive arguments, its arity is therefore 1. This signature is indexed by the sort of natural numbers, so as to ensure that the number of constructors is bounded: this is enforced by the typing discipline. We come back to the nuts and bolts of this definition in Example 2.5.

By working with many-sorted signatures, we can reason about indexed datatypes in the abstract, *extensionally*. This formalism materializes the intuition that indexed datatypes are a combination of a data-*structure* and of a data-*logic*. The choice of data-structure is forced upon us by dynamic, computational considerations: it is dictated by the run-time behavior we expect from the programs operating over the structure. For example, it corresponds to the difference between a list and a tree: the linear structure of the former allows for fast insertions of elements, while the branching structure of the latter allows for efficient searches. The world of ML datatypes is essentially a world of data-structures.

The choice of data-logic is governed by static, logical considerations: we enforce the invariants of our programs by expressing them at the level of types. By accepting our program, the type checker guarantees that *every* potential execution follows the data-logic. For example, this corresponds to the difference between a natural number and a finite set. At run-time, a finite set is nothing but a natural number. The logical information has no impact on the run-time behavior. However, finite sets – bounded natural numbers – provide more information to the type checker. This extra-information can then be used to enforce our program's invariant, as we did for the vlookup function.

For a given data-structure, we can combine it with various data-logics. Indeed, a data-logic captures an invariant suited for a specific use-case. We are likely to enforce multiple invariants over a single data-structure. For example, starting from binary trees, we might be interested in representing well-formed red-black trees (Guibas & Sedgewick, 1978) or well-formed AVL trees (Adelson-Velskii & Landis, 1962). Structurally, both are nothing but binary trees. Logically, however, they enforce incompatible invariants. Ornaments are a device that allows the programmer to graft data-logics over data-structures. Thanks to their rich algebra, ornaments enable precise invariants to be expressed, while preserving the structural ties that relate a datatype to its ornamented self.

We shall therefore present ornaments through the angle of *data-logic engineering*. We will see that, intrinsically, ornaments are simply structure-preserving transformations of datatypes. We shall illustrate how, using ornaments, the programmer can put his finger on a particular data-structure and engineer data-logics enforcing domain-specific invariants.

**Relation with previous presentations:** McBride (2011) has originally introduced ornaments in type theory, over a specific *universe of inductive families*. This seminal paper was focused on an operational account of ornaments, thus the emphasis on a syntactic approach through the use of universes. This approach was adapted to other universes by various authors, including Ko and Gibbons (2011) and the present author (Dagand & McBride, 2012). A purely categorical approach was also taken by the present author (Dagand & McBride, 2013). The objective was to provide a categorical semantics to ornaments. This was achieved by relating ornaments to specific morphisms – the Cartesian morphisms – in the category of polynomial functors (Gambino & Kock, 2013). Being categorical, this work was focused on extensional properties of ornaments, and is thus far removed from the programmer's concerns.

This article aims at establishing a middle-ground between these two presentations. On the one hand, we take a step in abstraction compared to the universe-based approach. Working over signatures allows us to get past the syntactic details and focus on the *essence* of ornaments. On the other hand, this model is concrete enough to appeal to a programmer's intuition, allowing us to rely on the computational meaning of various transformations on datatypes. Throughout this article, we shall focus on the concepts, leaving aside the proofs (in particular, of adequacy of our model). The interested reader will find these results in the more abstract setting (Dagand & McBride, 2013).

This article is a first-principles approach to ornaments. We shall motivate and formalize ornaments through the duality of *data-structure* and *data-logic*. Striving for generality, we first free ourselves from the syntactic peculiarities of datatypes, without sacrificing the computational aspects. In this framework, we then delineate what ornaments *are*, formally and in the abstract. Our contributions are the following:

- We recall a model of indexed datatypes based on many-sorted signatures (Section 2), expressed in the language of extensional type theory. We provide a wealth of examples to develop our understanding of datatypes in terms of *operation*, *arity*, and *sorts*. While not novel, this model is a key step toward a more abstract treatment of indexed datatypes and operations on them (such as ornaments). In particular, it provides a unifying framework in which to express the many forms of indexing, such as GADTs, refinement types, and indexed families.
- We then formally pin down the essence of ornaments in terms of structure-preserving transformations of signatures (Section 3). We shall see that an ornament is simply a device that lets us explain how to enrich a data-structure with a given data-logic. We come to grips with this intuition in two steps. First, we present a naive model of ornaments, mimicking the original definition of ornaments (McBride, 2011) on signatures. However, this first model is not mathematically pleasing: it focuses exclusively on transforming signatures, leaving aside the specific *invariant* governing the transformation. We refine it into a second model – expressed in terms of *Cartesian*

*morphism* of signatures. Aside from obtaining a more elegant definition, we shall see that ornaments are in fact a surprisingly simple notion. Coincidentally, it also corresponds to a widely-studied object in category theory.

- Having established a deep connection between ornaments and Cartesian morphisms, the algebraic properties of Cartesian morphisms turn into potential software artifacts. We shall hint at a few examples of such a *calculus of datastructures* (Section 4). This illustrates some of the categorical structure of Cartesian morphisms in terms of constructions on ornaments (*e.g.*, composition). We also cast McBride's original constructions in terms of operations in our calculus (*e.g.*, the ornamental algebra).

**Formal framework:** The models presented in this article are developed in extensional type theory (Martin-Löf, 1984; Constable, 1986). This formal framework offers a compromise between an intensional type theory – used by McBride in his original presentation of ornaments (McBride, 2011) – that forces a laboriously syntactic presentation; and category theory – used by the present author in his categorical treatment of ornaments (Dagand & McBride, 2013) – that glances over the computational meaning of ornaments.

By working in extensional type theory, we can manipulate inductive definitions as a programming object and treat ornaments as computational artifacts. At the same time, by working extensionally over signatures, we gain access to an unrestricted, abstract structure without too much syntactic noise.

The syntax of our type theory is conventional. We denote by $\textsc{Set}$ the type of all (small) types and $\textsc{Set}_1$ the type of all $\textsc{Set}$-types (we shall not need the full hierarchy of types). $\Pi$-types are written with a dependent arrow $(x:A) \to B$, $\Sigma$-types with a dependent product $(x:A) \times B$, for $A, B : \textsc{Set}$ where $x$ may occur freely in $B$. When $x$ does not occur freely in $B$, the $\Pi$-type degenerates into a simply-typed function space, written $A \to B$, while the $\Sigma$-type degenerates into a product type, written $A \times B$. We require our type theory to contain an empty type, written $\mathbb{0}$, a unit type, written $\mathbb{1}$, a sum type, written $A + B$, for $A, B : \textsc{Set}$. The (only) inhabitant of the unit type is $*$, the injections into the sum type are $\mathsf{inj}_l$ and $\mathsf{inj}_r$, while products are inhabited by pairs $(a, b)$. We rely on an intuitive pattern-matching notation to represent the eliminators. We write $\pi_0\ p$ (respectively, $\pi_1\ p$) to compute the first (respectively, second) projection of a pair $p$.

As in ML, unbound variables in type definitions are universally quantified. For example, we do not explicitly quantify over $n : \mathsf{Nat}$ or $A : \textsc{Set}$ in the definition of vector lookup:

$$\mathsf{vlookup}\ (m : \mathsf{Fin}\ n)\ (vs : \mathsf{Vec}\ A\ n) : A$$

While this dramatically reduces the burden of quantifiers, this does not cover all the cases where one would want to declare an argument as implicit. Case in point are higher-order type signatures. To indicate that an argument is implicit, we use the quantifier $\forall x.\,(\ldots)$ – or its annotated variant $\forall x : T.\,(\ldots)$ – as follows:

$$\mathsf{example}\ (f : \forall n.\, \mathsf{Vec}\ A\ n \to \mathbb{1})\ (xs : \mathsf{Vec}\ A\ k)\ (ys : \mathsf{Vec}\ A\ l) : f\ xs = f\ ys$$

This stylized syntax allows us to focus on the key aspects of our type-theoretic constructions. This notation aims at striking a balance between formalism and readability. For a fully formal and mechanized presentation, we refer the reader to the accompanying Agda development, which is available on the author's website.

## 2 Indexed datatypes = Data-structure + Data-logic

The aim of this section is to provide the reader with the tools to understand indexed datatypes in terms of structure and logic. To this end, we are going to present a model of indexed datatypes for which these two components are clearly visible. Besides, we shall illustrate our model with concrete examples, shedding a more structural/logical light on them. We have been careful to adopt a model which is extensional enough to allow abstract reasoning, freeing us from any unnecessary syntactic details.

Let us recall the definition of many-sorted signatures in type theory, using a terminology inspired from universal algebra. A signature is parameterized over a set $I : \mathrm{SET}$ of sorts, to account for the "many-sorted" nature of the signature. A signature consists of:

- a family of operations, one set for each sort:

$$\mathsf{Op} : I \to \mathrm{SET}$$

- a family of arities, one set for each operation:

$$\mathsf{Ar} : \mathsf{Op}\, i \to \mathrm{SET}$$

- a typing discipline, assigning a sort to each arity of each operation:

$$\mathsf{Ty} : \mathsf{Ar}\, op \to I$$

Altogether, the triple of $(\mathsf{Op}, \mathsf{Ar}, \mathsf{Ty})$ defines a signature. For conciseness, we organize these triples in a record-like structure of signatures indexed by a given set of sorts:

$$\begin{array}{lcl} \mathsf{Sig}\,(I : \mathrm{SET}) & : & \mathrm{SET}_1 \\ \mathsf{Sig} \quad I & \mapsto & \left\{ \begin{array}{l} \mathsf{Op} : I \to \mathrm{SET} \\ \mathsf{Ar} : \mathsf{Op}\, i \to \mathrm{SET} \\ \mathsf{Ty} : \mathsf{Ar}\, op \to I \end{array} \right. \end{array}$$

To visually distinguish these components, we write the triple $(\mathsf{Op}, \mathsf{Ar}, \mathsf{Ty})$ as $\mathsf{Op} \lhd^{\mathsf{Ty}} \mathsf{Ar}$.

The idea that datatypes can be understood as least fixpoint of signatures goes as far back as Goguen (1975) in a simply-typed (*i.e.*, mono-sorted) setting. In the indexed setting, we merely had to move to many-sorted signatures (Petersson & Synek, 1989), such as the ones described above. Doing so, we model the index of the datatype by a sort: a datatype indexed by $I$ is described by a signature in $\mathsf{Sig}\, I$. The constructors and their non-recursive arguments are modeled by the family of operations. The recursive arguments of a constructor are modeled by the arity of the signature. Finally, the indexing discipline is modeled by the typing discipline of the signature.

**2.1 Example** (Signature: vectors)**.** From the inductive definition of vectors

$$\begin{array}{l} \textbf{data}\ \mathsf{Vec}\,[A : \mathrm{SET}]\,(n : \mathsf{Nat}) : \mathrm{SET}\ \textbf{where} \\ \quad \mathsf{Vec}\, A \quad 0 \quad \ni \mathsf{nil} \\ \quad \mathsf{Vec}\, A\, (\mathsf{suc}\, n) \ni \mathsf{cons}\ (a : A)\,(vs : \mathsf{Vec}\, A\, n) \end{array}$$

we can read off its signature as follows. The set of sorts is $\mathsf{Nat}$, *i.e.* the sorts are the naturals. The signature of vectors, which we call $\Sigma_{\mathsf{Vec}}$, is therefore an element of $\mathsf{Sig}\ \mathsf{Nat}$.

At sort 0, there is a single operation, nil. At sort suc $n$, there is an $A$-indexed family of operations, cons $a$:

$$\begin{array}{lll} \mathsf{Op}_{\mathsf{Vec}}\,(n:\mathsf{Nat}) & : & \textsc{Set} \\ \mathsf{Op}_{\mathsf{Vec}}\quad 0 & \mapsto & \mathbb{1} \\ \mathsf{Op}_{\mathsf{Vec}}\;(\mathsf{suc}\;n) & \mapsto & A \end{array}$$

Since nil takes no argument, its arity is $\mathbb{0}$. The operations cons $a$ take one argument, thus having arity $\mathbb{1}$:

$$\begin{array}{llll} \mathsf{Ar}_{\mathsf{Vec}}\,(n:\mathsf{Nat})\,(op:\mathsf{Op}_{\mathsf{Vec}}\;n) & : & \textsc{Set} \\ \mathsf{Ar}_{\mathsf{Vec}}\quad 0 \qquad\quad * & \mapsto & \mathbb{0} \\ \mathsf{Ar}_{\mathsf{Vec}}\;(\mathsf{suc}\;n) \qquad a & \mapsto & \mathbb{1} \end{array}$$

Being of arity null, the typing of nil is trivial. The typing of cons states that the recursive argument of a suc $n$-indexed constructor is $n$, *i.e.* a vector of length suc $n$ has a tail of length $n$:

$$\begin{array}{llll} \mathsf{Ty}_{\mathsf{Vec}}\,(n:\mathsf{Nat})\,(op:\mathsf{Op}_{\mathsf{Vec}}\;n)\,(ar:\mathsf{Ar}_{\mathsf{Vec}}\;n\;ar) & : & \mathsf{Nat} \\ \mathsf{Ty}_{\mathsf{Vec}}\;(\mathsf{suc}\;n) \qquad a \qquad\quad * & \mapsto & n \end{array}$$

We have thus specified the signature of $\Sigma_{\mathsf{Vec}} \triangleq \mathsf{Op}_{\mathsf{Vec}} \lhd^{\mathsf{Ty}_{\mathsf{Vec}}} \mathsf{Ar}_{\mathsf{Vec}}$.

△

Many-sorted signatures offer a convenient, minimalistic language for *describing* inductive families. To build the inductive object, we take the least fixpoint of the (strictly-positive) functor they describe. As it turns out, signatures have a rather straightforward functorial interpretation. Provided a family $X : I \to \textsc{Set}$, a signature taken at a given sort $i$ interprets to a choice (*i.e.*, a $\Sigma$-type) of an operation $op : \mathsf{Op}\,i$, followed by a product (*i.e.*, a $\Pi$-type) of arity $ar : \mathsf{Ar}\,op$ of the family $X$ taken at sort $\mathsf{Ty}\,ar$:

$$\begin{array}{l} [\![(\Sigma:\mathsf{Sig}\;I)]\!]\;(X:I\to\textsc{Set}) : I\to\textsc{Set} \\ [\![\mathsf{Op}\lhd^{\mathsf{Ty}}\mathsf{Ar}]\!]\;X \mapsto \lambda i.\,(op:\mathsf{Op}\,i)\times((ar:\mathsf{Ar}\,op)\to X\,(\mathsf{Ty}\,ar)) \end{array}$$

The resulting endofunctor on $I \to \textsc{Set}$ is strictly-positive. It is in fact a *polynomial functor* built from a sum over $op : \mathsf{Op}\,i$ of monomials X exponentiated by $\mathsf{Ar}\,op$.

*2.2 Remark.* Our notion of signature is very similar to the indexed containers of Altenkirch and Morris (2009). The only difference stands in our treatment of arities and typing. Altenkirch and Morris (2009) write a single "arity and typing" function

$$\mathsf{Ar}' : \mathsf{Op}\,i \to I \to \textsc{Set}$$

while we chose to separate the two, writing:

$$\begin{cases} \mathsf{Ar} : \mathsf{Op}\,i \to \textsc{Set} \\ \mathsf{Ty} : \mathsf{Ar}\,op \to I \end{cases}$$

These two definition styles are in fact equivalent: the interpretation of either style yields isomorphic functors. Indeed, for $I : \textsc{Set}$, we have that $I \to \textsc{Set} \cong (X : \textsc{Set}) \times (X \to I)$: we can therefore translate the signatures themselves from one style to the other.

Our style enforces a clear separation between *structure* (dictated by Ar) and *logic* (dictated by Ty): this is a key ingredient in our treatment of structurally-equivalent (but logically-incompatible) datatypes.

◊

Given a signature, we have given its functorial interpretation. Being strictly-positive, the interpreted functor admits a least fixpoint (Smyth & Plotkin, 1977). We can therefore safely define a least fixpoint operator

$$\begin{array}{ll} \mu\ (\Sigma : \mathsf{Sig}\ I) & : I \to \textsc{Set} \\ \mu\ \Sigma & \mapsto [\![\Sigma]\!]\ (\mu\ \Sigma) \end{array}$$

with the confidence that this set is well-formed. In fact, it admits an initial algebra semantics (and, by extension, an induction principle):

$$(\!|(\alpha : \forall i.\ [\![\Sigma]\!]\ X\ i \to X\ i)|\!) : \mu\ \Sigma\ i \to X\ i$$

In order to get acquainted with signatures, let us consider the signature of some common datatypes. We leave it to the reader to check that taking their least fixpoint yields the expected datatypes.

**2.3 Example** (Signature: natural numbers)**.** Natural numbers are defined as:

$$\begin{array}{l} \textbf{data}\ \mathsf{Nat} : \textsc{Set}\ \textbf{where} \\ \quad \mathsf{Nat} \ni \mathsf{0} \\ \qquad \mid \mathsf{suc}\ (n : \mathsf{Nat}) \end{array}$$

To describe this datatype, we only need mono-sorted signatures: we therefore work in $\mathsf{Sig}\ \mathbb{1}$, the class of signatures indexed by the set with a single inhabitant. Natural numbers offer only two operations, $\mathsf{0}$ or $\mathsf{suc}$:

$$\begin{array}{lll} \mathsf{Op}_{\mathsf{Nat}}\ (* : \mathbb{1}) & : & \textsc{Set} \\ \mathsf{Op}_{\mathsf{Nat}}\ \ * & \mapsto & \mathbb{1} + \mathbb{1} \end{array}$$

The arity of the operation $\mathsf{0}$ is null, while the arity of the operation $\mathsf{suc}$ is one:

$$\begin{array}{lll} \mathsf{Ar}_{\mathsf{Nat}}\ (op : \mathsf{Op}_{\mathsf{Nat}}\ *) & : & \textsc{Set} \\ \mathsf{Ar}_{\mathsf{Nat}}\ \ (\mathsf{inj}_l\ *) & \mapsto & \mathbb{0} \\ \mathsf{Ar}_{\mathsf{Nat}}\ \ (\mathsf{inj}_r\ *) & \mapsto & \mathbb{1} \end{array}$$

Because the index is trivial, so is the typing:

$$\begin{array}{lll} \mathsf{Ty}_{\mathsf{Nat}}\ (ar : \mathsf{Ar}_{\mathsf{Nat}}\ op) & : & \mathbb{1} \\ \mathsf{Ty}_{\mathsf{Nat}}\ \ ar & \mapsto & * \end{array}$$

Altogether, we have defined the signature $\Sigma_{\mathsf{Nat}} \triangleq \mathsf{Op}_{\mathsf{Nat}} \lhd^{\mathsf{Ty}_{\mathsf{Nat}}} \mathsf{Ar}_{\mathsf{Nat}}$. To check that we have indeed described the signature functor of natural numbers, we can simply interpret (by $[\![-]\!]$) the signature $\Sigma_{\mathsf{Nat}}$: we obtain a functor isomorphic to the expected $X \mapsto 1 + X$. △

**2.4 Example** (Signature: lists)**.** The datatype of lists, specified by

$$\begin{array}{l} \textbf{data}\ \mathsf{List}\ [A : \textsc{Set}] : \textsc{Set}\ \textbf{where} \\ \quad \mathsf{List}\ A \ni \mathsf{nil} \\ \qquad \mid \mathsf{cons}\ (a : A)(as : \mathsf{List}\ A) \end{array}$$

is strongly similar to the datatype of natural numbers. In fact, seen as a signature, the only difference stands in the definition of operations: a list offers either an operation $\mathsf{nil}$ (related to the $\mathsf{0}$ of natural numbers), or an $A$-indexed family of $\mathsf{cons}\ a$ operations (which can be

projected down to the operation suc of natural numbers). We obtain the following definition of operations:

$$\begin{array}{lll} \mathsf{Op}_{\mathsf{List}}\,(*:\mathbb{1}) & : & \textsc{Set} \\ \mathsf{Op}_{\mathsf{List}}\quad * & \mapsto & \mathbb{1}+A \end{array}$$

The remaining part of the signature – arity and typing – need only be updated to thread an $a:A$ instead of $*:\mathbb{1}$. Aside from that, the signatures are identical: we leave it to the reader to give the signature $\Sigma_{\mathsf{List}}:\mathsf{Sig}\;\mathbb{1}$ in full. In particular, the arities of both signatures are morally equivalent: as we shall see later, this witnesses an ornament.

△

**2.5 Example** (Signature: finite sets)**.** Beside lists, finite sets are another example of linearly-structured datatype. Let us recall their specification:

$$\begin{array}{l} \textbf{data}\;\mathsf{Fin}\,(n:\mathsf{Nat}):\textsc{Set}\;\textbf{where} \\ \quad \mathsf{Fin}\quad 0\quad \ni \\ \quad \mathsf{Fin}\,(\mathsf{suc}\;n)\ni \mathsf{f0} \\ \qquad\qquad\quad \mid \mathsf{fsuc}\;(k:\mathsf{Fin}\;n) \end{array}$$

The signature $\Sigma_{\mathsf{Fin}}$ of finite sets thus belongs to $\mathsf{Sig}\;\mathsf{Nat}$. Interestingly, at sort 0, no operation is available: there is no set of size zero. At sort $\mathsf{suc}\;n$, there are two operations, a new element f0 and the inclusion operation fsuc:

$$\begin{array}{lll} \mathsf{Op}_{\mathsf{Fin}}\,(n:\mathsf{Nat}) & : & \textsc{Set} \\ \mathsf{Op}_{\mathsf{Fin}}\quad 0 & \mapsto & \mathbb{0} \\ \mathsf{Op}_{\mathsf{Fin}}\;(\mathsf{suc}\;n) & \mapsto & \mathbb{1}+\mathbb{1} \end{array}$$

The arity of f0 is, unsurprisingly, $\mathbb{0}$. It is structurally related to the operation 0 of natural numbers. The arity of fsuc is $\mathbb{1}$, relating it to the operation suc of natural numbers:

$$\begin{array}{lllll} \mathsf{Ar}_{\mathsf{Fin}}\,(n:\mathsf{Nat})\,(op:\mathsf{Op}_{\mathsf{Fin}}\;n) & : & \textsc{Set} \\ \mathsf{Ar}_{\mathsf{Fin}}\;(\mathsf{suc}\;n) & (\mathsf{inj}_l\;*) & \mapsto & \mathbb{0} \\ \mathsf{Ar}_{\mathsf{Fin}}\;(\mathsf{suc}\;n) & (\mathsf{inj}_r\;*) & \mapsto & \mathbb{1} \end{array}$$

Finally, the typing follows our specification, stating that finite sets of size $\mathsf{suc}\;n$ include finite sets of size $n$:

$$\begin{array}{lllll} \mathsf{Ty}_{\mathsf{Fin}}\,(n:\mathsf{Nat})\,(op:\mathsf{Op}_{\mathsf{Fin}}\;n)\,(ar:\mathsf{Ar}_{\mathsf{Fin}}\;op) & : & \mathsf{Nat} \\ \mathsf{Ty}_{\mathsf{Fin}}\;(\mathsf{suc}\;n) & (\mathsf{inj}_r\;*) & * & \mapsto & n \end{array}$$

Putting it all together, we have defined the signature $\Sigma_{\mathsf{Fin}} \triangleq \mathsf{Op}_{\mathsf{Fin}} \lhd^{\mathsf{Ty}_{\mathsf{Fin}}} \mathsf{Ar}_{\mathsf{Fin}}$. Note that despite the variation in operations and typing, the arities of $\Sigma_{\mathsf{Fin}}$ are morally equivalent to the arities of $\Sigma_{\mathsf{Nat}}$, $\Sigma_{\mathsf{List}}$, or $\Sigma_{\mathsf{Vec}}$: it is either $\mathbb{0}$ or $\mathbb{1}$. △

**2.6 Example** (Signature: binary tree)**.** So far, the arities of our examples – vectors (Example 2.1), natural numbers (Example 2.3), lists (Example 2.4), and finite sets (Example 2.5) – were either $\mathbb{0}$ or $\mathbb{1}$. And indeed, these datatypes share the same linear structure.

Stepping away from linear structures, we now consider binary trees, specified by:

$$\begin{array}{l} \textbf{data}\;\mathsf{Tree}\,[A:\textsc{Set}]:\textsc{Set}\;\textbf{where} \\ \quad \mathsf{Tree}\;A\;\ni \mathsf{leaf} \\ \qquad\qquad \mid \mathsf{node}\;(lb:\mathsf{Tree}\;A)(a:A)(rb:\mathsf{Tree}\;A) \end{array}$$

The family of operations (written $\mathsf{Op}_{\mathsf{Tree}}$) is essentially the same as $\mathsf{Op}_{\mathsf{List}}$ for lists: an operation is either a leaf or an $A$-indexed family of node $a$ operations. As for lists, the indexing being trivial, the typing is trivial as well.

The structural difference between trees and, say, lists is revealed by the arities. If the arity of leaf is null (and could thus be mapped to nil), the arity of the constructor node $a$ is 2, *i.e.* $\mathbb{1}+\mathbb{1}$:

$$\begin{array}{lll} \mathsf{Ar}_{\mathsf{Tree}}\,(op : \mathsf{Op}_{\mathsf{Tree}}\,*) & : & \textsc{Set} \\ \mathsf{Ar}_{\mathsf{Tree}}\quad (\mathsf{inj}_l\,*) & \mapsto & \mathbb{0} \\ \mathsf{Ar}_{\mathsf{Tree}}\quad (\mathsf{inj}_r\,a) & \mapsto & \mathbb{1}+\mathbb{1} \end{array}$$

This makes the operation node $a$ structurally incompatible with either nil (arity null) or cons $a$ (arity one). Incompatible arities account for structurally incompatible datatypes. Again, we leave it to the reader to work out the full signature from these indications.

△

*2.7 Remark* (Variation on a theme). The above tree data-*structure* admits many data-*logics*. In particular, its type can be indexed to account for various *balancing* strategies. One example is the type of red-black trees (Example 2.8). Another example would be the type of AVL trees. We leave it to the reader to work out the signature of their favorite brand of balanced tree. Once again, these signatures will share a similar (binarily-branching) structure, specified by their arity, while the operations and typing vary.

◊

**2.8 Example** (Signature: red-black tree)**.** Red-black trees are indexed over an enumerated type of colors

$$\begin{array}{l} \textbf{data}\ \mathsf{Color} : \textsc{Set}\ \textbf{where} \\ \quad \mathsf{Color} \ni \mathsf{black} \\ \quad\quad \mid \mathsf{red} \end{array}$$

and natural numbers, which counts the depth in terms of black nodes.

Red-black trees are built from black leaves (at depth 0), red nodes whose children are necessarily black nodes of equal depth, and black nodes whose children can be of any color as long as their depth is equal:

$$\begin{array}{l} \textbf{data}\ \mathsf{RBT}\,[A : \textsc{Set}](c : \mathsf{Color})(n : \mathsf{Nat}) : \textsc{Set}\ \textbf{where} \\ \quad \mathsf{RBT}\,A\,\mathsf{black}\quad 0 \quad\ni \mathsf{leaf} \\ \quad \mathsf{RBT}\,A\,\mathsf{black}\,(\mathsf{suc}\,n) \ni \mathsf{nodeB}\ (c_l, c_r : \mathsf{Color})(lb : \mathsf{RBT}\,A\,c_l\,n)(a : A)(rb : \mathsf{RBT}\,A\,c_r\,n) \\ \quad \mathsf{RBT}\,A\quad \mathsf{red}\quad (\mathsf{suc}\,n) \ni \mathsf{nodeR}\ (lb : \mathsf{RBT}\,A\,\mathsf{black}\,(\mathsf{suc}\,n))(a : A)(rb : \mathsf{RBT}\,A\,\mathsf{black}\,(\mathsf{suc}\,n)) \end{array}$$

The corresponding signature is thus sorted by the product $\mathsf{Color} \times \mathsf{Nat}$. The choice of operations depends on the current color and depth:

- at depth zero, there is only one black leaf (and no red operation),
- at depth suc $n$, there is an $A$-indexed family of red nodes
- at depth suc $n$, there is an $A$-indexed family of black nodes for each possible pair of children's colors

In terms of signature, this translates to:

$$\begin{array}{lll}
\mathsf{Op}_{\mathsf{RBT}}\,(cn : \mathsf{Color} \times \mathsf{Nat}) & : & \mathrm{SET} \\
\mathsf{Op}_{\mathsf{RBT}}\quad (\mathsf{red}, 0) & \mapsto & \mathbb{0} \\
\mathsf{Op}_{\mathsf{RBT}}\quad (\mathsf{black}, 0) & \mapsto & \mathbb{1} \\
\mathsf{Op}_{\mathsf{RBT}}\quad (\mathsf{red}, \mathsf{suc}\ n) & \mapsto & A \\
\mathsf{Op}_{\mathsf{RBT}}\quad (\mathsf{black}, \mathsf{suc}\ n) & \mapsto & \mathsf{Color} \times \mathsf{Color} \times A
\end{array}$$

As for the binary tree (Example 2.6), the leaf has arity 0 while the nodes (red or black) have arity 2:

$$\begin{array}{llll}
\mathsf{Ar}_{\mathsf{RBT}}\,(cn : \mathsf{Color} \times \mathsf{Nat})\,(op : \mathsf{Op}_{\mathsf{RBT}}\ cn) & : & \mathrm{SET} \\
\mathsf{Ar}_{\mathsf{RBT}}\quad (\mathsf{black}, 0) \quad * & \mapsto & \mathbb{0} \\
\mathsf{Ar}_{\mathsf{RBT}}\quad (\mathsf{red}, \mathsf{suc}\ n) \quad a & \mapsto & \mathbb{1} + \mathbb{1} \\
\mathsf{Ar}_{\mathsf{RBT}}\quad (\mathsf{black}, \mathsf{suc}\ n) \quad (c_l, (c_r, a)) & \mapsto & \mathbb{1} + \mathbb{1}
\end{array}$$

Finally, the typing implements the balancing strategy by ensuring that children of red nodes are black nodes (of same depth) while children of black nodes are of preceding depth, following the color specified by the operation:

$$\begin{array}{llllll}
\mathsf{Ty}_{\mathsf{RBT}}\,(cn : \mathsf{Color} \times \mathsf{Nat})\,(op : \mathsf{Op}_{\mathsf{RBT}}\ cn)\,(ar : \mathsf{Ar}_{\mathsf{RBT}}\ cn\ op) & : & \mathsf{Color} \times \mathsf{Nat} \\
\mathsf{Ty}_{\mathsf{RBT}}\quad (\mathsf{red}, \mathsf{suc}\ n) \quad a \quad p & \mapsto & (\mathsf{black}, \mathsf{suc}\ n) \\
\mathsf{Ty}_{\mathsf{RBT}}\quad (\mathsf{black}, \mathsf{suc}\ n) \quad (c_l, (c_r, a)) \quad (\mathsf{inj}_l\ *) & \mapsto & (c_l, n) \\
\mathsf{Ty}_{\mathsf{RBT}}\quad (\mathsf{black}, \mathsf{suc}\ n) \quad (c_l, (c_r, a)) \quad (\mathsf{inj}_r\ *) & \mapsto & (c_r, n)
\end{array}$$

△

Many-sorted signatures provide a uniform framework to study indexing, abstracting over its many syntactic embodiment. In terms of expressive power, it models exactly the inductive families of extensional type theory. Indeed, our presentation is derived from the indexed variant of W-types (Martin-Löf, 1984) of Petersson and Synek (1989), using insights from Abbott's work on W-types (Abbott, 2003). GADTs can also be reduced to inductive families (Hamana & Fiore, 2011). Similarly, Atkey *et al.* (2012) have given a categorical model of (algebraic) refinement types, while the present author has further related the categorical model with polynomial functors (Dagand & McBride, 2013).

Because of its expressive power, one might be tempted to use the above formalism of signatures as a basis for implementation. While it certainly is a convenient mathematical object to deal with, it makes for a rather ineffective object from an intensional standpoint. The heart of the matter stands in its inherently extensional nature: to describe a first-order object, such as natural numbers (Example 2.3), we had to resort to a higher-order encoding. This representation is unfit for an intensional theory (Goguen & Luo, 1993; Dybjer, 1997).

## 3 Ornaments for domain-specific data-logics

Using the language of signatures, we shall now boil ornaments down to the notion of "structure-preserving transformations of datatypes". Our motivation is to express the logical enrichment of datatypes by domain-specific logics, whilst enforcing the stability of the underlying data-structure. In effect, we shall describe a meta-programming pattern by which one can bake domain-specific invariants into datatypes.

The crux of this section consists in pinpointing the informal concept of "structure" of a datatype to the formal notion of arity of a signature. Indeed, the arity of a signature corresponds *exactly* to the underlying data-*structure* of a datatype: it specifies its recursive skeleton, which we intend to keep invariant. For instance, it distinguishes the data-structure of $\mathsf{List}\,A$ and $\mathsf{Tree}\,A$: the former has arity 0 or 1, whilst the latter has arity 0 or 2. Both carry the same payload (elements of type $A$), but under distinct structures.

Having the same recursive structures does not forbid having different operations: binary trees (Example 2.6) and red-black trees (Example 2.8) share the same binarily-branching structure, even if the latter has more constructors. Invariance of structure also supports distinct typing disciplines: the datatype of lists (Example 2.4) and vectors (Example 2.1) share the same linear structure, even if vectors guarantee a stronger invariant concerning their length.

We will begin our study with a naive model (Section 3.1) constructed from the operational description of ornaments. This operational bias should help programmers come to grips with the abstract framework of signatures. To get to the essence of ornaments, we then refine this model to an equivalent but conceptually simpler presentation (Section 3.2) centered around the idea of structural invariance.

### *3.1 A naive model of ornaments*

Our first model attacks the problem through a constructive angle: from a signature, what information can be inserted that preserves the original structure? Concretely, from a signature $\Sigma = \mathsf{Op} \lhd^{\mathsf{Ty}} \mathsf{Ar}$ indexed by a set $I$, we shall give the necessary ingredients to build an (ornamented) signature[1] $\Sigma^\dagger = \mathsf{Op}^\dagger \lhd^{\mathsf{Ty}^\dagger} \mathsf{Ar}^\dagger$ indexed by a set $I^\dagger$, sharing the structure of $\Sigma$.

The first requirement arises from sorts: to express the fact that the $I^\dagger$-indices of $\Sigma^\dagger$ *refine* the $I$-indices of $\Sigma$, we require a (total) reindexing function

$$u : I^\dagger \to I$$

This function establishes a refinement in the sense that several indices $i^\dagger : I^\dagger$ can be mapped to the same index $i : I$: the $I^\dagger$-indexing is thus more discriminating than the $I$-indexing.

On operations, we allow the ornamented signature to *extend* the operation of $\Sigma$ through

$$\mathsf{extend} : (i^\dagger : I^\dagger) \to \mathsf{Op}\,(u\,i^\dagger) \to \textsc{Set}$$

which lets us describe the extended operations

$$\begin{array}{lll} \mathsf{Op}^\dagger & (i^\dagger : I^\dagger) & : \textsc{Set} \\ \mathsf{Op}^\dagger & i^\dagger & \mapsto (op : \mathsf{Op}\,(u\,i^\dagger)) \times \mathsf{extend}\,i^\dagger\,op \end{array}$$

Having clearly separated the extension from the original operations, we then trivially define the arity of the ornamented signature by

$$\begin{array}{lll} \mathsf{Ar}^\dagger & (op^\dagger : \mathsf{Op}^\dagger\,i^\dagger) & : \textsc{Set} \\ \mathsf{Ar}^\dagger & op^\dagger & \mapsto \mathsf{Ar}\,(\pi_0\,op^\dagger) \end{array}$$

[1] To identify the ornamented signature as "decorated", we mark its components with a superscript $-^\dagger$. This is nothing but an informal, notational convention.

guaranteeing, by construction, that $\mathsf{Ar}^\dagger$ is equal to $\mathsf{Ar}$, *i.e.* we have:

$$\forall\, op^\dagger : \mathsf{Op}^\dagger\, i^\dagger .\, \mathsf{Ar}\,(\pi_0\; op^\dagger) = \mathsf{Ar}^\dagger\; op^\dagger$$

Finally, we can refine the sorts of the recursive arguments, provided that the refined sort is related to the underlying sort through $u$. The refinement is given by a function

$$\mathsf{refine} : (i^\dagger : I^\dagger)(e : \mathsf{extend}\; i^\dagger\; op) \to \mathsf{Ar}\; op \to I^\dagger$$

which is subject to the coherence condition:

$$\mathsf{coh} : \forall i^\dagger : I^\dagger .\, \forall e : \mathsf{extend}\; i^\dagger\; op .\, \forall ar : \mathsf{Ar}\; op .\, u\,(\mathsf{refine}\; i^\dagger\; e\; ar) = \mathsf{Ty}\; ar$$

Remark that refine corresponds exactly to the typing discipline of the signature $\Sigma^\dagger$, *i.e.*

$$\begin{array}{llll} \mathsf{Ty}^\dagger\; (i^\dagger : I^\dagger)\; (op^\dagger : \mathsf{Op}^\dagger\; i^\dagger)\; (ar^\dagger : \mathsf{Ar}^\dagger\; op^\dagger) & : & I^\dagger \\ \mathsf{Ty}^\dagger \quad i^\dagger \qquad (op, e) \qquad ar^\dagger & \mapsto & \mathsf{refine}\; i^\dagger\; e\; ar^\dagger \end{array}$$

An ornament can thus be seen as the data of a refinement function $u$, an extension extend, a typing refinement refine and its associated coherence condition coh:

$$\mathsf{COrn}\,(\Sigma : \mathsf{Sig}\; I)\,(u : I^\dagger \to I) : \textsc{Set}_1$$

$$\mathsf{COrn}\,(\mathsf{Op} \lhd^{\mathsf{Ty}} \mathsf{Ar})\; u \mapsto \left\{ \begin{array}{ll} \mathsf{extend} : & (i^\dagger : I^\dagger) \to \mathsf{Op}\,(u\, i^\dagger) \to \textsc{Set} \\ \mathsf{refine} : & (i^\dagger : I^\dagger)(e : \mathsf{extend}\; i^\dagger\; op) \to \mathsf{Ar}\; op \to I^\dagger \\ \mathsf{coh} : & \forall i^\dagger : I^\dagger .\, \forall e : \mathsf{extend}\; i^\dagger\; op .\, \forall ar : \mathsf{Ar}\; op . \\ & \quad u\,(\mathsf{refine}\; i^\dagger\; e\; ar) = \mathsf{Ty}\; ar \end{array} \right.$$

As for signatures, we write the tuple $(\mathsf{extend}, \mathsf{refine}, \mathsf{coh})$ with the more lightweight notation $\mathsf{extend} \blacktriangleleft^{\mathsf{refine}}$, leaving aside the computationally-irrelevant coherence proof coh.

In parallel, we have given their interpretation as an ornamented signature $\Sigma^\dagger$. Ornaments thus interpret to signatures:

$$\begin{array}{lll} [\![(\tau : \mathsf{COrn}\; \mathsf{Op} \lhd^{\mathsf{Ty}} \mathsf{Ar})]\!]_{\mathsf{COrn}} & : & \mathsf{Sig}\; I^\dagger \\ [\![\mathsf{extend} \blacktriangleleft^{\mathsf{refine}}]\!]_{\mathsf{COrn}} & \mapsto & (\lambda i^\dagger .\, (op : \mathsf{Op}\,(u\, i^\dagger)) \times \mathsf{extend}\; i^\dagger\; op) \lhd^{\mathsf{refine}} (\lambda (op, e) .\, \mathsf{Ar}\; op) \end{array}$$

**3.1 Example** (Ornamenting natural numbers to lists)**.** Let $A : \textsc{Set}$. The ornamentation of natural numbers (Example 2.3) to describe lists (Example 2.4) is an example of a purely extensive ornament: no refinement is introduced on the indices since they are both trivial. We only extend the suc operation of natural numbers by asking for an inhabitant of $A$:

$$\begin{array}{llll} \mathsf{extend}_{\mathsf{List}}\; (* : \mathbb{1})\; (op : \mathsf{Op}_{\mathsf{Nat}}\; *) & : & \textsc{Set} \\ \mathsf{extend}_{\mathsf{List}} \quad * \qquad (\mathsf{inj}_l\; *) & \mapsto & \mathbb{1} \\ \mathsf{extend}_{\mathsf{List}} \quad * \qquad (\mathsf{inj}_r\; *) & \mapsto & A \end{array}$$

The typing discipline being the obvious one

$$\begin{array}{llll} \mathsf{refine}_{\mathsf{List}}\; (* : \mathbb{1})\; (e : \mathsf{extend}_{\mathsf{List}}\; *\; op)\; (ar : \mathsf{Ar}_{\mathsf{Nat}}\; op) & : & \mathbb{1} \\ \mathsf{refine}_{\mathsf{List}} \quad * \qquad e \qquad ar & \mapsto & * \end{array}$$

and the coherence condition trivial.

We easily check that the signature thus described (after interpretation with $[\![-]\!]_{\mathsf{COrn}}$) corresponds to the signature of lists, up to type isomorphisms:

$$\begin{aligned}
&[\![[\![\mathsf{extend}_{\mathsf{List}} \blacktriangleleft^{\mathsf{refine}_{\mathsf{List}}} ]\!]_{\mathsf{COrn}}]\!]\, X \\
&\quad = [\![(\lambda *.\, (op : \mathsf{Op}_{\mathsf{Nat}}\, *) \times \mathsf{extend}_{\mathsf{List}}\, *\, op) \lhd^{\mathsf{refine}_{\mathsf{List}}} (\lambda (op, e).\, \mathsf{Ar}_{\mathsf{Nat}}\, op)]\!]\, X \\
&\quad = \lambda *.\, (op^{\dagger} : (op : \mathsf{Op}_{\mathsf{Nat}}\, *) \times \mathsf{extend}_{\mathsf{List}}\, *\, op) \times ((ar : \mathsf{Ar}_{\mathsf{Nat}}\, (\pi_0\, op^{\dagger})) \to X\, *) \\
&\quad = \lambda *.\, (op^{\dagger} : (op : \mathbb{1} + \mathbb{1}) \times \mathsf{extend}_{\mathsf{List}}\, *\, op) \times ((ar : \mathsf{Ar}_{\mathsf{Nat}}\, (\pi_0\, op^{\dagger})) \to X\, *) \\
&\quad \cong \lambda *.\, (op : \mathbb{1} + A) \times ((ar : \mathsf{Ar}_{\mathsf{List}}\, op) \to X\, *) \\
&\quad = \lambda *.\, (op : \mathsf{Op}_{\mathsf{List}}) \times ((ar : \mathsf{Ar}_{\mathsf{List}}\, op) \to X\, *)
\end{aligned}$$

△

**3.2 Example** (Ornamenting natural numbers to finite sets)**.** Natural numbers can also be ornamented to finite sets (Example 2.5). This ornament both extends the operations of natural numbers and refines their type to $\mathsf{Nat}$.

The extension consists in enforcing that the input index $n : \mathsf{Nat}$ is strictly-positive

$$\begin{array}{llll}
\mathsf{extend}_{\mathsf{Fin}} & (n : \mathsf{Nat}) & (op : \mathsf{Op}_{\mathsf{Nat}}\, *) & : \mathsf{SET} \\
\mathsf{extend}_{\mathsf{Fin}} & n & op & \mapsto (n' : \mathsf{Nat}) \times n = \mathsf{suc}\, n'
\end{array}$$

since no operation is available at index $\mathsf{0}$.

In the case of the $\mathsf{fsuc}$ constructor, we must then specify the indexing discipline by asking the recursive argument to have sort $n - 1$. Knowing that $n = \mathsf{suc}\, n'$, we define the sort by:

$$\begin{array}{llllll}
\mathsf{refine}_{\mathsf{Fin}} & (n : \mathsf{Nat}) & (op : \mathsf{Op}_{\mathsf{Nat}}\, *) & (e : \mathsf{extend}_{\mathsf{Fin}}\, n\, op) & (ar : \mathsf{Ar}_{\mathsf{Nat}}\, op) & : \mathsf{Nat} \\
\mathsf{refine}_{\mathsf{Fin}} & n & (\mathsf{inj}_r\, *) & (n', q) & * & \mapsto n'
\end{array}$$

Again, we easily check that the signature thus described (after interpretation with $[\![-]\!]_{\mathsf{COrn}}$) corresponds to the signature of finite sets, up to type isomorphisms.

△

**3.3 Example.** Other examples include various ornamentations of binary trees, either specifying where data are stored in the structure (at the leaves, at the nodes, or both) but also various balancing strategies (AVL, red-black, . . . ).

△

### *3.2 Ornaments as Cartesian morphisms*

The model introduced in the previous Section suffers from a very operational bias: it focuses on describing *how* to build an ornamented signature. The reason *why* an object ornaments another is not yet clear. This bias leads to a definition focused on signatures, rather than a particular (structural) relation between them.

We would like to formalize our intuition that a signature ornaments another if they share a similar recursive structure. We are going to massage this initial model to account for such an invariant. Doing so, we shall obtain an algebraic characterization of ornaments, more natural and suited to abstract reasoning.

In Section 2, we gave an interpretation of signatures in terms of functors. One is then tempted to look for a notion of morphism of signature accounting for the natural transformations between these functors (Abbott, 2003; Morris, 2007). In our setting, we are

interested in a more specific notion, where the arities of both signatures are equal. This amounts to having a *Cartesian morphism* between the signatures (Abbott *et al.*, 2005; Gambino & Kock, 2013). Let us recall the definition of Cartesian morphisms and their interpretation as Cartesian natural transformations.

**3.4 Definition** (Cartesian morphism of signature (Gambino & Kock, 2013))**.** Let $\Sigma^\dagger \triangleq \mathsf{Op}^\dagger \lhd^{\mathsf{Ty}^\dagger} \mathsf{Ar}^\dagger : \mathsf{Sig}\ I^\dagger$ and $\Sigma \triangleq \mathsf{Op} \lhd^{\mathsf{Ty}} \mathsf{Ar} : \mathsf{Sig}\ I$ be two signatures, indexed respectively by $I^\dagger$ and $I$. Let $u : I^\dagger \to I$ be the function mapping the indices of the former to the latter.

A Cartesian morphism from $\Sigma^\dagger$ to $\Sigma$ is given by a morphism on operations $\sigma$ that translates the $\Sigma^\dagger$-operations into $\Sigma$-operations

$$\sigma : \mathsf{Op}^\dagger\ i^\dagger \to \mathsf{Op}\ (u\ i^\dagger)$$

together with a proof $\rho$, stating that the arities of both signatures are equal through $\sigma$

$$\rho : \forall\, op^\dagger : \mathsf{Op}^\dagger\ i^\dagger .\ \mathsf{Ar}\ (\sigma\ op^\dagger) = \mathsf{Ar}^\dagger\ op^\dagger$$

and a proof coh stating that the indexing of $\Sigma^\dagger$ is coherent with respect to the indexing of $\Sigma$ through $u$:

$$\mathsf{coh} : \forall\, op^\dagger : \mathsf{Op}^\dagger\ i^\dagger .\ \forall\, ar : \mathsf{Ar}\ (\sigma\ op^\dagger) .\ u\ (\mathsf{Ty}^\dagger\ ar) = \mathsf{Ty}\ ar$$

For conciseness, we write a Cartesian morphism $(\sigma, \rho, \mathsf{coh})$ simply as $\sigma$, thus eluding the two proofs, whose computational content is void. The class of Cartesian morphisms from $\Sigma^\dagger$ to $\Sigma$ along a reindexing $u$ is written $\Sigma^\dagger \overset{u}{\Longrightarrow}^c \Sigma$.

▽

**3.5 Definition** (Interpretation of a Cartesian morphism)**.** Following this intuition, the interpretation of Cartesian morphism defines a natural transformation from $[\![\Sigma^\dagger]\!]\ (X \circ u)\ i^\dagger$ to $[\![\Sigma]\!]\ X\ (u\ i^\dagger)$:

$$\begin{array}{l} [\![(\sigma : \Sigma^\dagger \overset{u}{\Longrightarrow}^c \Sigma)]\!]\ (xs : [\![\Sigma^\dagger]\!]\ (X \circ u)\ i^\dagger) : [\![\Sigma]\!]\ X\ (u\ i^\dagger) \\ [\![\sigma]\!]\ (op^\dagger, Xs) \mapsto (\sigma\ op^\dagger, Xs) \end{array}$$

▽

*3.6 Remark.* The equality on arities and the coherence condition are implicitly used in the definition of the interpretation to match up the type of the recursive arguments given by $Xs$. Indeed, the argument $Xs$ has type $(ar^\dagger : \mathsf{Ar}^\dagger\ op^\dagger) \to (X \circ u)\ (\mathsf{Ty}^\dagger\ ar^\dagger)$. By $\rho$, we can transport its argument to $\mathsf{Ar}\ (\sigma\ op^\dagger)$ while, by coh, we can transport its result to $X\ (\mathsf{Ty}\ ar)$. We thus obtain a function $(ar : \mathsf{Ar}\ (\sigma\ op^\dagger)) \to X\ (\mathsf{Ty}\ ar)$, mapping the $\Sigma$-arity to recursive arguments in $X$.

◊

*3.7 Remark* (Inverse image construction). In type theory, a function $f : A \to B$ can be seen as a predicate over $B$: the set $A$ is then understood as a collection of properties of the $B$-elements and the function $f$ mapping the $A$-properties to the $B$-elements. The inverse image $f^{-1} : B \to \textsc{Set}$ collects the $A$-properties associated to each $B$-elements. It can be defined with an inductive family

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

or, equivalently, with a Σ-type:

$$\begin{array}{ll}(f : A \to B)^{-1}\,(b : B) & : \ \textsc{Set} \\ f^{-1}\,b & \mapsto (a : A) \times f\,a = b\end{array}$$

Conversely, from a predicate $P : B \to \textsc{Set}$, we can build a function $\pi_0 : (b : B) \times P\,b \to B$. These two transformations are inverse to each other: this is the type-theoretic incarnation of the equivalence between the category $\textsc{Set}^B$ of predicates over $B$ and the slice category $\textsc{Set}/B$.

◊

**3.8 Example** (Cartesian morphism from natural numbers to lists)**.** The equivalence between the naive model of ornaments and the Cartesian morphism is nothing but an instance of that isomorphism. To gain some intuition on the transformation, let us consider the ornament of natural numbers to lists (Example 3.1). For $A : \textsc{Set}$, we have specified its extension as:

$$\begin{array}{lll}\mathsf{extend}_{\mathsf{List}}\,(op : \mathbb{1} + \mathbb{1}) & : & \textsc{Set} \\ \mathsf{extend}_{\mathsf{List}}\quad (\mathsf{inj}_l\ *) & \mapsto & \mathbb{1} \\ \mathsf{extend}_{\mathsf{List}}\quad (\mathsf{inj}_r\ *) & \mapsto & A\end{array}$$

In fact, $\mathsf{extend}_{\mathsf{List}} : \mathbb{1} + \mathbb{1} \to \textsc{Set}$ can be understood as the inverse image of the (equivalent) function

$$\begin{array}{lll}\sigma_{\mathsf{List}}\,(op : \mathbb{1} + A) & : & \mathbb{1} + \mathbb{1} \\ \sigma_{\mathsf{List}}\quad (\mathsf{inj}_l\ *) & \mapsto & \mathsf{inj}_l\ * \\ \sigma_{\mathsf{List}}\quad (\mathsf{inj}_r\ a) & \mapsto & \mathsf{inj}_r\ *\end{array}$$

that is:

$$\mathsf{extend}_{\mathsf{List}} \cong {\sigma_{\mathsf{List}}}^{-1}$$

We are then left to check that the arity of natural numbers matches the arity of lists through $\sigma_{\mathsf{List}}$ and that the typing (which is trivial) is coherent. We have defined a Cartesian morphism from natural numbers to lists.

△

This construction generalizes to any ornament. Provided a Cartesian morphism $(\sigma, \rho, \mathsf{coh})$ from $\Sigma^\dagger$ to $\Sigma$, we obtain an ornament $\sigma^{-1} \blacktriangleleft^{\mathsf{Ty}^\dagger}$ by taking $\mathsf{extend} \triangleq \sigma^{-1}$, $\mathsf{refine} \triangleq \mathsf{Ty}^\dagger$, and $\mathsf{coh} \triangleq \mathsf{coh}$. Conversely, provided an ornament $\mathsf{extend}\blacktriangleleft^{\mathsf{refine}}$ of a signature $\Sigma$, we obtain a Cartesian morphism from $\Sigma^\dagger \triangleq [\![\mathsf{extend}\blacktriangleleft^{\mathsf{refine}}]\!]_{\mathsf{COrn}}$ to $\Sigma$ by taking $\sigma \triangleq \pi_0$. The arity and coherence of both signatures is respected, by construction.

Whilst the naive model of ornaments relies on an operational intuition of ornaments (as "introducing more information"), the Cartesian model puts the emphasis on the structure-preserving nature of the transformation, through the condition on arities $\rho$. The notion of extension is carried by the morphism on operations $\sigma$, which describes how the extra-information of the ornamented signature is lost by going to the target signature. The notion of typing refinement is captured, as for the naive model, by the reindexing function $u$ and the coherence condition $\mathsf{coh}$.

This presentation gives a simple criterion to decide whether a datatype is the ornament of another: we must be able to relate their operations in such a way that their arity matches. Hence, all the linear structures – such as List, Fin, Vec, *etc.*– can be seen as ornaments of

natural numbers: operations of arity null can be mapped to the operation 0, while operations of arity 1 can be mapped to the operation suc.

A non-example of ornament are lists and binary trees: we are unable to map the operations cons $a$ (of arity 1) in the signature of lists to an operation of arity 1 in the signature of binary trees. Conversely, there is no operation of arity 2 in the signature of lists to which we could map the operation node $a$ of the signature of binary trees.

Another non-example of ornament are AVL trees and red-black trees. While both signatures share a binarily-branching structure, their indexing disciplines are incompatible: the former is not a refinement of the latter, nor conversely.

**Discussion:** These two models of ornaments – naive or Cartesian – are complementary. The naive model is subject to an operational bias: it is in fact at the heart of the original presentation of ornaments by McBride (2011). In that original presentation, Cartesian morphisms appear as a side-result of the construction of the *ornamental algebra* (Section 4), the algebra projecting the ornamented type to its underlying type. As a result, this presentation is better suited to give concrete examples. Conversely, the categorical model of Dagand and McBride (2013) revolves around the Cartesian presentation, which is mathematically more convenient but operationally imprecise. The equivalence between the universe-based presentation of ornaments and the Cartesian one was formally established in this latter paper.

## 4 A Calculus of Data-structure

By characterizing ornaments as structure-preserving transformations, the Cartesian presentation gives us a semantic criterion for identifying ornaments. Being expressed in terms of morphisms, it also offers a compositional toolbox. It is in fact a genuine calculus of data-structure that is open to us. To further stir up the reader's curiosity, let us consider a few concrete examples. We voluntarily adopt a *descriptive* approach, using our fresh understanding of ornaments to identify ornamental patterns seen in the wild. For each example we provide, one can easily belabor the definition of the Cartesian morphism witnessing the existence of an ornament. We shall leave aside their generalization and theoretical justification, which can be found elsewhere (Dagand & McBride, 2013) in a categorical setting: finding a satisfactory *prescriptive* presentation of ornaments is still subject of active research, a question to which we shall come back in Section 5.

**Ornamental algebra:** We can always recover the raw datatype underpinning an ornamented type. For example, the Cartesian morphism mapping lists to naturals (Example 3.8) yields a natural transformation

$$[\![\sigma_{\mathsf{List}}]\!] : \forall X.\, [\![\Sigma_{\mathsf{List}}]\!]\, X * \to [\![\Sigma_{\mathsf{Nat}}]\!]\, X *$$

that we can post-compose with the initial algebra of lists at Nat, thus obtaining the so-called *ornamental algebra*

$$\begin{aligned} \mathsf{forget}\,\sigma_{\mathsf{List}} &: [\![\Sigma_{\mathsf{List}}]\!]\,\mathsf{Nat} * \to \mathsf{Nat} \\ \mathsf{forget}\,\sigma_{\mathsf{List}} &\mapsto \mu \circ ([\![\sigma_{\mathsf{List}}]\!]\,\mathsf{Nat}) \end{aligned}$$

In effect, this algebra computes the length of a list, *i.e.* it forgets the extra data inserted by the ornament.

**Vertical and horizontal composition:** Like the natural transformations they represent, Cartesian morphisms support both vertical and horizontal composition. Vertical composition allows us to *fuse* two successive ornamentation of a datatype into a single one. For example, full binary trees (*i.e.*, binary trees storing data at the nodes and leaves)

```
data Full [A : Set] : Set where
  Full A ∋ leaf (a : A)
         | node (lb : Full A)(a : A)(rb : Full A)
```

are an ornament (through an extension) of node binary trees (*i.e.*, binary trees storing data at the nodes, as in Example 2.6) while perfect binary trees (*i.e.*, trees for which all the leaves have the same depth)

```
data Perfect [A : Set](n : Nat) : Set where
  Perfect A    0     ∋ leaf (a : A)
  Perfect A (suc n) ∋ node (lb : Perfect A n)(a : A)(rb : Perfect A n)
```

are an ornament (through a refinement of indices) of full binary trees. By vertical composition, we deduce that perfect binary trees ornament node binary trees.

Horizontal composition lets us combine two ornaments (and, thus, four signatures) into a single one relating the composition of the underlying signatures. For example, we have that vectors (Example 2.1) ornaments (through a refinement of indices) lists (Example 2.4) by enforcing the length invariant. We also have that the the Nat-indexed functor $F^\dagger \triangleq X_n \mapsto X_n \times X_{\mathsf{suc}\, n} + X_{\mathsf{suc}\, n} \times X_n$ ornaments the $\mathbb{1}$-indexed functor $F \triangleq X \mapsto X * \times X *$. By horizontal composition, we deduce that balanced binary trees

```
data BST [A : Set](n : Nat) : Set where
  BST A    0     ∋ leaf
  BST A (suc n) ∋ nodeL (lb : BST A (suc n))(a : A)(rb : BST A n)
  BST A (suc n) ∋ nodeR (lb : BST A n)(a : A)(rb : BST A (suc n))
```

ornament node binary trees by noticing that the signature of BST amounts to the composition of the signature of vectors and the functor $F^\dagger$ while the signature of Tree amounts to the composition of the signature of lists and the functor $F$.

**Pullback:** Given two ornaments of the same signature, we can combine both ornamentations into a single signature. This amounts to taking the *fibred product* of the two Cartesian morphisms. For example, bounded lists

```
data BList [A : Set](n : Nat) : Set where
  BList A    0     ∋
  BList A (suc n) ∋ nil
  BList A (suc n) ∋ cons (a : A)(vs : BList A n)
```

ornament natural numbers by combining the ornamentation of natural numbers into lists (Example 3.1) *and* the ornamentation of natural numbers into finite sets (Example 3.2). We thus obtain a type of lists indexed by $n : \mathsf{Nat}$ and whose length is *at most* $n$ (unlike vectors,

whose length is *precisely* $n$). A typical use-case for bounded lists is the dependently-typed filter on vectors, whose type would be $\mathsf{Vec}\,A\,n \to (A \to 2) \to \mathsf{BList}\,A\,n$

**Algebraic ornament:** Provided a recursive function (presented as a catamorphism) over an inductive type, the algebraic ornament indexes the datatype by the result of the function. The canonical example of an algebraic ornament is due to McBride (2011): consider a datatype of arithmetic expressions

$$\begin{array}{l}\textbf{data}\ \mathsf{Expr} : \textsc{Set}\ \textbf{where}\\ \quad \mathsf{Expr} \ni \mathsf{const}\ (n : \mathsf{Nat})\\ \quad \mathsf{Expr} \ni \mathsf{add}\ (d, e : \mathsf{Expr})\end{array}$$

for which we have defined its denotation by means of a catamorphism

$$\begin{array}{lcl}\mathsf{eval}\ (e : \mathsf{Expr}) & : & \mathsf{Nat}\\ \mathsf{eval}\quad e & \mapsto & (\!|\alpha_{\mathsf{eval}}\ e|\!)\end{array}\qquad\qquad \begin{array}{lcl}\alpha_{\mathsf{eval}}\ (xs : [\![\Sigma_{\mathsf{Expr}}]\!]\ \mathsf{Nat}) & : & \mathsf{Nat}\\ \alpha_{\mathsf{eval}}\quad (\text{'const}\ n) & \mapsto & n\\ \alpha_{\mathsf{eval}}\quad (\text{'add}\ m\ n) & \mapsto & m+n\end{array}$$

where, by convention, we write 'constr the constructor of the signature functor corresponding to the datatype constructor constr.

The algebraic ornament enables us to fuse the algebra of the catamorphism as a static typing discipline for the datatype, thus yielding the type of expressions indexed by their semantics:

$$\begin{array}{l}\textbf{data}\ \mathsf{SemExpr}\ (k : \mathsf{Nat}) : \textsc{Set}\ \textbf{where}\\ \quad \mathsf{SemExpr}\ k \ni \mathsf{const}\ (n : \mathsf{Nat})(q : k = n)\\ \quad \mathsf{SemExpr}\ k \ni \mathsf{add}\ (m\,n : \mathsf{Nat})(d : \mathsf{SemExpr}\ m)(e : \mathsf{SemExpr}\ n)(q : k = m+n)\end{array}$$

Algebraic ornaments are characterized by a *coherence* property, stating that the ornamented type represents exactly the elements of the base type whose denotation is given by the index. For example, the coherence of algebraic ornaments establishes the following isomorphism between Expr and SemExpr for any index $k : \mathsf{Nat}$:

$$\mathsf{SemExpr}\ k \cong (e : \mathsf{Expr}) \times \mathsf{eval}\ e = k$$

**Relational ornament:** The existence of catamorphisms arise from the fact that polynomial functors admit an initial algebra in the category Set, where morphisms are total functions. In fact, this result can be generalized to the category *Rel*, where morphisms are binary relations (Fumex, 2012). As demonstrated by Ko and Gibbons (2013), this observation translates into *relational ornaments*, by which one bakes recursively-defined predicates into a typing discipline.

For example, we can recursively define the relation $m < n$ through a catamorphism:

$$\begin{array}{lcl}<\ (m : \mathsf{Nat})\ (n : \mathsf{Nat}) & : & \textsc{Set}\\ <\quad m \qquad n & \mapsto & (\!|\alpha_{<}|\!)\ m\ n\end{array}\qquad\qquad \begin{array}{lcl}\alpha_{<}\ (xs : [\![\Sigma_{\mathsf{Nat}}]\!]\ \mathsf{Nat})\ (n : \mathsf{Nat}) & : & \textsc{Set}\\ \alpha_{<}\quad \text{'0} \qquad n & \mapsto & (k : \mathsf{Nat}) \times n = \mathsf{suc}\ k\\ \alpha_{<}\quad (\text{'suc}\ m) \qquad n & \mapsto & n = \mathsf{suc}\ m\end{array}$$

Thanks to relational ornaments, we can integrate this relation as a typing discipline over natural numbers: this is precisely defining the type of finite sets (Example 2.5), with the added benefit that we obtain, through the coherence property, an isomorphism for any index

$n : \mathsf{Nat}$ between the inductive family and a subset of the natural numbers

$$\mathsf{Fin}\ n \cong (m : \mathsf{Nat}) \times m < n$$

Ornaments being compositional, we can combine algebraic and relational ornaments in a straightforward manner. For example, let us consider the type of well-sized stack expressions (Chlipala, 2013)

$$\begin{array}{l}\textbf{data}\ \mathsf{InstrIO}\ (i : \mathsf{Nat})(o : \mathsf{Nat}) : \textsc{Set}\ \textbf{where}\\ \quad \mathsf{InstrIO}\quad i\quad (\mathsf{suc}\ i) \ni \mathsf{CONST}\ (k : \mathsf{Nat})\\ \quad \mathsf{InstrIO}\ (\mathsf{suc}\ (\mathsf{suc}\ i))\quad i\quad \ni \mathsf{PLUS}\end{array}$$

whose index $i$ describes the expected input stack size while the index $o$ describes the resulting output stack size. We notice that the output index $o$ can be recursively computed from the input index and the constructors. Therefore, we can see $\mathsf{InstrIO}$ as the ornament of the type

$$\begin{array}{l}\textbf{data}\ \mathsf{InstrI}\ (i : \mathsf{Nat}) : \textsc{Set}\ \textbf{where}\\ \quad \mathsf{InstrI}\quad i\quad \ni \mathsf{CONST}\ (k : \mathsf{Nat})\\ \quad \mathsf{InstrI}\ (\mathsf{suc}\ (\mathsf{suc}\ i)) \ni \mathsf{PLUS}\end{array}$$

by the algebra

$$\begin{array}{l}\mathsf{size}\ (c : \mathsf{InstrI}\ i)\ :\ \mathsf{Nat}\\ \mathsf{size}\quad c\quad \mapsto (\!|\alpha_{\mathsf{size}}|\!)\ c\end{array}\qquad \begin{array}{lllll}\alpha_{\mathsf{size}} & (i : \mathsf{Nat}) & (c : [\![\Sigma_{\mathsf{InstrI}}]\!]\ \mathsf{Nat}) & : & \mathsf{Nat}\\ \alpha_{\mathsf{size}} & i & (\text{'}\mathsf{CONST}\ k) & \mapsto & \mathsf{suc}\ i\\ \alpha_{\mathsf{size}} & (\mathsf{suc}\ (\mathsf{suc}\ i)) & \text{'}\mathsf{PLUS} & \mapsto & \mathsf{suc}\ i\end{array}$$

However, the type $\mathsf{InstrI}$ itself can be seen as the ornament of the non-indexed datatype

$$\begin{array}{l}\textbf{data}\ \mathsf{Instr} : \textsc{Set}\ \textbf{where}\\ \quad \mathsf{Instr} \ni \mathsf{CONST}\ (k : \mathsf{Nat})\\ \qquad\ \ |\ \mathsf{PLUS}\end{array}$$

by the relational algebra

$$\begin{array}{l}\mathsf{valid}\ (c : \mathsf{Instr})\ (i : \mathsf{Nat})\ :\ \textsc{Set}\\ \mathsf{valid}\quad c\quad i\quad \mapsto (\!|\alpha_{\mathsf{valid}}|\!)\ c\ i\end{array}\qquad \begin{array}{llll}\alpha_{\mathsf{valid}}\ (c : [\![\Sigma_{\mathsf{InstrI}}]\!]\ \mathsf{Nat}) & (i : \mathsf{Nat}) & : & \mathsf{Nat}\\ \alpha_{\mathsf{valid}}\quad (\text{'}\mathsf{CONST}\ k) & i & \mapsto & \mathbb{1}\\ \alpha_{\mathsf{valid}}\quad \text{'}\mathsf{PLUS} & 0 & \mapsto & \mathbb{0}\\ \alpha_{\mathsf{valid}}\quad \text{'}\mathsf{PLUS} & (\mathsf{suc}\ 0) & \mapsto & \mathbb{0}\\ \alpha_{\mathsf{valid}}\quad \text{'}\mathsf{PLUS} & (\mathsf{suc}\ (\mathsf{suc}\ i)) & \mapsto & \mathbb{1}\end{array}$$

which ensures that the input index for the $\mathsf{PLUS}$ constructor is necessarily of the form $\mathsf{suc}\ (\mathsf{suc}\ i)$. By composition of ornament, we deduce that the precisely-indexed $\mathsf{InstrIO}$ type ornaments the non-indexed $\mathsf{Instr}$ datatype.

**Reornament:** Every ornament induces an ornamental algebra over the ornamented type while every algebra induces an algebraic ornament. By combining both constructions, we obtain the *algebraic ornament by the ornamental algebra* (McBride, 2011), or *reornament* (Dagand & McBride, 2012) for short, which consists in an ornamented datatype indexed by its underlying elements. The canonical example of reornament is the type of vectors (Example 2.1), which can be seen as the algebraic ornament of lists (seen as ornament of natural numbers, as in Example 3.1) by the ornamental algebra $[\![\sigma_{\mathsf{List}}]\!]$ that computes the length of a list.

Interestingly, the ornament from which we compute the reornament can be obtained by any of the compositional means available to us. For example, we observe that the inductive comparison predicate

```
data (m : Nat) < (n : Nat) : SET where
  m < (suc n) ∋ lt-z-s
  m < (suc n) ∋ lt-s-s (q : m < n)
```

corresponds to the reornament of the type of finite sets Fin $n$, seen as a (relational) ornament of Nat: the index $m$ of the inductive predicate $m<n$ corresponds precisely to the integer underpinning the inhabitants of the type Fin $n$.

**Reindexing:** Ornaments can also solely refine the indexing strategy of an inductive family. Given an inductive family indexed by a type $I$ and a function refining its index (*i.e.*, a function of type $I^\dagger \to I$), we obtain an inductive family indexed by $I^\dagger$ through *reindexing*. For example, we might be interested in the subset of arithmetic expressions whose denotation is a strictly-positive number

```
data SemExpr⁺ (k : Nat*) : SET where
  SemExpr⁺ k ∋ const (n : Nat)(q : toNat k = n)
               | add  (m n : Nat*)(d : SemExpr⁺ m)(e : SemExpr⁺ n)
                      (q : toNat k = toNat m + toNat n)
```

where toNat : Nat* → Nat is the embedding function from the strictly-positive subset of natural numbers to natural numbers. SemExpr⁺ is in fact merely a reindexing of the inductive family SemExpr along the function toNat.

**Derivative:** Whilst we have focused our presentation on the combinatorics of ornaments, some operations on datatypes also carry on to operations on ornaments. For example, the derivative of a data-type – from which one obtains the Zipper (Huet, 1997) – behaves *functorially* over ornaments (Dagand & McBride, 2013): if a datatype ornaments another, then its derivative ornaments the derivative of the latter. For example, we have that red-black trees (Example 2.8) ornament node binary trees (Example 2.6). We therefore deduce that the derivative of red-black tree

```
data ∂RBT [A : SET][RBT : Color → SET] : SET where
  ∂RBT black red   ∋ ∂nodeRL (lb : RBT black)(a : A)
  ∂RBT black red   ∋ ∂nodeRR (a : A)(rb : RBT black)
  ∂RBT   c   black ∋ ∂nodeBL (lb : RBT c)(a : A)
  ∂RBT   c   black ∋ ∂nodeBR (a : A)(rb : RBT c)
```

ornaments the derivative of binary tree

```
data ∂Tree [A : SET][Tree : SET] : SET where
  ∂Tree ∋ ∂nodeL (lb : Tree)(a : A)
         | ∂nodeR (a : A)(rb : Tree)
```

**Numerical representations** A mostly untapped source of ornaments is the realm of numerical representations (Knuth, 1981): the work of Okasaki (1998) and Hinze (1998) sug-

gests that numerical representations form an interesting class of ornaments, from which one would build data-structures from numerical systems, taking advantage of their structural ties to determine their algorithmic complexity. Ko (2014) gave an extensive treatment of binomial heaps, which correspond to a binary representation system. To the best of my knowledge, a systematic treatment of numerical representation through the lenses of ornaments remains to be done.

## 5 Past, Present, and Future of Ornaments

The categorical presentation of Dagand and McBride (2013) generalizes the present article, working in any locally Cartesian-closed category (Seely, 1983) and manipulating polynomial functors (Gambino & Kock, 2013). It also carefully establishes several equivalences between the type theoretic notions and their categorical models. For pedagogical purposes, the present article is exclusively developed in type theory (which morally corresponds to the internal language of a locally Cartesian-closed category). This allows us to bridge the gap between an operational understanding of ornaments and their mathematical structure, thus appealing to programmers. The purely categorical presentation (Dagand & McBride, 2013) provides the proofs which we have left out in this article. In particular, the adequacy of our model of ornaments and the validity of their algebra are taken for granted here.

In effect, this article provides a middle-ground between the type theoretic presentation of McBride (2011) and the categorical presentation of Dagand and McBride (2013). The type theoretic presentation, further extended by Ko and Gibbons (2011) and Dagand and McBride (2012), benefits from its intensionality: it is computationally effective. However, its syntactic nature impedes conceptual clarity. Case in point: each of the aforementioned articles start by introducing a customized universe of ornaments. Indeed, ornaments are tightly coupled with the underlying universe of datatypes. Our more extensional presentation allows us to give a crisp definition of ornaments as structure-preserving transformations, independently of the actual presentation of datatypes.

Nonetheless, the importance of the type theoretic approach should not be underestimated: its syntactic nature allows us to experiment with the challenge of offering ornaments in a programming environment. In this context, much of the prescriptive treatment of ornaments remain to be done. While we have observed many incarnations of the algebra of ornaments in Section 4, we lack a high-level language to build these artifacts. Currently, we are lead to directly (and tediously) manipulate signatures and their algebras. For instance, we had to manually write algebras to obtain algebraic or relational ornaments, which would hardly be acceptable in a user-friendly environment.

Ornaments also aspire to be a key component in the non-dependent programmer's toolbox. The work of William *et al.* (2014) puts ornaments in practice in the context of a purely functional subset of the OCaml language, taking a pragmatic stance. It explores various applications of ornaments, ranging from code transformation to semi-automated refactoring. While pragmatism imposes several restrictions on how ornaments can be presented to the users, the theoretical framework developed in this article provides a blueprint through which to study and compare the restrictions that arise in practice.

Moving upward on the dependency spectrum, ornaments would also be well-suited to functional languages offering GADTs. Datatypes being meta-theoretical objects in those

systems, this raises some interesting questions in terms of implementation: in line with the work of William *et al.*, a purely syntactic treatment would be necessary. Beyond datatypes, programmers are also interested in transporting functions across ornaments: having implemented the addition of natural numbers and presented lists as ornaments of numbers, one would like to be able to "derive" the concatenation of lists from addition. If the theoretical aspects have been partially treated from a semantic standpoint (Dagand & McBride, 2012), a more syntactic (and practical) treatment is still in its infancy (Williams *et al.*, 2014).

Overall, ornaments offer a controlled form of meta-programming over datatypes. Using the algebra of ornaments, we can build and combine datatypes in a compositional manner. The first promise of ornaments is thus to free the programmer from most of the burden of defining domain-specific data-logics over general-purpose data-structures. In this context, it makes sense to restrict ourselves to structure-preserving transformations: because they preserve the recursive structure of datatypes, we can relate the functions defined recursively over such datatypes. This observation is at the heart of numerical representations, in which the complexity of, say, merging two data-structures boils down to the complexity of adding two numbers in the corresponding numerical system. The second promise of ornaments is that we can help programmers in adapting library-provided functions to account for their domain-specific invariants while preserving their computational behavior.

Ornaments are still at an early stage of development: we have yet to fulfill the above promises in a user-friendly environment while much of the applications and algebraic structure of ornaments remain to be discovered and exploited.

## 6 Conclusion

In this article, we have distilled the essence of ornaments. To do so, we first gave a model of indexed datatypes based on many-sorted signatures. Adopting a more semantic approach allowed us to embrace the many forms of indexing, ranging from GADTs to inductive families. In this framework, we have characterized ornaments as structure-preserving transformations of datatypes.

The formalism of many-sorted signatures benefits from being a type theoretic object, yet being sufficiently abstract for our purposes. By its roots in type theory, it is effective and makes for an interesting experimental platform. Its abstract nature allows us to reason extensionally, which was a key ingredient in obtaining a simple model of ornaments.

By relating ornaments to Cartesian morphisms, we have shed a new light on the previous works in type theory. Our emphasis on the extensional property of ornaments – preserving the recursive structure – gives a simple, easily verifiable criterion for ornaments. We can for instance easily determine whether a datatype is the ornament of another. Because Cartesian morphisms are a standard mathematical object, we also benefit from a large body of mathematical results, which then translate to interesting computational artifacts.

Throughout this article, our objective has been to reach a wider audience, beyond type theoretic or category theoretic circles. In the process, we have stripped the presentation of ornaments from its type theoretic specificities. The language of many-sorted signatures provides a *lingua franca* to model the indexed datatypes of a large variety of programming languages. They would also guide the translation of the concepts of ornaments back to these languages.

**Acknowledgments** I am very grateful to the Journal of Functional Programming reviewers, whose comments helped significantly improve this article. I also wish to thank Gabor Greif, who provided feedback on an early draft. This work was partially supported by the Emergence(s) program of the City of Paris.

## References

Abbott, Michael. (2003). *Categories of containers*. Ph.D. thesis, University of Leicester.

Abbott, Michael, Altenkirch, Thorsten, McBride, Conor, & Ghani, Neil. (2005). $\partial$ for data: Differentiating data structures. *Fundamenta informaticae*, **65**(1-2), 1–28.

Adelson-Velskii, Georgy, & Landis, Evgenii. (1962). An algorithm for the organization of information. *Doklady akademii nauk USSR*, **146**(2), 263–266.

Atkey, Robert, Johann, Patricia, & Ghani, Neil. (2012). Refining inductive types. *Logical methods in computer science*, **8**.

Brady, Edwin. (2013). Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of functional programming*, **23**(5), 552–593.

Cheney, James, & Hinze, Ralf. (2003). *First-class phantom types*. Tech. rept. Cornell University.

Chlipala, Adam. (2013). *Certified programming with dependent types*. MIT Press.

Constable, R. L. (1986). *Implementing mathematics with the nuprl proof development system*. Prentice Hall.

Coq development team. (2015). *The Coq proof assistant reference manual*.

Dagand, Pierre-Evariste, & McBride, Conor. (2012). Transporting functions across ornaments. *Pages 103–114 of: International conference on functional programming*.

Dagand, Pierre-Evariste, & McBride, Conor. (2013). A categorical treatment of ornaments. *Logics in computer science*.

Dybjer, Peter. (1994). Inductive families. *Formal aspects of computing*, **6**(4), 440–465.

Dybjer, Peter. (1997). Representing inductively defined sets by wellorderings in Martin-Löf's type theory. *Theoretical computer science*, **176**(1-2), 329–335.

Freeman, Tim, & Pfenning, Frank. (1991). Refinement types for ML. *Pages 268–277 of: Programming language design and implementation*.

Fumex, Clément. (2012). *Induction and coinduction schemes in category theory*. Ph.D. thesis, University of Strathclyde.

Gambino, Nicola, & Hyland, Martin. (2004). Wellfounded trees and dependent polynomial functors. *Pages 210–225 of: Types for proofs and programs*, vol. 3085.

Gambino, Nicola, & Kock, Joachim. (2013). Polynomial functors and polynomial monads. *Mathematical proceedings of the cambridge philosophical society*, **154**(1), 153–192.

Goguen, Healfdene, & Luo, Zhaohui. (1993). Inductive data types: well-ordering types revisited. *Pages 198–218 of: Workshop on logical environments*.

Goguen, J. A., Thatcher, J. W., Wagner, E. G., & Wright, J. B. 1975 (may). Abstract Data-Types as initial algebras and correctness of data representations. *Proceedings of the conference on computer graphics, pattern recognition and data structure*.

Guibas, Leo J., & Sedgewick, Robert. (1978). A dichromatic framework for balanced trees. *Pages 8–21 of: Foundations of computer science*.

Hamana, Makoto, & Fiore, Marcelo. (2011). A foundation for GADTs and inductive families: dependent polynomial functor approach. *Pages 59–70 of: Workshop on generic programming*.

Hinze, Ralf. (1998). *Numerical representations as higher-order nested datatypes*. Tech. rept.

Huet, Gerard. (1997). The zipper. *Journal of functional programming*, **7**(05), 549–554.

Knuth, Donald E. (1981). *The art of computer programming, volume II: Seminumerical algorithms*. Addison-Wesley.

Ko, Hsiang-Shang. (2014). *Analysis and synthesis of inductive families*. Ph.D. thesis, University of Oxford.

Ko, Hsiang-Shang, & Gibbons, Jeremy. (2011). Modularising inductive families. *Pages 13–24 of: Workshop on generic programming*.

Ko, Hsiang-Shang, & Gibbons, Jeremy. (2013). Relational algebraic ornaments. *Pages 37–48 of: Workshop on dependently-typed programming*.

Martin-Löf, Per. (1984). *Intuitionistic type theory*. Bibliopolis Napoli.

McBride, Conor. (2011). *Ornamental algebras, algebraic ornaments*. Unpublished.

Morris, Peter. (2007). *Constructing universes for generic programming*. Ph.D. thesis, University of Nottingham.

Morris, Peter, & Altenkirch, Thorsten. (2009). Indexed containers. *Logics in computer science*.

Morris, Peter, Altenkirch, Thorsten, & Ghani, Neil. (2009). A universe of strictly positive families. *International journal of foundations of computer science*, **20**(1), 83–107.

Norell, Ulf. (2007). *Towards a practical programming language based on dependent type theory*. Ph.D. thesis, Chalmers University of Technology.

Okasaki, Chris. (1998). *Purely functional data structures*. Cambridge University Press.

Petersson, Kent, & Synek, Dan. (1989). A set constructor for inductive sets in Martin-Löf's type theory. *Pages 128–140 of: Category theory and computer science*.

Pottier, François, & Régis-Gianas, Yann. (2006). Stratified type inference for generalized algebraic data types. *Pages 232–244 of: Principles of programming languages*.

Schrijvers, Tom, Peyton Jones, Simon, Sulzmann, Martin, & Vytiniotis, Dimitrios. (2009). Complete and decidable type inference for GADTs. *Pages 341–352 of: International conference on functional programming*.

Seely, R. A. G. (1983). Locally cartesian closed categories and type theory. *Mathematical proceedings of the cambridge philosophical society*, **95**.

Sheard, Tim, & Linger, Nathan. (2007). Programming in Omega. *Pages 158–227 of: Central european functional programming school*. Lecture Notes in Computer Science, vol. 5161.

Smyth, M.B., & Plotkin, G.D. (1977). The category-theoretic solution of recursive domain equations. *Pages 13–17 of: Foundations of computer science*.

Swamy, Nikhil, Chen, Juan, Fournet, Cédric, Strub, Pierre-Yves, Bhargavan, Karthikeyan, & Yang, Jean. (2011). Secure distributed programming with value-dependent types. *Pages 266–278 of: International conference on functional programming*.

Williams, Thomas, Dagand, Pierre-Évariste, & Rémy, Didier. (2014). Ornaments in practice. *Pages 15–24 of: Workshop on generic programming*.