Under consideration for publication in J. Functional Programming

Transporting Functions across Ornaments

Pierre-Evariste Dagand
Inria

Conor McBride
University of Strathclyde
(*e-mail:* pierre-evariste.dagand@inria.fr,conor@cis.strath.ac.uk)

Abstract

Programming with dependent types is a blessing and a curse. It is a blessing to be able to bake invariants into the definition of datatypes: we can finally write correct-by-construction software. However, this extreme accuracy is also a curse: a datatype is the combination of a structuring medium together with a special purpose logic. These domain-specific logics hamper any attempt to reuse code across similarly structured data.

In this article, we capitalise on the structural invariants of datatypes. To do so, we first adapt the notion of ornament to our universe of inductive families. We then show how code reuse can be achieved by ornamenting functions. Using these functional ornaments, we capture the relationship between functions such as the addition of natural numbers and the concatenation of lists. With this knowledge, we demonstrate how the implementation of the former informs the implementation of the latter: the user can ask the definition of addition to be lifted to lists and she will only be asked the details necessary to carry on adding lists rather than numbers.

Our presentation is formalised in a type theory with a universe of datatypes and all our constructions have been implemented as generic programs, requiring no extension to the type theory.

1 Introduction

Imagine designing a library for an ML-like language. For instance, we start with natural numbers and their operations, then we move to binary trees, then rose trees, *etc.* It is the garden of Eden: datatypes are data-*structures*, each coming with its unique and optimised set of operations. If we move to a language with richer datatypes, such as a dependently-typed language, we enter the Augean stables. Where we used to have binary trees, now we have complete binary trees, red-black trees, AVL trees, and countless other variants. Worse, we have to duplicate code across these tree-like datatypes: because they are defined upon this common binarily branching structure, a lot of computationally identical operations will have to be duplicated for the type checker to be satisfied.

Since their first introduction in ML, datatypes have evolved: besides providing an organising *structure* for computation, they are now offering more *control* over what is a valid result. With richer datatypes, we can enforce invariants on top of the data-structures. In such a system, programmers strive to express the correctness of their programs in the types: a well-typed program is correct *by construction*.

A simple yet powerful recipe to obtain richer datatypes is to *index* the data-structure. These datatypes have originally been studied in type theory under the name of *inductive families* (Dybjer, 1994; Morris *et al.*, 2009). Inductive families made it to mainstream functional programming with Generalised Algebraic Data-Types (GADTs) (Cheney & Hinze, 2003; Schrijvers *et al.*, 2009), a subset of inductive families for which the principal-types property is preserved thus enabling modular (local) type inference.

Refinement types (Freeman & Pfenning, 1991; Swamy *et al.*, 2011) are another technique to equip data-structures with rich invariants. In a dependently-typed setting, refinement types are expressible in terms of Σ-types. As such, they offer a clear-cut separation between the data (what is predicated upon, *i.e.* the first projection of the Σ-type) and the logic (the predicate, *i.e.* the second projection of the Σ-type). This approach benefits from a straightforward compilation strategy, which simply erases the refining predicates. Atkey *et al.* (2012) have shown how refinement types relate to inductive families.

1.1 Ornaments?

However, these carefully crafted datatypes are a threat to any library design: the same data-*structure* is used for logically incompatible purposes. This explosion of specialised datatypes is overwhelming: these objects are too specialised to fit in a global library. Yet, because they share this common structure, many operations on them are extremely similar, if not exactly the same. To address this issue, McBride (2013) developed *ornaments*, describing how one datatype can be enriched into another *with the same structure*. Such structure-preserving transformations take two forms. First, we can *extend* the initial type with more information.

1.1 Example (Ornament: Extending the Booleans to the option type)**.** We can extend the Booleans to the option type by attaching an $a : A$ to the constructor true:

$$\begin{array}{l}\textbf{data}\ \mathsf{Bool} : \textsc{Set}\ \textbf{where} \\ \quad \mathsf{Bool} \ni \mathsf{true} \\ \quad\quad \mid \mathsf{false}\end{array} \quad \overset{\mathsf{Maybe\text{-}Orn}\,A}{\Rightarrow} \quad \begin{array}{l}\textbf{data}\ \mathsf{Maybe}\,[A : \textsc{Set}] : \textsc{Set}\ \textbf{where} \\ \quad \mathsf{Maybe}\,A \ni \mathsf{just}\,(a : A) \\ \quad\quad \mid \mathsf{nothing}\end{array}$$

△

1.2 Example (Ornament: Extending numbers to lists)**.** Or we can extend natural numbers to lists by inserting an $a : A$ at each successor node:

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

△

Second, we can *refine* the indexing of the initial type, following a finer discipline. By refining the indices of a datatype, we make it logically more discriminating.

1.3 Example (Ornament: Refining numbers to finite sets)**.** We refine natural numbers to finite sets by indexing the number with an upper-bound:

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

Put otherwise, the datatype $\mathsf{Fin}\,n$ is a type of cardinality n.

△

We can also do both at the same time, as illustrated by the following example.

1.4 Example (Ornament: Extending and refining numbers to vectors)**.** We extend natural numbers to lists while refining the index to represent the length of the list thus constructed:

$$\begin{array}{l}\textbf{data}\ \mathsf{Nat} : \textsc{Set}\ \textbf{where} \\ \quad \mathsf{Nat} \ni 0 \\ \quad\quad \mid \mathsf{suc}\,(n : \mathsf{Nat})\end{array} \overset{\mathsf{VecO}\,A}{\Longrightarrow} \begin{array}{l}\textbf{data}\ \mathsf{Vec}\,[A : \textsc{Set}]\,(n : \mathsf{Nat}) : \textsc{Set}\ \textbf{where} \\ \quad \mathsf{Vec}\,A\quad (n = 0) \quad \ni \mathsf{nil} \\ \quad \mathsf{Vec}\,A\,(n = \mathsf{suc}\,n') \ni \mathsf{cons}\,(n' : \mathsf{Nat})\,(a : A)\,(ys : \mathsf{Vec}\,A\,n')\end{array}$$

Note that we declare datatype parameters in brackets – *e.g.*, $[A : \textsc{Set}]$ – and datatype indices in parentheses – *e.g.*, $(n : \mathsf{Nat})$. We make equational constraints on the latter only when needed, and explicitly – *e.g.*, $(n = \mathsf{suc}\,n')$. We come back to the notation for inductive definitions in Section 3.3.

△

Because of their constructive nature, ornaments are not merely identifying similar structures: they give an effective recipe to build new datatypes from old, guaranteeing by construction that the structure is preserved. Hence, we can obtain a plethora of new datatypes with minimal effort.

1.2 Functional ornaments!

Whilst we have a good handle on the transformation of individual datatypes, we are still facing a major reusability issue: a datatype often comes equipped with a set of operations. Ornamenting this datatype, we have to entirely re-implement many similar operations. For example, the datatype Nat comes with operations such as addition and subtraction. When defining lists as an ornament of natural numbers, it seems natural to transport the structure-preserving functions of Nat to $\mathsf{List}\,A$, such as moving from addition of natural numbers to concatenation of lists:

$$\begin{array}{llllll}(m : \mathsf{Nat}) & + & (n : \mathsf{Nat}) & : & \mathsf{Nat} & \\ 0 & + & n & \mapsto & n & \\ (\mathsf{suc}\,m) & + & n & \mapsto & \mathsf{suc}\,(m + n) & \end{array} \Rightarrow \begin{array}{llllll}(xs : \mathsf{List}\,A) & +\!\!+ & (ys : \mathsf{List}\,A) & : & \mathsf{List}\,A & \\ \mathsf{nil} & +\!\!+ & ys & \mapsto & ys & \\ (\mathsf{cons}\,a\,xs) & +\!\!+ & ys & \mapsto & \mathsf{cons}\,a\,(xs +\!\!+ ys) & \end{array}$$

or, from subtraction of natural numbers to dropping a prefix:

$$\begin{array}{llllll}(m : \mathsf{Nat}) & - & (n : \mathsf{Nat}) & : & \mathsf{Nat} & \\ 0 & - & n & \mapsto & 0 & \\ m & - & 0 & \mapsto & m & \\ (\mathsf{suc}\,m) & - & (\mathsf{suc}\,n) & \mapsto & m - n & \end{array} \Rightarrow \begin{array}{llllll}\mathsf{drop}\,(xs : \mathsf{List}\,A) & (n : \mathsf{Nat}) & : & \mathsf{List}\,A & & \\ \mathsf{drop} & \mathsf{nil} & n & \mapsto & \mathsf{nil} & \\ \mathsf{drop} & xs & 0 & \mapsto & xs & \\ \mathsf{drop}\,(\mathsf{cons}\,a\,xs) & (\mathsf{suc}\,n) & \mapsto & \mathsf{drop}\,xs\,n & & \end{array}$$

More interestingly, the function we start with may involve several datatypes, each of which may be ornamented differently. In this paper, we develop the notion of *functional ornament* as a generalisation of ornaments to functions:

- We manually transport the comparison of numbers to the list lookup function in Section 2. This example provides the impetus for the rest of this article: we aim at explaining the structure behind it, generalise, and automate it;
- For this article to be self-contained, we recall the type theoretic foundations (Chapman *et al.*, 2010) upon which this article builds in Section 3. We strive to provide an intuition for our universe-based presentation of datatypes, and describe a convenient notation for inductive definitions;
- We adapt ornaments to our universe of datatypes in Section 4. This presentation benefits greatly from our ability to inspect indices when defining datatypes. This allows us to consider ornaments that *delete* information, yielding a key simplification in the construction of the algebraic ornament from the ornamental algebra;

- We describe how functions can be transported through functional ornaments. We formalise the concept of functional ornament by a universe construction in Section 5. Based on this universe, we establish the connection between a base function (such as addition and subtraction) and its ornamented version (such as, respectively, $- +\!\!+ -$ and drop). Within this framework, we redevelop the example of Section 2 with all the automation offered by our framework;
- In Section 6, we provide further support to drive the computer into lifting functions. As we can see from our examples above, the lifted functions often follow the same recursion pattern and return similar constructors: with a few smart constructors, we shall remove further clutter from our libraries.

This article is an exercise in constructive mathematics: upon identifying an isomorphism, we shall look at it with our constructive glasses and obtain an effective procedure to cross it. It is crucial to note that this article is built entirely *within* type theory. No change or adaptation to the meta-theory is required. In particular, the validity of our constructions is justified by mere type checking.

1.5 Remark (Notations). We shall write our code in a syntax inspired by the Epigram programming language (McBride & McKinna, 2004). For an optimal experience, we recommend reading the colour version of this article, available on Dagand's webpage. Colours are used to classify the terms of the type theory. We also make use of the *by* ($\Leftarrow$) and *return* ($\mapsto$) programming gadgets, further extending them to account for the automatic lifting of functions. For brevity, we write pattern-matching definitions when the recursion pattern is evident and unremarkable.

As in ML, unbound variables in type signatures are universally quantified, further abating syntactic noise. For higher-order functions, we indicate the implicit arguments with the quantifier $\forall x.\,(\ldots)$ – or its annotated variant $\forall x\!:\!T.\,(\ldots)$ – as follows:

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

Because we implicitly quantify over unbound type variables, these variables are not explicitly bound in the definition. We rely on the convention that these implicit arguments are automatically in scope of the definition, using the same variable name. For example, in the following definition, n is universally quantified in the type declaration and is in scope in the definition of lengthVec:

$$\begin{array}{lll} \mathsf{lengthVec}\,(vs\!:\!\mathsf{Vec}\,A\,n) & : & \mathsf{Nat} \\ \mathsf{lengthVec} \quad vs & \mapsto & n \end{array}$$

The syntax of datatype definitions draws upon the ML tradition as well: its novelty will be presented by way of examples in Section 3. Following mathematical usage, we shall extensively use mixfix operators, *i.e.* operators in prefix, infix, postfix, or closed form.

All the constructions presented in this article have been modelled in Agda, using only standard inductive definitions and two levels of universes. Rather than presenting the machine-checked code, we have chosen to use an high-level notation. This notation relies on the reader's ability to cope with ambiguity. We shall therefore indulge in many abuse of language, as is common in mathematics. This allows us to focus on conveying ideas to the reader, rather than on satisfying a type checker. Throughout this article, we relate the high-level definitions to their Agda counterparts using a MODEL footnote indicating their location. For an absolutely formal treatment, we therefore refer the reader to the companion formalisation. The formalisation is available on Dagand's website.

$\Diamond$

$$
\begin{array}{llllll}
(m:\mathsf{Nat}) & < & (n:\mathsf{Nat}) & : & \mathsf{Bool} & \\
m & < & 0 & \mapsto & \mathsf{false} & \stackrel{?}{\Longrightarrow} \\
0 & < & (\mathsf{suc}\,n) & \mapsto & \mathsf{true} & \\
(\mathsf{suc}\,m) & < & (\mathsf{suc}\,n) & \mapsto & m<n &
\end{array}
\qquad
\begin{array}{llllll}
\mathsf{lookup} & (m:\mathsf{Nat}) & (xs:\mathsf{List}\,A) & : & \mathsf{Maybe}\,A & \\
\mathsf{lookup} & m & \mathsf{nil} & \mapsto & \mathsf{nothing} & \\
\mathsf{lookup} & 0 & (\mathsf{cons}\,a\,xs) & \mapsto & \mathsf{just}\,a & \\
\mathsf{lookup} & (\mathsf{suc}\,n) & (\mathsf{cons}\,a\,xs) & \mapsto & \mathsf{lookup}\,n\,xs &
\end{array}
$$

Fig. 1: Implementation of $-<-$ and lookup

A shorter version of this article has appeared in the proceedings of ICFP 2012 (Dagand & McBride, 2012). This version benefits from several presentational modifications to include significantly more examples (in particular, in Section 3 and Section 4). The running example – lifting the comparison function of natural numbers – is also complemented by another example, lifting the addition of natural numbers. Worked out examples, throughout the paper, shall help the reader build a strong intuition of the generic constructions at play. We have also extended the original paper with new material. In Section 4.2, we cast the forcing and detagging transformations of Brady *et al.* (Brady *et al.*, 2003) in our universe of datatypes. Using these intuitions, we have streamlined the definition of reornaments and discuss the limits of iterating reornaments (Section 4.5). We have extended the lifting of recursion patterns to handle induction and case analysis (Section 6). Our treatment of the lifting of constructors (Section 6) has also treated in more details and its underlying mechanisms has been thoroughly illustrated.

2 From Comparison to Lookup, Manually

There is an astonishing resemblance between the comparison function $-<-$ on numbers and the list lookup function (Fig. 1). Interestingly, the similarity is not merely at the level of types. It is also in their implementation: their definition follows the same pattern of recursion (first, case analysis on the second argument; then induction on the first argument) and they both return a failure value (respectively, false and nothing) in the first case analysis and a success value (respectively, true and just) in the base case of the induction.

This raises the question: what *exactly* is the relation between $-<-$ and lookup? Also, could we use the implementation of $-<-$ to guide the construction of lookup? First, let us work out the relation at the type level. To this end, we use ornaments to explain how each individual datatype has been promoted when going from $-<-$ to lookup:

$$
\begin{array}{lcccccc}
-<- & : & \mathsf{Nat} & \to & \mathsf{Nat} & \to & \mathsf{Bool} \\
 & & \Big\downarrow{\scriptstyle \mathsf{idO}\,\mathsf{Nat}\text{-}\mathsf{func}} & & \Big\downarrow{\scriptstyle \mathsf{List}\text{-}\mathsf{Orn}\,A} & & \Big\downarrow{\scriptstyle \mathsf{Maybe}\text{-}\mathsf{Orn}\,A} \\
\mathsf{lookup} & : & \mathsf{Nat} & \to & \mathsf{List}\,A & \to & \mathsf{Maybe}\,A
\end{array}
$$

Note that the first argument is ornamented to itself, or put differently, it has been ornamented by the identity ornament idO (Definition 4.9, p.20).

Each of these ornaments come with a forgetful map:

$$
\begin{array}{lllll}
\mathsf{length} & (as:\mathsf{List}\,A) & : & \mathsf{Nat} & \\
\mathsf{length} & \mathsf{nil} & \mapsto & 0 & \\
\mathsf{length} & (\mathsf{cons}\,a\,as) & \mapsto & \mathsf{suc}\,(\mathsf{length}\,as) &
\end{array}
\qquad
\begin{array}{lllll}
\mathsf{isJust} & (m:\mathsf{Maybe}\,A) & : & \mathsf{Bool} & \\
\mathsf{isJust} & \mathsf{nothing} & \mapsto & \mathsf{false} & \\
\mathsf{isJust} & (\mathsf{just}\,a) & \mapsto & \mathsf{true} &
\end{array}
$$

As we shall see in Section 4.3, the forgetful maps can be automatically derived from the ornament definition.

Using these forgetful maps we deduce a relation, at the operational level, between $-<-$ and lookup. This relation is uniquely determined by the ornamentation of the individual datatypes. This *coherence property* is expressed as follows

$$(n : \mathsf{Nat})(xs : \mathsf{List}\,A) \to \mathsf{isJust}\,(\mathsf{lookup}\,n\,xs) = n < \mathsf{length}\,xs$$

or, equivalently, using a commuting diagram:

$$\begin{array}{ccc} \mathsf{Nat} \times \mathsf{List}\,A & \xrightarrow{\mathsf{lookup}} & \mathsf{Maybe}\,A \\ \downarrow{\scriptstyle \mathsf{id} \times \mathsf{length}} & & \downarrow{\scriptstyle \mathsf{isJust}} \\ \mathsf{Nat} \times \mathsf{Nat} & \xrightarrow[-<-]{} & \mathsf{Bool} \end{array}$$

2.1 Remark (Vocabulary). We call the function we start with the *base function* (here, $-<-$), its type being the *base type* (here, $\mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Bool}$). The richer function type built by ornamenting the individual pieces is called the *functional ornament*[1] of the base type (here, $\mathsf{Nat} \to \mathsf{List}\,A \to \mathsf{Maybe}\,A$). A function inhabiting this type is called a *lifting* (here, lookup). A lifting is said to be *coherent* if it satisfies the coherence property.

◊

2.2 Remark (Coherence and functional ornament). It is crucial to understand that the coherence of a lifting is relative to a given functional ornament: the same base function ornamented differently would give rise to a different coherence property.

For example, the base type $\mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}$ can be functionally ornamented to

$$\mathsf{Nat} \to \mathsf{List}\,A \to \mathsf{List}\,A$$

for which drop is a coherent solution with respect to $---$. However, it can also be functionally ornamented to

$$\mathsf{List}\,A \to \mathsf{List}\,A \to \mathsf{List}\,A$$

for which $-{+}{+}-$ is a coherent solution with respect to $-+-$. Nonetheless, the two solutions are exclusive: drop is not coherent with respect to $-+-$, nor is $-{+}{+}-$ coherent with respect to $---$.

◊

We now have a better grasp of the relation between the base function and its lifting. However, lookup remains to be implemented while making sure that it satisfies the coherence property. Traditionally, one would stop here: we would implement lookup and prove the coherence as a theorem. This works rather well in a system like Coq since it offers a powerful theorem proving environment. It does not work so well in a system like Agda that does not offer tactics to its users, forcing them to write explicit proof terms. It would not work at all in an ML language with GADTs, which has no notion of proof.

Historically, Coq has been engineered to specify (simply-typed) programs using (dependently-typed) propositions (Paulin-Mohring, 1989). This paradigm started shifting with languages such as Epigram,

[1] Note that a functional ornament is entirely determined by the ornamentation of its individual components, which we shall (unambiguously) call "a functional ornament" in Section 5.2. This follows the common usage of saying that "lists are an ornament of natural numbers", when in fact lists are the *result* of interpreting an ornament of natural numbers.

Cayenne, or Agda: these systems offer an environment for *dependently-typed programming*. In this setting, the approach that consists in writing a simply-typed programming to prove it correct *after the fact* feels utterly laborious. If we have dependent types, why should we use them only for *proofs*, as an afterthought? A dependently-typed programming environment lets us write a lookup function *correct by construction*: by implementing a more finely indexed version of lookup, the user drives the type checker into verifying the necessary invariants. This article is an exploration of this paradigm, in which we develop techniques to relieve the dependently-typed programmer from the burden of proofs.

To get the computer to work for us, we would rather implement the function ilookup

$$\begin{array}{lll} \mathsf{ilookup}\ (m : \mathsf{Nat})\ (vs : \mathsf{Vec}\ A\ n) & : & \mathsf{IMaybe}\ A\ (m < n) \\ \mathsf{ilookup}\quad m\quad\quad \mathsf{nil} & \mapsto & \mathsf{nothing} \\ \mathsf{ilookup}\quad \mathsf{0}\quad (\mathsf{cons}\ a\ vs) & \mapsto & \mathsf{just}\ a \\ \mathsf{ilookup}\ (\mathsf{suc}\ m)\quad (\mathsf{cons}\ a\ vs) & \mapsto & \mathsf{ilookup}\ m\ vs \end{array}$$

where IMaybe$A$ is the option type indexed by the truth value computed by isJust. It is defined as follows

$$\begin{array}{l} \textbf{data}\ \mathsf{IMaybe}\ [A : \textsc{Set}]\ (b : \mathsf{Bool}) : \textsc{Set}\ \textbf{where} \\ \quad \mathsf{IMaybe}\ A\ (b = \mathsf{true}) \ni \mathsf{just}\ (a : A) \\ \quad \mathsf{IMaybe}\ A\ (b = \mathsf{false}) \ni \mathsf{nothing} \end{array}$$

and it comes with a forgetful map:

$$\begin{array}{lll} \mathsf{forgetIMaybe}\ (ima : \mathsf{IMaybe}\ A\ b) & : & (ma : \mathsf{Maybe}\ A) \times \mathsf{isJust}\ ma = b \\ \mathsf{forgetIMaybe}\quad (\mathsf{just}\ a) & \mapsto & (\mathsf{just}\ a, \mathsf{refl}) \\ \mathsf{forgetIMaybe}\quad \mathsf{nothing} & \mapsto & (\mathsf{nothing}, \mathsf{refl}) \end{array}$$

2.3 Remark. We overload the constructors of Maybe and IMaybe: for a bi-directional type checker, there is no ambiguity as constructors are checked against their type.

◊

The rationale behind ilookup is to *index* the types of lookup by their unornamented version. In effect, we index the types of its arguments by the (respective) arguments $m : \mathsf{Nat}$ and $n : \mathsf{Nat}$ of $- < -$ and we index the type of its result by $m < n$. By doing so, we can make sure that the result computed by ilookup respects the output of $- < -$ on the unornamented indices: the result is necessarily correct, *by indexing*! The type of ilookup is naturally derived from the ornamentation of $- < -$ into lookup and is uniquely determined by the functional ornament we start with. We shall automate its construction in Section 5.3 with the notion of *patch* (Definition 5.15).

2.4 Remark (Vocabulary). Expanding further our vocabulary, such a finely indexed function that is coherent by construction is called a *coherent liftings*.

We had separately introduced the notion of *lifting* and *coherence* in Remark 2.1, with the idea that a lifting is not necessarily coherent. Here, we are defining the *coherent lifting* as those liftings that are coherent by construction. In fact, we shall establish in Theorem 5.19 that a lifting satisfying the coherence condition is isomorphic to a coherent lifting. There is therefore no ambiguity in identifying both notions of a "coherent lifting" and of a "lifting satisfying the coherence condition".

◊

Ko and Gibbons (2011) use ornaments to specify the coherence requirements for functional liftings, but we work the other way around. From ilookup, we extract the lookup function

$$\begin{array}{lll} \mathsf{lookup}\ (m : \mathsf{Nat})\ (xs : \mathsf{List}\ A) & : & \mathsf{Maybe}\ A \\ \mathsf{lookup}\quad m\quad\quad xs & \mapsto & \pi_0(\mathsf{forgetIMaybe}\ (\mathsf{ilookup}\ m\ (\mathsf{makeVec}\ xs))) \end{array}$$

and its proof of correctness

$$\begin{array}{l}\mathsf{cohLookup}\,(n:\mathsf{Nat})\,(xs:\mathsf{List}\,A)\;:\;\mathsf{isJust}\,(\mathsf{lookup}\,n\,xs)=n<\mathsf{length}\,xs\\ \mathsf{cohLookup}\quad m\quad xs\quad\mapsto \pi_1\,(\mathsf{forgetIMaybe}\,(\mathsf{ilookup}\,m\,(\mathsf{makeVec}\,xs)))\end{array}$$

where $\mathsf{makeVec}:(xs:\mathsf{List}\,A)\to\mathsf{Vec}\,A\,(\mathsf{length}\,xs)$ turns a list into a vector of the corresponding length. Operationally, it is an identity. In Section 4.4, we show that it can be automatically derived from the ornament of lists.

The construction of lookup and cohLookup feels automatic: we shall see in Section 5.4 how lookup can automatically be obtained by *patching* (Definition 5.22), while cohLookup is merely an instance of a generic *coherence* result (Definition 5.23).

2.5 Remark (ilookup *vs.* vlookup). The function ilookup is very similar to the more familiar vlookup function:

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

These two definitions are actually equivalent, thanks to the isomorphism

$$(m:\mathsf{Nat})\to\mathsf{IMaybe}\,A\,(m<n)\quad\cong\quad\mathsf{Fin}\,n\to A$$

where we silently lift the boolean predicate $m<n$ at the type-level, as is common practice in SSReflect (Gonthier *et al.*, 2008) for instance.

Intuitively, we can move the constraint "$m<n$" from the result – where we return an object of type $\mathsf{IMaybe}\,A\,(m<n)$ – to the premise – where we expect an object of type $\mathsf{Fin}\,n$. Indeed, we can think of the type $\mathsf{Fin}\,n$ as the combination of a number $m:\mathsf{Nat}$ together with a proof that $m<n$.

◊

With this example, we have manually unfolded the key steps of the construction of a lifting of $-<-$. Let us recapitulate each steps:

- Start with a *base function*, here $-<-:\mathsf{Nat}\to\mathsf{Nat}\to\mathsf{Bool}$;
- Ornament its inductive components as desired, here Nat to $\mathsf{List}\,A$ and Bool to $\mathsf{Maybe}\,A$ in order to describe the lifting of interest, here $\mathsf{lookup}:\mathsf{Nat}\to\mathsf{List}\,A\to\mathsf{Maybe}\,A$ satisfying

$$(n:\mathsf{Nat})(xs:\mathsf{List}\,A)\to\mathsf{isJust}\,(\mathsf{lookup}\,n\,xs)=n<\mathsf{length}\,xs$$

- Implement a carefully indexed version of the lifting, here

$$\mathsf{ilookup}:(m:\mathsf{Nat})(vs:\mathsf{Vec}\,A\,n)\to\mathsf{IMaybe}\,A\,(m<n)$$

- Derive the lifting, here lookup, and its coherence proof, without *writing* a proof!

Besides, ilookup is a useful addition to our library: it corresponds to the familiar vector lookup function, a function that one would have implemented anyway. Thus, ilookup is not just some scaffolding necessary to define lookup, it is a purposeful operation on its own.

This manual unfolding of the lifting is instructive: it involves many constructions on datatypes (here, the datatypes $\mathsf{List}\,A$ and $\mathsf{Maybe}\,A$) as well as on functions (here, the type of ilookup, the definition of lookup and its coherence proof). Yet, it feels like a lot of these constructions could be automated. In Section 5, we shall build the machinery to describe these transformations and obtain them *within* the type theory itself.

3 A Universe of Datatypes

In dependently-typed systems such as Coq or Agda, datatypes are an external entity: each datatype definition extends the type theory with new introduction and elimination forms. The validity of a datatype definition is guaranteed by a positivity-checker that is part of the meta-theory of the proof system. A consequence is that, from within the type theory, it is not possible to create or manipulate inductive definitions, since they belong to the meta-theory.

In previous work (Chapman *et al.*, 2010), we have shown how to internalise inductive families into type theory. The practical outcome of this approach is that we can manipulate datatype declarations as first-class objects. We can program over the grammar of datatypes and, in particular, we can compute new datatypes from old. This is particularly useful to formalise the notion of ornament entirely within type theory. This also has a theoretical outcome: we do not need to prove meta-theoretical properties of our constructions, we can work in our type theory and use its logic as a formal system.

3.1 Remark (Theory *vs.* meta-theory). The constructions described in this article are *also* applicable in a setting where datatype definitions are *not* internalised: all our constructions could be justified at the meta-level and then be syntactically presented in a language, such as, say, Agda, Coq, or an ML with GADTs. Working with an internalised presentation, we can simply avoid these two levels of logic and work in the logic provided by type theory.

Our requirements on the ambient type theory are boxed in a TYPE THEORY frame (*e.g.*, Fig. 2, p.10), whilst constructions within that type theory are boxed in a DEFINITION frame (*e.g.*, Fig. 3, p.17). Because our work is grounded in an intensional reading of extensional isomorphisms, we box the original, extensional results in META-THEOREM frames (*e.g.*, Equation 4.20, p.24). The proof of these results are absent from our (intensional) Agda model. Examples illustrating the various concepts are left unboxed. ◊

3.1 The type theory

Following our previous work, our requirements on the type theory are minimal: we will need Σ-types, Π-types, the unit set $\mathbb{1}$, and at least two universes, SET and SET_1. Σ-types are denoted $(a{:}A) \times B$, introduced by pairs (x, y) and eliminated by first and second projections, respectively π_0 and π_1. Π-types are denoted $(a{:}A) \to B$, introduced by $\lambda x.\, b$ and eliminated by function application. The unit set is (uniquely) inhabited by $*$. For convenience, we require the η-laws for the unit set, Σ-types (*i.e.* surjective pairing), and Π-types to hold definitionally. When x is not free in B, we use the following abbreviations:

$$(x{:}A) \to B \triangleq A \to B$$
$$(x{:}A) \times B \triangleq A \times B$$

Hence obtaining the (non dependent) implication and conjunction from the dependent quantifiers.

For convenience, we ask for our type theory to support *enumerations* of tags. We shall not dwell on their type theoretic definition, which can be found elsewhere (Dagand, 2013). We declare a (finite) enumeration of tags `'a`, `'b`, and `'c` by writing $\{\text{'a}\,\text{'b}\,\text{'c}\}$. Similarly, we define functions from such a collection by exhaustively enumerating the returned values, using the notation

$$\{\text{'a} \mapsto e_a, \text{'b} \mapsto e_b, \text{'c} \mapsto e_c\}$$

When the cases are vertically aligned, we shall skip the separating commas.

We also need a pre-existing notion of propositional equality, denoted $- = -$. We shall assume that it is reflexive, as witnessed by refl, and substitutive. Because we perform numerous pattern-matches on

TYPE THEORY

$$\begin{array}{ll}
\textbf{data}\ \mathsf{IDesc}\,[I:\mathrm{SET}]:\mathrm{SET}_1\ \textbf{where} & [\![(D:\mathsf{IDesc}\,I)]\!]\,(X:I\to\mathrm{SET}):\mathrm{SET} \\
\quad \mathsf{IDesc}\,I \ni \mathsf{var}\,(i:I) & [\![\mathsf{var}\,i]\!]\ X \mapsto X\,i \\
\quad\quad \mid 1 & [\![1]\!]\ X \mapsto \mathbb{1} \\
\quad\quad \mid \Pi\,(S:\mathrm{SET})\,(T:S\to\mathsf{IDesc}\,I) & [\![\Pi\,S\,T]\!]\ X \mapsto (s:S)\to[\![T\,s]\!]\,X \\
\quad\quad \mid \Sigma\,(S:\mathrm{SET})\,(T:S\to\mathsf{IDesc}\,I) & [\![\Sigma\,S\,T]\!]\ X \mapsto (s:S)\times[\![T\,s]\!]\,X
\end{array}$$

Fig. 2: Universe of inductive families

indexed datatypes, we also require our equality to satisfy the K rule (McBride, 1999), *i.e.* refl is the unique inhabitants of a propositional equality.

3.2 Universe of descriptions

We internalise inductive families by a universe construction. The role of this universe is to *describe* signature functors over SET indexed by I, *i.e.* functors from SET^I to SET^I. However, up to some currying-uncurrying, this type is subject to the isomorphism

$$\begin{aligned}
\mathrm{SET}^I\to\mathrm{SET}^I &\triangleq (I\to\mathrm{SET})\to(I\to\mathrm{SET}) \\
&\cong I\to(I\to\mathrm{SET})\to\mathrm{SET}
\end{aligned}$$

that lets us focus on describing functors from $\mathrm{SET}^I\to\mathrm{SET}$, with the I-indexing being pulled away in the exponential. Following this remark, we focus first on describing signature functors of type $\mathrm{SET}^I\to\mathrm{SET}$ (Definition 3.2). Then, to capture signature functors between slices of SET, we simply introduce an exponential (Definition 3.3), a construction akin to the `Reader` monad.

3.2 Definition (Universe of descriptions [2])**.** The universe of descriptions is defined in Fig. 2. The meaning of codes – the inhabitants of IDesc – is given by their interpretation in SET:

- Σ codes Σ-types – to build sums-of-products;
- Π codes Π-types – to capture higher-order arguments;
- 1 codes the unit type – to terminate codes;
- var codes the recursive arguments of inductive definitions, taken at an index i.

▽

3.3 Definition (Descriptions [2])**.** We obtain the universe of descriptions func by simply pulling the I-index to the front. The interpretation $[\![-]\!]$ extends pointwise to func:

TYPE THEORY

$$\begin{array}{ll}
\mathsf{func}\,(I:\mathrm{SET}):\mathrm{SET}_1 & [\![(D:\mathsf{func}\,I)]\!]\,(X:I\to\mathrm{SET}):I\to\mathrm{SET} \\
\mathsf{func}\,I\mapsto I\to\mathsf{IDesc}\,I & [\![D]\!]\ X \mapsto \lambda i.\,[\![D\,i]\!]\,X
\end{array}$$

[2] MODEL: `IDesc.IDesc`

Because it describes functors, this universe offers a generic map operator:

TYPE THEORY

$$[\![(D : \mathsf{func}\, I)]\!]^{\to} : (f : \forall i.\, X\, i \to Y\, i)(xs : [\![D]\!]\, X\, i) \to [\![D]\!]\, Y\, i$$

Inhabitants of the func type are called *descriptions*.

▽

3.4 Remark (Overloaded notation). We overload the symbol $[\![-]\!]$ to denote both the interpretation of a description code to a functor from $\textsc{Set}^I$ to $\textsc{Set}$ (Definition 3.2), and the interpretation of a description to an endofunctor from $\textsc{Set}^I$ to $\textsc{Set}^I$ (Definition 3.3). Indeed, the latter is merely of pointwise lifting of the former.

◊

Descriptions, by interpreting to strictly positive functors on slices of $\textsc{Set}$, admit a least fixpoint[3] construction:

TYPE THEORY

$$\begin{array}{l}\textbf{data}\ \mu\ [D : \mathsf{func}\, I](i : I) : \textsc{Set}\ \textbf{where}\\ \quad \mu\, D\, i \ni \mathsf{in}\, (xs : [\![D]\!]\, (\mu\, D)\, i)\end{array}$$

The inductive types thus formed are eliminated by a generic elimination principle[4]:

TYPE THEORY

$$\begin{array}{l}\mathsf{induction} : \forall P : \forall i : I.\, \mu\, D\, i \to \textsc{Set}.\\ \qquad (\alpha : (i : I)(xs : [\![D]\!]\, (\mu\, D)\, i) \to \Box_D\, P\, xs \to P\, (\mathsf{in}\, xs))\\ \qquad (x : \mu\, D\, i) \to P\, x\end{array}$$

that corresponds to transfinite induction over tree-like structures. A formal description of induction can be found elsewhere (Dagand, 2013). Intuitively, $\Box_D\, P\, xs$ asserts that all sub-trees of xs satisfy P: this captures precisely the inductive hypothesis. The argument α therefore corresponds to the inductive step: given that the induction hypothesis holds for sub-elements of xs, we must prove that P holds for the whole. In the categorical literature (Hermida & Jacobs, 1998; Fumex, 2012), $\Box_D$ is called the canonical lifting[5] of D.

Well-founded recursive definitions by pattern-matching can be expressed in terms of the induction principle (McBride, 2002). For conciseness, we adopt a pattern-matching style when the recursive pattern is unsurprising, with the confidence that it can be expressed in terms of to induction.

[3] MODEL: `IDesc.IDesc.InitialAlgebra`
[4] MODEL: `IDesc.IDesc.Induction`
[5] MODEL: `IDesc.IDesc.Lifting`

3.3 Inductive definitions

Whilst we could *code* our inductive families directly in this universe, let us introduce an *informal notation* to declare datatypes. Our purpose here is to relate the reader's intuition for inductive definitions with our encodings. By relying on a more intuitive notation, we wish to make our examples more palatable. Our notation is strongly inspired by Agda's datatype declarations.

As witnessed by the Agda model, all the inductive definitions given in this article can be translated into descriptions. For a datatype T, we write T-func the code it elaborates to. Similarly, we call T-elim the elimination principle of T, and T-case the case analysis over T. These operations can be reduced generically derived from induction (McBride *et al.*, 2004; Dagand & McBride, 2013b).

We also write datatype constructors (in expressions) and constructor patterns (in pattern-matching clauses) using the high-level notation. In fact, they correspond to (low-level) terms built from the generic in constructor and a tuple of arguments. This elaboration mechanism is described elsewhere (Dagand, 2013).

3.5 Example. For Peano numbers (*i.e.* natural numbers Nat), these induction principles amount to – after suitable currying and simplification – the propositions:

$$\mathsf{Nat\text{-}elim} : \forall P : \mathsf{Nat} \to \mathrm{SET}.\, P\,0 \to ((m : \mathsf{Nat}) \to P\,m \to P\,(\mathsf{suc}\,m)) \to (n : \mathsf{Nat}) \to P\,n$$
$$\mathsf{Nat\text{-}case} : \forall P : \mathsf{Nat} \to \mathrm{SET}.\, P\,0 \to ((m : \mathsf{Nat}) \to P\,(\mathsf{suc}\,m)) \to (n : \mathsf{Nat}) \to P\,n$$

△

A formal presentation of the elaboration of inductive definitions to code will be found elsewhere (Dagand & McBride, 2013b). However, it is intuitive enough to be understood with a few examples. Three key ideas are at play:

- Constructors are presented as sums of products, à la ML (Example 3.6);
- Indices can be constrained by equality, à la Agda (Example 3.7);
- Indices can be matched upon (Examples 3.8).

3.6 Example (Sums of products, following the ML tradition)**.** We name the datatype and then comes a choice of constructors. Each constructor is then defined by a Σ-telescope of arguments. For example, the list datatype[6] is defined by

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

$$\vdots$$

$$\begin{array}{l}\mathsf{List\text{-}func}\,(A : \mathrm{SET}) : \mathsf{func}\,\mathbb{1}\\ \mathsf{List\text{-}func}\,A \mapsto \lambda *.\, \Sigma \left\{ \begin{array}{l} \text{'nil} \\ \text{'cons} \end{array} \right\} \left\{ \begin{array}{ll} \text{'nil} & \mapsto 1 \\ \text{'cons} & \mapsto \Sigma A\, \lambda \_ .\, \mathsf{var}\, * \end{array} \right\}\end{array}$$

[6] MODEL: `IDesc.Examples.List`

Ordinals[7] also follow this pattern:

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

$$\wr$$

$$\begin{array}{l}\mathsf{Ord\text{-}func} : \mathsf{func}\,\mathbb{1}\\ \mathsf{Ord\text{-}func} \mapsto \lambda *.\,\Sigma \left\{ \begin{array}{l} \text{'0} \\ \text{'suc} \\ \text{'lim} \end{array} \right\} \left\{ \begin{array}{l} \text{'0} \mapsto 1 \\ \text{'suc} \mapsto \mathsf{var}\,* \\ \text{'lim} \mapsto \Pi\,\mathsf{Nat}\,\lambda\_.\,\mathsf{var}\,* \end{array} \right\}\end{array}$$

Note the use of the higher-order Π code to express the limit ordinal.

The datatypes Bool – the Booleans[8] – and Nat – the natural numbers[9] – fall in this category too. They elaborate to descriptions indexed by $\mathbb{1}$, respectively Bool-func and Nat-func, following a similar sums-of-products pattern. We leave it as an exercise to compute their code, guided by the following remarks:

- Nat-func is a degenerate case of List-func: it is a list taking no A-argument;
- Bool-func is a degenerate case of Nat-func: it offers two constructors, but no recursive argument.

△

3.7 Example (Indexing, following the Agda convention)**.** Indices can be *constrained* to some particular value. For example, vectors can be defined by constraining the index to be 0 in the nil case and $\mathsf{suc}\,n'$ for some $n' : \mathsf{Nat}$ in the cons case[10]:

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

$$\wr$$

$$\begin{array}{l}\mathsf{Vec\text{-}func}\,(A : \textsc{Set}) : \mathsf{func}\,\mathsf{Nat}\\ \mathsf{Vec\text{-}func}\,A \mapsto \lambda n.\,\Sigma \left\{ \begin{array}{l} \text{'nil} \\ \text{'cons} \end{array} \right\} \left\{ \begin{array}{l} \text{'nil} \mapsto \Sigma\,(n = 0)\,\lambda\_.\,1 \\ \text{'cons} \mapsto \Sigma\,\mathsf{Nat}\,\lambda n'.\,\Sigma\,(n = \mathsf{suc}\,n')\lambda\_.\,\Sigma A\,\lambda\_.\,\mathsf{var}\,n' \end{array} \right\}\end{array}$$

The constraint notation $(n = t)$ reads "for any index n, as long as n equals t", following the Henry Ford principle (McBride, 1999). In particular, it must *not* be confused with a definition pattern-matching on the index.

[7] MODEL: `IDesc.Examples.Ordinal`
[8] MODEL: `IDesc.Examples.Bool`
[9] MODEL: `IDesc.Examples.Nat`
[10] MODEL: `IDesc.Examples.Vec`

In the same vein, finite sets[11] can be defined by constraining the upper-bound n to always be strictly positive, and indexing the argument of fsuc by the predecessor:

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

⋮

Fin-func : func Nat
Fin-func ↦ λn. Σ { 'f0   } { 'f0   ↦ Σ Nat λn'. Σ (n = suc n') λ _ . 1     }
                 { 'fsuc } { 'fsuc ↦ Σ Nat λn'. Σ (n = suc n') λ _ . var n' }
```

△

Note that elaboration captures the constraints on indices by using propositional equality. In the case of Vec, we first abstract over the index n, introduce the choice of constructors with the first Σ and then, once the constructors have been chosen, we restrict n to its possible value(s): 0 in the first case and $\mathsf{suc}\,n'$ for some n' in the second case. Hence the placement of the equality constraints in the elaborated code: after the constructor is chosen, we first introduce a fresh variable and then constrain the index with it. If no fresh variable needs to be introduced, we directly constrain the index.

3.8 Example (Computing over indices)**.** We can also use the crucial property that a datatype definition is, in effect, a *function* from its indices to a choice of datatype constructors. Our notation should reflect this ability. For instance, inspired by Brady *et al.* (2003), we give an alternative presentation of vectors that matches on the index to determine the constructor to be presented[12], hence removing the need for constraints:

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

⋮

Vec-func (A : SET) : func Nat
Vec-func   A   n     ⇐ Nat-case n
  Vec-func A   0     ↦ Σ {'nil} λ _ . 1
  Vec-func A (suc n) ↦ Σ {'cons} λ _ . Σ A λ _ . var n
```

In order to be fully explicit about computations, we use the "by" ($\Leftarrow$) gadget, which lets us appeal to any elimination principle. For simplicity, we shall use a pattern-matching style when the recursion pattern is unremarkable.

[11] MODEL: `IDesc.Examples.Fin`
[12] MODEL: `IDesc.Examples.Vec`

Using pattern-matching, we define the computational counterpart of finite sets by matching on n[13], offering no constructor in the 0 case, and the two expected constructors in the suc n case:

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

  ;

Fin-func : func Nat
Fin-func   0     ↦ Σ 𝟘 𝟘-elim
Fin-func (suc n) ↦ Σ { 'f0   } { 'f0   ↦ 1     }
                     { 'fsuc } { 'fsuc ↦ var n }
```

Note the pattern used here to provide *no* constructor when the index is 0: we ask for a witness of the empty set, effectively preventing any constructor to be introduced.

△

3.9 Remark (Forcing and detagging (Brady *et al.*, 2003)). This technique of extracting information by case analysis on the indices applies to descriptions exactly where Brady's *forcing* and *detagging* optimisations apply in compilation. They eliminate just those constructors, indices and constraints which are redundant even in *open* computation.

Detagging amounts to restricting the choice of constructors by matching on the index. For detagging to apply, constructors must be in injective correspondence with the indices. Our presentation of vectors above is obtained by detagging. By noticing whether the index is 0 or suc, we deduce the vector's constructor.

Forcing amounts to computing the argument $x : X$ of a constructor from its index $i : I$. Hence, for forcing to apply, we must have a function f – ideally, the identity – from I to X such that $f\, i \mapsto x$. Our alternative presentation of Fin above is obtained by forcing: instead of storing an index n', we pattern-match on the index n and directly use its predecessor in the recursive argument.

◊

This last definition style departs radically from the one adopted by Coq, Agda, or GADTs. While it is possible to *encode* these definitions in Coq and Agda (through a large elimination), we have to step outside the realm of inductive definitions. Doing so, we lose access to the system's machinery for handling inductive reasoning. For instance, one would have to manually provide an elimination principle for the resulting object.

It is crucial to understand that this notation is but reflecting the actual semantics of inductive families as functors of type $I \to (I \to \text{SET}) \to \text{SET}$. By using the function space $I \to -$ to its full potential, we can *compute* over indices, not merely constrain them. With our syntax, we give the user the ability to write these *functions*: the reader should now understand a datatype definition as a special kind of function definition, taking indices as arguments, potentially computing over them, and eventually emitting a choice of constructors.

The original definition of ornaments was based on a universe following the Agda convention, which could only capture the indexing disciplines through equality constraints. Our ability to compute over

[13] MODEL: `IDesc.Examples.Fin`

indices has far-reaching consequences on ornaments. First, it enables the definition of a novel *deletion* ornament (Section 4), which uses the indexing information to delete redundant arguments in an ornamented datatype. Second, it enables a better structured and, consequently, more space-efficient definition of the algebraic ornament by the ornamental algebra (Section 4.5).

3.10 Example. We can sensibly mix these definition styles. An example that benefits from this approach is the presentation of minimal logic[14] – *i.e.*, from the other side of Curry-Howard, the simply-typed lambda calculus (Benton *et al.*, 2012) – given as an inductively-defined inference system. We express the judgement $\Gamma \vdash T$ through an inductive family indexed by a context Γ of typed variables and a type T:

$$\begin{array}{l}\textbf{data}\ (\Gamma : \mathsf{Context}) \vdash (T : \mathsf{Type}) : \textsc{Set}\ \textbf{where} \\ \quad \Gamma \vdash \quad T \quad \ni \mathsf{var}\,(v : T \in \Gamma) \\ \qquad\qquad\quad \mid \mathsf{app}\,(S : \mathsf{Type})(f : \Gamma \vdash S \Rightarrow T)(s : \Gamma \vdash S) \\ \quad \Gamma \vdash \ \mathsf{unit} \ \ni * \\ \quad \Gamma \vdash S \Rightarrow T \ni \mathsf{lam}\,(b : \Gamma ; S \vdash T)\end{array}$$

where, for simplicity, we have restricted the language of types to the unit and the exponential:

$$\begin{array}{l}\textbf{data}\ \mathsf{Type} : \textsc{Set}\ \textbf{where} \\ \quad \mathsf{Type} \ni \mathsf{unit} \\ \qquad \mid (S : \mathsf{Type}) \Rightarrow (T : \mathsf{Type})\end{array} \qquad \begin{array}{l}\textbf{data}\ \mathsf{Context} : \textsc{Set}\ \textbf{where} \\ \quad \mathsf{Context} \ni \varepsilon \\ \qquad \mid (\Gamma : \mathsf{Context}) ; (T : \mathsf{Type})\end{array}$$

and for which we can define (inductively, in fact) a predicate $T \in \Gamma$ that indexes a variable of type T in context Γ.

Crucially, the variable and application rules take the index *as is*, without constraint or computation. The remaining rules depends on the index: if it is an exponential, we give the abstraction rule; if it is the unit type, we give the (only) inhabitant of that type.

△

3.11 Remark (Constraints and equality). We have been careful in using equality to *introduce* constraints here: our definition of datatypes is absolutely agnostic in the notion of propositional equality offered by the underlying type theory. For instance, our universe of inductive families cannot be used to *define* equality through the identity type: the identity type would only *expose* the underlying notion of equality to the user.

This is unlike systems such as Coq or Agda, where propositional equality is introduced by the identity type

$$\begin{array}{l}\textbf{data}\ \mathsf{Id}\,[a_1 : A](a_2 : A) : \textsc{Set}\ \textbf{where} \\ \quad \mathsf{Id}\,a_1\,(a_2 = a_1) \ni \mathsf{refl}\end{array}$$

whose elimination principle gives the J-rule (Hofmann & Streicher, 1994).

In our system, this definition[15] would elaborate to a description *packaging* the propositional equality:

$$\begin{array}{lll}\mathsf{Id\text{-}func}\,(a_1 : A) & : & \mathsf{func}\,A \\ \mathsf{Id\text{-}func} \quad a_1 & \mapsto & \lambda a_2.\,\Sigma\,\{\text{'refl}\}\{\Sigma\,(a_2 = a_1)\,\lambda - .\,1\}\end{array}$$

◊

[14] MODEL: `IDesc.Examples.STLC`
[15] MODEL: `IDesc.Examples.Id`

DEFINITION

$$
\begin{array}{l}
\textbf{data}\ \mathsf{Orn}\,(D : \mathsf{IDesc}\,K)[u : I \to K] : \textsc{Set}_1\ \textbf{where} \\
\quad \textit{– Extend with S:} \\
\quad \mathsf{Orn}\quad D\quad u\quad \ni\quad \mathsf{insert}\,(S : \textsc{Set})(D^+ : S \to \mathsf{Orn}\,D\,u) \\
\quad \textit{– Refine index:} \\
\quad \mathsf{Orn}\ (\mathsf{var}\,k)\ u\quad \ni\quad \mathsf{var}\,(i : u^{-1}\,k) \\
\quad \textit{– Copy the original:} \\
\quad \mathsf{Orn}\quad 1\quad u\quad \ni\quad 1 \\
\quad \mathsf{Orn}\,(\Pi\,S\,T)\,u\quad \ni\quad \Pi\,(T^+ : (s : S) \to \mathsf{Orn}\,(T\,s)\,u) \\
\quad \mathsf{Orn}\,(\Sigma\,S\,T)\,u\quad \ni\quad \Sigma\,(T^+ : (s : S) \to \mathsf{Orn}\,(T\,s)\,u) \\
\quad \textit{– Delete S:} \\
\qquad\qquad\quad |\quad \mathsf{delete}\,(s : S)(T^+ : \mathsf{Orn}\,(T\,s)\,u)
\end{array}
$$

$$
\begin{array}{lcl}
[\![(O : \mathsf{Orn}\,D\,u)]\!]_{\mathsf{orn}} & : & \mathsf{IDesc}\,I \\
[\![\mathsf{insert}\,S\,D^+]\!]_{\mathsf{orn}} & \mapsto & \Sigma\,S\,\lambda s.\,[\![D^+\,s]\!]_{\mathsf{orn}} \\
[\![\mathsf{var}\,(\mathsf{inv}\,i)]\!]_{\mathsf{orn}} & \mapsto & \mathsf{var}\,i \\
[\![1]\!]_{\mathsf{orn}} & \mapsto & 1 \\
[\![\Pi\,T^+]\!]_{\mathsf{orn}} & \mapsto & \Pi\,S\,\lambda s.\,[\![T^+\,s]\!]_{\mathsf{orn}} \\
[\![\Sigma\,T^+]\!]_{\mathsf{orn}} & \mapsto & \Sigma\,S\,\lambda s.\,[\![T^+\,s]\!]_{\mathsf{orn}} \\
[\![\mathsf{delete}\,s\,T^+]\!]_{\mathsf{orn}} & \mapsto & [\![T^+]\!]_{\mathsf{orn}}
\end{array}
$$

Fig. 3: Universe of ornaments

4 A Universe of Ornaments

Originally, McBride (2013) presented the notion of ornament for a universe where the indexing discipline could *only* be enforced by equality constraints. As a result, the deletion ornament was not expressible in that setting. We shall now adapt the original definition to our system.

4.1 Definition (Universe of ornaments[16])**.** The grammar of ornaments (Fig. 3) is similar to the original one. It is defined over a base datatype D indexed by K and ornaments it to a datatype indexed by I. The (forgetful) function $u : I \to K$ specifies a refinement of the K-indices into I-indices. We can *copy* the base datatype (with the codes 1, Π, and Σ), *extend* it by inserting sets (with the code insert), and *refine* the indexing subject to the relation imposed by u (with the code var). Also, following Brady's insight that *inductive families need not store their indices* (Brady *et al.*, 2003), we can delete parts of the datatype definition as long as we can provide a witness. This witness will typically be provided by the index, here in the context.

The extension of ornaments computes the description of the extended datatype. This amounts to traversing the ornament code, packing the freshly insert-ed data into Σ codes. In the delete case, no Σ code is generated: we use the witness to compute the extension of the rest of the ornament. The Π and Σ ornament codes simply duplicate the underlying datatype definition: we retrieve the set S from the index ornament D (which is equal to $\Sigma\,S\,T$ for a Σ ornament, and to $\Pi\,S\,T$ for a Π ornament).

▽

4.2 Remark (Inverse image[17])*.* The inverse of a function f is defined by the following inductive type

DEFINITION

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

[16] MODEL: `Orn.Ornament`
[17] MODEL: `Logic.Logic`

Equivalently, it can be defined with a Σ-type:

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

◊

4.3 Definition (Ornament[18])**.** An ornament is defined upon a base datatype – specified by a description $D : \mathsf{func}\,K$ – and a refined set of indices – specified by a function $u : I \to K$. The ornament of D is an *I-family* of ornament codes for each $D\,(u\,i)$, with $i : I$:

DEFINITION

$$\begin{array}{ll}\mathsf{orn}\,(D : \mathsf{func}\,K)\,(u : I \to K) : \mathrm{SET}_1 & [\![(o : \mathsf{orn}\,D\,u)]\!]_{\mathsf{orn}} : \mathsf{func}\,I \\ \mathsf{orn}\,D\,u \mapsto (i : I) \to \mathsf{Orn}\,(D\,(u\,i))\,u & [\![o]\!]_{\mathsf{orn}} \mapsto \lambda i.\,[\![o\,i]\!]_{\mathsf{orn}}\end{array}$$

In effect, ornamenting a description func consists merely in *lifting* the ornamentation of IDesc codes to a family indexed by I.

▽

4.4 Remark (Overloaded notation). We overload the symbol $[\![-]\!]_{\mathsf{orn}}$ to denote both the interpretation of an ornament code to a description code (Definition 4.1), and the interpretation of an ornament to a description (Definition 4.3). As for descriptions (Remark 3.4), the latter is merely of pointwise lifting of the former.

◊

4.1 Notation

As for inductive definitions, we adopt an *informal notation* to succinctly define ornaments. The idea is to simply mirror our **data** definition, adding **from** which datatype the ornament builds upon. When specifying a constructor, we can then extend it with new information – using $[x : S]$ – or delete an argument originally named x by providing a witness – using $[x \triangleq s]$. We require the order of constructors to be preserved across ornamentation, as their name might change from the original to the ornamented version.

This high-level notation enables us to succinctly specify ornaments. It provides an abstraction over the code of ornament, in the same manner that inductive definitions let us abstract over the code of description (Section 3.3). From the definition of an ornamented type T, we conventionally call T-Orn its ornament code. A formal description of the translation is beyond the scope of this article. Nonetheless, a few examples are enough to illustrate our notation and shall help us build some intuition for ornaments.

In an effort to reduce the syntactic noise, our notation for ornaments does not specify the forgetful function relating the indices of the ornamented type to its base type. For the examples given in this article, these functions can be inferred from the context. If in doubt, the reader should consult the corresponding definitions in the Agda model. In an actual implementation, the user would have to provide this information.

[18] MODEL: `Orn.Ornament`

4.5 Example (Ornament: from Booleans to the option type[19])**.** We obtain the option type from the Booleans by inserting an extra $a : A$ in the true case:

$$\begin{array}{l}\textbf{data}\ \mathsf{Maybe}\,[A : \textsc{Set}]\ \textbf{from}\ \mathsf{Bool}\ \textbf{where}\\ \quad \mathsf{Maybe}\,A \ni \mathsf{just}\,[a : A]\\ \qquad\qquad \mid \mathsf{nothing}\end{array}$$

$$\wr$$

$$\begin{array}{l}\mathsf{Maybe\text{-}Orn}\,(A : \textsc{Set}) \ :\ \mathsf{orn}\,\mathsf{Bool\text{-}func}\,\mathsf{id}\\ \mathsf{Maybe\text{-}Orn}\quad A \quad \mapsto \lambda *.\,\mathsf{insert}\left\{\begin{array}{l}\text{'just}\\ \text{'nothing}\end{array}\right\}\left\{\begin{array}{ll}\text{'just} & \mapsto \mathsf{delete}\,\text{'true}\,(\mathsf{insert}\,A\,\lambda\_.\,1)\\ \text{'nothing} & \mapsto \mathsf{delete}\,\text{'false}\,1\end{array}\right\}\end{array}$$

We leave it to the reader to verify that the interpretation of this ornament (by $[\![-]\!]_{\mathsf{orn}}$) followed by the interpretation of the resulting description (by $[\![-]\!]$) yields the signature functor of the option type $X \mapsto 1 + A$:

$$[\![[\![\mathsf{Maybe\text{-}Orn}\,A]\!]_{\mathsf{orn}}]\!]\,X \cong 1 + A$$

△

4.6 Remark (Notation). To reduce the notational burden, we overload the interpretation of ornaments $[\![-]\!]_{\mathsf{orn}}$ to denote both the description and the interpretation of that description. For instance, we write the above isomorphism as follows:

$$[\![\mathsf{Maybe\text{-}Orn}\,A]\!]_{\mathsf{orn}}\,X \cong 1 + A$$

◊

4.7 Example (Ornamenting natural numbers to lists[20])**.** We obtain lists from natural numbers with the following ornament:

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

$$\wr$$

$$\begin{array}{l}\mathsf{List\text{-}Orn}\,(A : \textsc{Set}) \ :\ \mathsf{orn}\,\mathsf{Nat\text{-}func}\,\mathsf{id}\\ \mathsf{List\text{-}Orn}\quad A \quad \mapsto \lambda *.\,\mathsf{insert}\left\{\begin{array}{l}\text{'nil}\\ \text{'cons}\end{array}\right\}\left\{\begin{array}{ll}\text{'nil} & \mapsto \mathsf{delete}\,\text{'0}\,1\\ \text{'cons} & \mapsto \mathsf{delete}\,\text{'suc}\,(\mathsf{insert}\,A\,\lambda\_.\,\mathsf{var}\,(\mathsf{inv}\,*))\end{array}\right\}\end{array}$$

Unfolding the interpretations, we check that we obtain the signature functor of lists $X \mapsto 1 + A \times X$:

$$[\![\mathsf{List\text{-}Orn}\,A]\!]_{\mathsf{orn}}\,X \cong 1 + A \times X$$

△

[19] MODEL: `Orn.Ornament.Examples.Maybe`
[20] MODEL: `Orn.Ornament.Examples.List`

4.8 Example (Ornamenting natural numbers to finite sets[21])**.** 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':

$$\begin{array}{l}\textbf{data}\ \mathsf{Fin}\,(n:\mathsf{Nat})\ \textbf{from}\ \mathsf{Nat}\ \textbf{where}\\ \quad \mathsf{Fin}\, n \ni \mathsf{f0}\,[n':\mathsf{Nat}][q:n=\mathsf{suc}\, n'] \\ \quad\quad \mid\ \mathsf{fsuc}\,[n':\mathsf{Nat}][q:n=\mathsf{suc}\, n'](k:\mathsf{Fin}\, n')\end{array}$$

$$\wr$$

$$\begin{array}{l}\mathsf{Fin\text{-}Orn}\ :\ \mathsf{orn}\,\mathsf{Nat\text{-}func}\,(\lambda n.*)\\ \mathsf{Fin\text{-}Orn} \mapsto \lambda n.\,\mathsf{insert}\left\{\begin{array}{l}\text{'f0}\\ \text{'fsuc}\end{array}\right\}\left\{\begin{array}{ll}\text{'f0} & \mapsto \mathsf{delete}\ \text{'0}\\ & \quad (\mathsf{insert}\,\mathsf{Nat}\,\lambda n'.\,\mathsf{insert}\,(n=\mathsf{suc}\, n')\,\lambda - .\,1)\\ \text{'fsuc} & \mapsto \mathsf{delete}\ \text{'suc}\\ & \quad (\mathsf{insert}\,\mathsf{Nat}\,\lambda n'.\,\mathsf{insert}\,(n=\mathsf{suc}\, n')\,\lambda - .\,\mathsf{var}\,(\mathsf{inv}\, n'))\end{array}\right\}\end{array}$$

Again, we leave it as an exercise to unfold the interpretations of this ornament and verify that it is indeed describing the signature of finite sets.

△

4.9 Example (Identity ornament[22])**.** In Section 2, we have introduced the *identity ornament* idO as the (trivial) ornament that merely duplicates the definition of its base type. This construction is a straightforward generic program over description codes:

$$\begin{array}{lll}\mathsf{idO}\,(D:\mathsf{IDesc}\, I) & : & \mathsf{Orn}\, D\,\mathsf{id}\\ \mathsf{idO}\quad (\mathsf{var}\, i) & \mapsto & \mathsf{var}\,(\mathsf{inv}\, i)\\ \mathsf{idO}\quad 1 & \mapsto & 1\\ \mathsf{idO}\quad (\Pi S\, T) & \mapsto & \Pi\lambda s.\,\mathsf{idO}\,(T\, s)\\ \mathsf{idO}\quad (\Sigma S\, T) & \mapsto & \Sigma\lambda s.\,\mathsf{idO}\,(T\, s)\end{array}$$

which lifts then pointwise to ornaments:

$$\mathsf{idO} : (D:\mathsf{func}\, I) \to \mathsf{orn}\, D\,\mathsf{id}$$

△

In Section 3.3, we had to adapt the original presentation of ornaments to our universe. In the process, we have discovered a new ornamental operation, namely the "deletion ornament". In Section 4.2, we explore some of the possibilities offered by having such a code in our system. However, we shall also verify that the ornamental constructions presented in the original framework still apply: this shall be the topic of Section 4.3 – where we recast the *ornamental algebra* in our setting – and Section 4.4 – where we recast the *algebraic ornament* construction. Finally, we revisit the *algebraic ornament by the ornamental algebra* in Section 4.5, making crucial use of the deletion ornament and of the optimisations discussed in the following section.

[21] MODEL: `Orn.Ornament.Examples.Fin`
[22] MODEL: `Orn.Ornament.Identity`

4.2 Brady optimisations, internalised

In Example 3.8, we have seen an example of Brady's forcing and detagging optimisations, respectively on finite sets and vectors. We have explained how, thanks to our definition of descriptions as functions, we could (manually) craft datatypes in this form. Instead of a presentation based on constraints, we gave an equivalent but less redundant definition. Seen as a datatype transformation, this operation is (obviously) structure-preserving. In fact, these transformations are an instance of ornamentation. The key ingredient is the delete code that lets us delete parts of a definition, using a witness extracted from the index. We can therefore craft our own Brady-optimised datatypes by ornamentation and benefit from this optimisation as early as at type checking.

To illustrate this approach, we give an example of detagging (Example 4.10) and forcing (Example 4.11). To focus solely on the Brady optimisations, we define these ornaments on the naive indexed family we wish to optimise. In practice, we would *compose* (Dagand & McBride, 2013a) the ornament of natural numbers (Example 1.4 (p.2) for vectors, and Example 4.8 for finite sets) with these optimisations. The composition would directly give the optimised version of, respectively, vectors and finite sets, thus avoiding an unnecessary duplication of isomorphic datatypes.

4.10 Example (Detagging, by ornamentation[23])**.** The definition of vectors in Example 3.7 mirrors Agda's convention of constraining indices with equality. Our definition of ornaments lets us define a version of Vec that does not store its indices. Indeed, we can describe Vec by an ornament that matches the index n to determine which constructor to offer:

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

                        ≀

Vec'-Orn   (A : SET)          :  orn Vec-func id
  Vec'-Orn   A        0       ↦ insert {'nil} λ _ . delete 'nil (delete refl 1)
  Vec'-Orn   A      (suc n)   ↦ insert {'cons} λ _ .
                                delete 'cons (delete n (delete refl (Σ λ _ . var (inv n))))
```

Such a definition was unavailable in the original presentation of ornaments (McBride, 2013). We have internalised detagging: the constructors of the datatype are determined by the index.

△

4.11 Example (Forcing, by ornamentation[24])**.** The definition of finite sets given in Example 3.7 is also subject to an optimisation: by matching the index, we can avoid the duplication of n by deleting n' with the matched predecessor and trivialising the proofs. Hence, Fin can be further ornamented to the optimised

[23] MODEL: `Orn.Ornament.Examples.Vec`
[24] MODEL: `Orn.Ornament.Examples.Fin`

Fin', which makes crucial use of deletion:

$$\begin{array}{l}\textbf{data}\ \mathsf{Fin'}\ (n : \mathsf{Nat})\ \textbf{from}\ \mathsf{Fin}\ n\ \textbf{where}\\ \quad \mathsf{Fin'}\ \ 0 \ni [b : \mathbb{0}]\\ \quad \mathsf{Fin'}\ (\mathsf{suc}\ n) \ni \mathsf{f0}\ [n' \triangleq n]\\ \qquad\qquad\quad \mid \mathsf{fsuc}\ [n' \triangleq n]\ (k : \mathsf{Fin'}\ n')\end{array}$$

$$\wr$$

$$\begin{array}{l}\mathsf{Fin'\text{-}Orn} : \mathsf{orn}\ \mathsf{Fin\text{-}func}\ \mathsf{id}\\ \mathsf{Fin'\text{-}Orn}\ \ 0 \mapsto \mathsf{insert}\ \mathbb{0}\ \mathbb{0}\text{-}\mathsf{elim}\\ \mathsf{Fin'\text{-}Orn}\ (\mathsf{suc}\ n) \mapsto \Sigma \left\{\begin{array}{l}\text{`f0} \mapsto \mathsf{delete}\ n\ (\mathsf{delete}\ \mathsf{refl}\ 1)\\ \text{`fsuc} \mapsto \mathsf{delete}\ n\ (\mathsf{delete}\ \mathsf{refl}\ (\mathsf{var}\ (\mathsf{inv}\ n)))\end{array}\right\}\end{array}$$

Note that when n is 0, there is in fact no constructor: we insert the empty set $\mathbb{0}$ to account for the absence of constructor at this index.

Again, this definition was previously unavailable to us. We have internalised forcing: the content of the constructors – here n' – are retrieved from the index, instead of being needlessly duplicated.

△

4.12 Remark (Non-generic transformations). The above transformations are *ad-hoc*: we have to manually give the ornament that performs the detagging and/or forcing. Because of the higher-order nature of our universe of descriptions, we cannot analyse the link between the indices, and the constructor choice and the constructor's contents. To achieve this from within type theory, we would need a first-order language for describing (a sufficiently expressive fragment of) the function space $I \to \mathsf{IDesc}\ I$. This is left to future work.

◊

4.3 Ornamental algebra

Every ornament induces an *ornamental algebra* (McBride, 2013): an algebra that forgets the extra information introduced by the extensions, mapping the ornamented datatype back to its original form.

4.13 Definition (Cartesian morphism[25])**.** For an ornament $O : \mathsf{Orn}\ D\ u$, there is a function – actually, a natural transformation (Dagand & McBride, 2013a) – projecting the ornamented functor down to the non-ornamented one (Fig. 4). This function then lifts pointwise to ornaments:

DEFINITION

$$\mathsf{forgetNT} : (o : \mathsf{orn}\ D\ u) \to [\![o]\!]_{\mathsf{orn}}\ (X \circ u)\ i \to [\![D]\!]\ X\ (u\ i)$$

▽

[25] MODEL: `Orn.Ornament.CartesianMorphism`

DEFINITION

$$
\begin{array}{lll}
\mathsf{forgetNT}\ (O : \mathsf{Orn}\ D\ u)\ (xs : [\![O]\!]_{\mathsf{orn}}\ (X \circ u)) & : & [\![D]\!]\ X \\
\mathsf{forgetNT}\ (\mathsf{insert}\ S\ D^{+}) \quad (s, xs) & \mapsto & \mathsf{forgetNT}\ (D^{+}\ s)\ xs \\
\mathsf{forgetNT}\ (\mathsf{var}\ (\mathsf{inv}\ i)) \quad xs & \mapsto & xs \\
\mathsf{forgetNT}\ \quad 1 \quad\quad * & \mapsto & * \\
\mathsf{forgetNT}\ \quad (\Pi\ T^{+}) \quad\quad f & \mapsto & \lambda s.\ \mathsf{forgetNT}\ (T^{+}\ s)\ (f\ s) \\
\mathsf{forgetNT}\ \quad (\Sigma\ T^{+}) \quad (s, xs) & \mapsto & (s, \mathsf{forgetNT}\ (T^{+}\ s)\ xs) \\
\mathsf{forgetNT}\ (\mathsf{delete}\ s\ T^{+}) \quad xs & \mapsto & (s, \mathsf{forgetNT}\ T^{+}\ xs)
\end{array}
$$

Fig. 4: Cartesian morphism

4.14 Definition (Ornamental algebra [26])**.** Applied with $\mu\ D$ for X and post-composed with the initial algebra in, this Cartesian morphism induces the ornamental algebra:

DEFINITION

$$
\begin{array}{lll}
\mathsf{forgetAlg}\ (o : \mathsf{orn}\ D\ u) & : & [\![o]\!]_{\mathsf{orn}}\ (\mu\ D \circ u)\ i \to \mu\ D\ (u\ i) \\
\mathsf{forgetAlg} \quad\quad o & \mapsto & \mathsf{in} \circ (\mathsf{forgetNT}\ o)
\end{array}
$$

▽

4.15 Definition (Forgetful map [26])**.** In turn, this algebra induces a forgetful map from the ornamented type to its original form:

DEFINITION

$$
\begin{array}{lll}
\mathsf{forget}\ (o : \mathsf{orn}\ D\ u) & : & \mu\ [\![o]\!]_{\mathsf{orn}}\ i \to \mu\ D\ (u\ i) \\
\mathsf{forget} \quad\quad o & \mapsto & (\!|\mathsf{forgetAlg}\ o|\!)
\end{array}
$$

where $(\!|(\alpha : \forall\ i.\ [\![D]\!]\ X\ i \to X\ i)|\!) : \forall\ i.\ \mu\ D\ i \to X\ i$ denotes the catamorphism, which can be implemented in terms of induction (Chapman *et al.*, 2010).

▽

4.16 Example (From lists back to natural numbers[27])**.** Applied to the ornament List-Orn, the Cartesian morphism removes the extra information added through insert, *i.e.* the inhabitant of A. The resulting algebra thus takes nil to 0, and cons a to suc. In turn, the forgetful map computes the length of the list. We have (automatically) constructed the length function from Section 2.

△

[26] MODEL: `Orn.Ornament.Algebra`
[27] MODEL: `Orn.Ornament.Examples.List`

4.17 Example (From the option type back to Booleans[28])**.** Applied to the ornament Maybe-Orn, the Cartesian morphism removes the $a:A$ we attached to the constructor true. The forgetful map corresponds exactly to the function isJust (Section 2).

△

4.18 Example (From finite sets back to natural numbers [29])**.** Applied to the ornament Fin-Orn, the Cartesian morphism removes the equations introduced by insert and forgets the indexing discipline enforced by the var code. The resulting forgetful map computes the cardinality – a natural number – of a finite set. It corresponds to the following function:

$$\begin{array}{lll} \mathsf{forget\,Fin\text{-}Orn}\ (k:\mathsf{Fin}\ n) & : & \mathsf{Nat} \\ \mathsf{forget\,Fin\text{-}Orn}\quad \mathsf{f0} & \mapsto & \mathsf{0} \\ \mathsf{forget\,Fin\text{-}Orn}\ \ (\mathsf{fsuc}\,k) & \mapsto & \mathsf{suc}\,(\mathsf{forget\,Fin\text{-}Orn}\,k) \end{array}$$

△

4.19 Example (From optimised finite sets to naïve finite sets [29])**.** When an ornament relies on a delete operation, the *forgetful* map has the – perhaps counter-intuitive – task to re-introduce the deleted information into the base datatype. To do so, it simply uses the information obtained from the index to fill-in the deleted arguments. For example, the forgetful map obtained from Fin'-Orn corresponds to the following function:

$$\begin{array}{lll} \mathsf{forget\,Fin'\text{-}Orn}\ (n:\mathsf{Nat})\ (k:\mathsf{Fin'}\ n) & : & \mathsf{Fin}\ n \\ \mathsf{forget\,Fin'\text{-}Orn}\ \ (\mathsf{suc}\,n)\quad \mathsf{f0} & \mapsto & \mathsf{f0}\,n\,\mathsf{refl} \\ \mathsf{forget\,Fin'\text{-}Orn}\ \ (\mathsf{suc}\,n)\ \ (\mathsf{fsuc}\,k) & \mapsto & \mathsf{fsuc}\,n\,\mathsf{refl}\,(\mathsf{forget\,Fin'\text{-}Orn}\,n\,k) \end{array}$$

where, to be explicit about the origin of the recovered data, we pass the index n as an explicit argument.

△

4.4 Algebraic ornaments

An important class of datatypes is constructed by *algebraic ornamentation* of a base datatype. An algebraic ornament[30] indexes an inductive type by the result of a catamorphism over its elements. From the code $D:\mathsf{func}\,K$ and an algebra $\alpha:\forall k.\,[\![D]\!]\,X\,k\to X\,k$, we define the algebraic ornament, denoted D^α, as the signature indexed by $(k:K)\times X\,k$ that satisfies the following coherence property:

META-THEOREM

For all $k:K$ and $x:\mu\,D\,k$, we have:

$$\mu\,[\![D^\alpha]\!]_{\mathsf{orn}}\,(k,x) \quad\cong\quad (t:\mu\,D\,k)\times(\!|\alpha|\!)\,t = x \tag{4.20}$$

Seen as a refinement type, this states that $\mu\,[\![D^\alpha]\!]_{\mathsf{orn}}\,(k,x)$ is an inductive definition equivalent to the refinement type[31] $\{t\in\mu\,D\,k\mid(\!|\alpha|\!)\,t=x\}$. A categorical presentation is given by Atkey *et al.* (2012), who explore the connection between refinement types and inductive families.

[28] MODEL: `Orn.Ornament.Examples.Maybe`
[29] MODEL: `Orn.Ornament.Examples.Fin`
[30] MODEL: `Orn.AlgebraicOrnament`
[31] Keeping with standard notations, we denote refinement types with a set comprehension. In type theory, this amounts to a Σ-type.

The type-theoretic construction of D^α was originally given by McBride (2013). We shall not reiterate it here, the implementation being essentially the same.The idea is to define – by ornamentation of D – a description whose fixpoint will satisfy the above coherence property.

4.21 Remark (Computational interpretation). Constructively, the coherence property (4.20) gives us two (mutually inverse) functions, $\mathsf{coherentOrn}$ and $\mathsf{make}\,D^\alpha$.

The direction $\mu\,[\![D^\alpha]\!]_{\mathsf{orn}}(k,x) \to (t : \mu\,D\,k) \times (\!|\alpha|\!)\,t = x$ relies on the generic forgetful map $\mathsf{forget}\,D^\alpha$ to compute the first component of the pair and gives us the following theorem[32]:

DEFINITION

$$\mathsf{coherentOrn} : (t^+ : \mu\,[\![D^\alpha]\!]_{\mathsf{orn}}(k,x)) \to (\!|\alpha|\!)\,(\mathsf{forget}\,D^\alpha\,t^+) = x$$

This corresponds to the $\mathsf{Recomputation}$ theorem of McBride (2013). We shall not reprove it here, the construction being similar.

In the other direction, the isomorphism (4.20) gives us a function of type

$$(t : \mu\,D\,k) \times (\!|\alpha|\!)\,t = x \to \mu\,[\![D^\alpha]\!]_{\mathsf{orn}}(k,x)$$

which, after simplifying the equation, gives a function that lifts a datatype to its algebraic version[33], at the index computed by the predicate:

DEFINITION

$$\mathsf{make}\,(D : \mathsf{func}\,I)^{(\alpha : \forall i.\,[\![D]\!]\,X\,i \to X\,i)} : (t : \mu\,D\,k) \to \mu\,[\![D^\alpha]\!]_{\mathsf{orn}}(k, (\!|\alpha|\!)\,t)$$

This corresponds to the $\mathsf{remember}$ function of McBride (2013). Again, we will assume this construction here.

◊

4.22 Example (Algebraic ornament: vectors)**.** Ornamenting natural numbers to lists, we obtain an ornamental algebra: the algebra computing the length of a list. We can therefore build the algebraic ornament of lists by the length algebra[34]. This corresponds *exactly* to the datatype of vectors (Example 3.7): the resulting signatures are isomorphic, and both rely on constraints to enforce the indexing discipline.

Note that this operation generalises to all ornaments: any ornament induces an ornamental algebra. Therefore, we can always build the algebraic ornament by the ornamental algebra. We shall study this operation in more details in Section 4.5.

△

[32] MODEL: `Orn.AlgebraicOrnament.Coherence`
[33] MODEL: `Orn.AlgebraicOrnament.Make`
[34] MODEL: `Orn.AlgebraicOrnament.Examples.Vec`

4.23 Example (Algebraic ornament: less-than-or-equal relation[35])**.** For a given natural number $m : \mathsf{Nat}$, the addition $m + - : \mathsf{Nat} \to \mathsf{Nat}$ is obtained by folding the algebra

$$\begin{array}{lllll} \mathsf{plusAlg}\ (m : \mathsf{Nat})\ (xs : [\![\mathsf{Nat\text{-}func}]\!]\ \mathsf{Nat}\ *) & : & \mathsf{Nat} \\ \mathsf{plusAlg} & m & (\text{'}0, *) & \mapsto & m \\ \mathsf{plusAlg} & m & (\text{'}\mathsf{suc}, n) & \mapsto & \mathsf{suc}\, n \end{array}$$

By algebraically ornamenting Nat by this algebra, we obtain the relation $m \leq - : \mathsf{Nat} \to \textsc{Set}$ that is characterised by the isomorphism

$$m \leq n \quad \cong \quad (k : \mathsf{Nat}) \times m + k = n$$

Put explicitly, the datatype computed by the algebraic ornament corresponds to

$$\begin{array}{l} \textbf{data}\ [m : \mathsf{Nat}] \leq (n : \mathsf{Nat}) : \textsc{Set}\ \textbf{where} \\ \quad m \leq (n = m) \quad \ni 0 \\ \quad m \leq (n = \mathsf{suc}\, n') \ \ni \mathsf{suc}\, (k : m \leq n') \end{array}$$

△

4.24 Example (Algebraic ornament: indexing by semantics[36])**.** A typical use-case of algebraic ornaments is the implementation of semantic-preserving operations (McBride, 2013). For example, let us consider arithmetic expressions, whose semantics is given by interpretation to Nat:

$$\begin{array}{l} \textbf{data}\ \mathsf{Expr} : \textsc{Set}\ \textbf{where} \\ \quad \mathsf{Expr} \ni \mathsf{const}\, (n : \mathsf{Nat}) \\ \qquad \mid \mathsf{add}\, (d : \mathsf{Expr})(e : \mathsf{Expr}) \end{array} \qquad \begin{array}{lllll} \mathsf{evalAlg}\ (es : [\![\mathsf{Expr\text{-}func}]\!]\ \mathsf{Nat}\ *) & : & \mathsf{Nat} \\ \mathsf{evalAlg} & (\text{'}\mathsf{const}, n) & \mapsto & n \\ \mathsf{evalAlg} & (\text{'}\mathsf{add}, (m, n)) & \mapsto & m + n \end{array}$$

Using the algebra $\mathsf{evalAlg}$, we construct the algebraic ornament of Expr and obtain expressions indexed by their semantics:

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

We can now enforce the preservation of semantics by typing. For example, let us optimise away all additions of the form "$0 + e$":

$$\begin{array}{lllll} \mathsf{optimise\text{-}0+}\ (e : \mathsf{Expr}^{\mathsf{evalAlg}}\, n) & : & \mathsf{Expr}^{\mathsf{evalAlg}}\, n \\ \mathsf{optimise\text{-}0+} \quad (\mathsf{const}\, n) & \mapsto & \mathsf{const}\, n \\ \mathsf{optimise\text{-}0+} \quad (\mathsf{add}\, 0\, n\, d\, e) & \mapsto & \mathsf{optimise\text{-}0+}\, e \\ \mathsf{optimise\text{-}0+}\ (\mathsf{add}\, (\mathsf{suc}\, m)\, n\, d\, e) & \mapsto & \mathsf{add}\, (\mathsf{suc}\, m)\, n\, d\, e \end{array}$$

Because the type checker accepts our definition, we have that, by construction, this operation preserves the semantics. We can then prune the semantics from the types using the forgetful map and retrieve the transformation on raw syntax trees. The make function (Remark 4.21) lets us lift raw syntax trees to semantically-indexed ones, while the $\mathsf{coherentOrn}$ theorem (Remark 4.21) certifies that the pruned tree satisfies the invariant we enforced by indexing.

△

[35] MODEL: `Orn.AlgebraicOrnament.Examples.Leq`
[36] MODEL: `Orn.AlgebraicOrnament.Examples.Expr`

4.5 Reornaments

In this article, we are particularly interested in a sub-class of algebraic ornaments. In Definition 4.14, we have constructed, for an ornament o, its ornamental algebra $\mathsf{forgetAlg}\,o$ that forgets the extra information introduced by the ornament. As hinted at in Example 4.22, given an ornament o, we can always algebraically ornament $[\![o]\!]_{\mathsf{orn}}$ using the ornamental algebra $\mathsf{forgetAlg}\,o$. McBride (2013) calls this construction the *algebraic ornament by the ornamental algebra*.

4.25 Remark (Notation). We write $\lceil o \rceil$ to denote the algebraic ornament of o by the ornamental algebra. For brevity, we call it the *reornament* of o.

◊

4.26 Example (Reornament: vectors)**.** Paraphrasing Example 4.22, we have that vectors are a reornament of List-Orn. Explicitly, a vector is the algebraic ornament of List by the algebra computing its length, *i.e.* the ornamental algebra from List to Nat.

△

4.27 Example (Reornament: indexed option type)**.** In Example 4.5, we ornamented Booleans to the option type. We can thus reornament the option type with its Boolean status. Unfolding the definition of the reornament, we obtain the $\mathsf{IMaybe}\,A$ datatype that was introduced in Section 2. The function forgetIMaybe corresponds to the left-to-right reading of the isomorphism (4.20) specialised to the Maybe ornament.

△

Reornaments are thus straightforwardly obtained through a two steps process: first, compute the ornamental algebra and, second, construct the algebraic ornament by this algebra. However, such a simplistic construction introduces a lot of spurious equality constraints and duplication of information. For instance, using this naive definition of reornaments, a vector indexed by n is constructed as *any* list *as long as* it is of length n.

4.28 Example (Reornamenting vectors, efficiently)**.** Let us consider the ornament List-Orn, taking natural numbers to lists. We gave its code in Example 4.7. Here, for simplicity, we shall work on the following variant

$$\begin{array}{lll} \mathsf{List\text{-}Orn}\,(A : \textsc{Set}) & : & \mathsf{orn}\,\mathsf{Nat\text{-}func}\,\mathsf{id} \\ \mathsf{List\text{-}Orn} \quad A & \mapsto & \lambda *.\,\Sigma \left\{ \begin{array}{ll} \text{'0} & \mapsto 1 \\ \text{'suc} & \mapsto \mathsf{insert}\,A\,\lambda \_ .\,\mathsf{var}\,(\mathsf{inv}\,*) \end{array} \right\} \end{array}$$

which does not update the constructor names, allowing us to focus on the essential transformations.

We can adopt a more fine-grained approach yielding an isomorphic but better structured datatype. In our setting, where we can compute over the index, a finer construction of the reornament of List-Orn is as follows:

- We retrieve the index, hence obtaining a number $n : \mathsf{Nat}$;
- By inspecting the ornament List-Orn, we obtain the *exact* relationship between the index n and its ornament describing lists
- If $n = 0$, we are in the first branch of the Σ code, and the ornamentation of n is necessarily the empty list. The corresponding reornament can therefore delete the choice of constructor (since it is entirely determined by the index), set it to '0, and terminate immediately:

$$\lceil \mathsf{List\text{-}Orn}\,A\,* \rceil\ (\text{'0}, *) \rightsquigarrow \mathsf{delete}\,\text{'0}\,1$$

DEFINITION

$$
\begin{array}{lll}
\lceil (O : \mathsf{Orn}\, D\, u) \rceil\ (xs : [\![D]\!]\ (\mu\, E)) & : & \mathsf{Orn}\, [\![O]\!]_{\mathsf{orn}}\ (\pi_0 : (i : I) \times \mu\, E\ (u\, i) \to I) \\
\lceil \mathsf{insert}\, S\, D^+ \rceil \quad xs & \mapsto & \Sigma \lambda s.\ \lceil D^+\ s \rceil\ xs \\
\lceil \mathsf{var}\ (\mathsf{inv}\, i) \rceil \quad xs & \mapsto & \mathsf{var}\ (\mathsf{inv}\ (i, xs)) \\
\lceil 1 \rceil \quad * & \mapsto & 1 \\
\lceil \Pi\, T^+ \rceil \quad f & \mapsto & \Pi \lambda s.\ \lceil T^+\ s \rceil\ (f\ s) \\
\lceil \Sigma\, T^+ \rceil \quad (s, xs) & \mapsto & \mathsf{delete}\, s\ (\lceil T^+\ s \rceil\ xs) \\
\lceil \mathsf{delete}\, s\, T^+ \rceil \quad (s', xs) & \mapsto & \mathsf{insert}\ (s = s')\ \lambda -.\ \lceil T^+ \rceil\ xs
\end{array}
$$

Fig. 5: Reornament

- If $n = \mathsf{suc}\, n'$, we are in the second branch of the Σ code, and the ornament of $\mathsf{suc}\, n'$ is a necessarily non-empty list. Again, the corresponding reornament deletes the choice of constructor, by deducing from the index that it must be 'cons. Since the list ornament extends natural numbers with an argument of type A (through the insert code), we must preserve this information in the reornament (through a Σ code). Finally, we index the recursive argument of the reornamented datatype by n':

$$\lceil \mathsf{List\text{-}Orn}\, A\, * \rceil\ (\text{'}\mathsf{suc}, n') \rightsquigarrow \mathsf{delete}\, \text{'}\mathsf{suc}\ (\Sigma \lambda a.\, \mathsf{var}\ (\mathsf{inv}\ (*, n')))$$

Altogether, we have ornamented lists by their length: when the index is 0, the ornamented list is empty; when the index is $\mathsf{suc}\, n'$, the ornamented list is non-empty and takes an argument indexed by n'. We have effectively described the datatype of vectors.

△

4.29 Definition (Reornament[37])**.** A reornament (Fig. 5) is thus defined over an ornament code $O : \mathsf{Orn}\, D\, u$ (for some description $D : \mathsf{IDesc}\, I$) and an index belonging to the base datatype $xs : [\![D]\!]\ (\mu\, D)$. On the 1 and Π codes, the reornament simply mirrors the underlying ornament, while peeling off the index: the *structure* of the three datatypes is identical. On a var code, the reornament also duplicates the underlying structure by pairing the index of the ornament (provided by the index i) with the recursive argument of the base datatype (provided by the argument xs). On an insert code, the reornament preserves the extra-information introduced by the ornament since it is absent from the index. However, on a Σ code, the ornament is merely duplicating information already provided by xs: in this case, we delete the argument, filling in the gap with the data provided by the index xs. On a delete code, we make sure – through an equality constraint – that the index xs is in sync with the data deleted by the ornament.

This definition over codes then lifts pointwise to ornaments:

DEFINITION

$$
\begin{array}{lll}
\lceil (o : \mathsf{orn}\, D\, u) \rceil & : & \mathsf{orn}\, [\![o]\!]_{\mathsf{orn}}\ (\pi_0 : (i : I) \times \mu\, D\ (u\, i) \to I) \\
\lceil o \rceil & \mapsto & \lambda (i, \mathsf{in}\, xs).\ \lceil o\, i \rceil\ xs
\end{array}
$$

▽

[37] MODEL: `Orn.Reornament`

4.30 Example (Reornament: vectors[38])**.** Applied to the ornament List-Orn (Example 4.7), this construction gives the fully Brady-optimised – detagged and forced – version of vectors (Example 3.8). That is, we determine which constructor of Vec is available by pattern-matching on the index. This is unlike the naive reornament (Example 4.26), which relies on constraints to enforce the indexing discipline.

△

4.31 Example (Reornament: indexed option type[39])**.** Under this definition, the reornament of Maybe-Orn (Example 4.5) describes the datatype

$$\begin{array}{l}\textbf{data}\ \mathsf{IMaybe}\,[A:\textsc{Set}](b:\mathsf{Bool}):\textsc{Set}\ \textbf{where}\\ \quad \mathsf{IMaybe}\,A\,\mathsf{true} \ni \mathsf{just}\,(a:A)\\ \quad \mathsf{IMaybe}\,A\,\mathsf{false} \ni \mathsf{nothing}\end{array}$$

where, similarly, constraints are off-loaded by pattern-matching on the indices (Example 3.8). Again, this must be compared with the definition obtained through the naive construction (Example 4.27), where we relied on constraints.

△

Note that our ability to *compute* over indices is crucial for this construction to work. Also, the datatypes we obtain are isomorphic to the datatypes one would have obtained by the algebraic ornament of the ornamental algebra, *i.e.*:

META-THEOREM

For all ornament $o : \mathsf{orn}\,D\,u$, we have

$$[\![\lceil o \rceil]\!]_{\mathsf{orn}} \quad \cong \quad [\![[\![o]\!]_{\mathsf{orn}}{}^{\mathsf{forgetAlg}\,o}]\!]_{\mathsf{orn}}$$

4.32 Remark (Computational interpretation)*.* Consequently, the coherence property of algebraic ornaments (Equation 4.20) is still valid. Constructively, this isomorphism gives the coherentOrn theorem[40] in one direction and the make function[41] in the other.

◊

4.33 Remark (Iterating reornamentation[42])*.* Every ornament induces a reornament. A reornament is itself an ornament: it therefore induces yet another reornament. We are naturally led to wonder if this process ever stops, and if so when. For example, the ornament of natural numbers into lists reornaments to vectors. Reornamenting vectors, we obtain an inductive predicate representing the length function $\mathsf{Length} : \mathsf{Nat} \to \mathsf{List}\,A \to \textsc{Set}$. Reornamenting Length leads to an object with no computational content: all its information has been erased and is provided by the indices.

The same pattern arises in general: every chain of reornaments is bound to end with a computationally trivial object. We deduce this from our massaged definition of reornaments (Definition 4.29). To illustrate our reasoning, we simultaneously iterate the reornamentation of the following (artificial) ornament

[38] MODEL: `Orn.Reornament.Examples.List`
[39] MODEL: `Orn.Reornament.Examples.Maybe`
[40] MODEL: `Orn.Reornament.Coherence`
[41] MODEL: `Orn.Reornament.Make`
[42] MODEL: `Orn.Reornament.Examples.Iterative`

indexed by $n : \mathsf{Nat}$

$$\begin{aligned} \mathsf{o} &: \mathsf{orn}\, D\, (\lambda n.\, *) \\ \mathsf{o} &\mapsto \lambda n.\, \Sigma\, \lambda a.\, \mathsf{insert}\, C\, \lambda c.\, \Pi\, \lambda b.\, \mathsf{delete}\, \mathsf{true}\, (\mathsf{var}\, (\mathsf{suc}\, n)) \end{aligned}$$

which ornaments the description

$$\begin{aligned} \mathsf{D} &: \mathsf{func}\, \mathbb{1} \\ \mathsf{D} &\mapsto \lambda *.\, \Sigma\, A\, \lambda a.\, \Pi\, B\, \lambda b.\, \Sigma\, \mathsf{Bool}\, \lambda x.\, \mathsf{var}\, * \end{aligned}$$

where A, B, and C are sets.

We proceed by case analysis on the ornament. On a 1, Π, and var code, the reornamentation proceeds purely structurally, merely duplicating the ornament's code and introducing no information (Definition 4.29, first 3 cases). The reornamentation *deletes* Σ codes, using the indexing information (Definition 4.29, fourth case). On a delete code, the reornament inserts an equality constraint (Definition 4.29, sixth case), which contains no information per se: it is only enforcing the indexing discipline. Only on an insert code does the reornament introduce new information through a Σ code (Definition 4.29, fifth case). On our example, the first reornament is defined by

$$\begin{aligned} o^{+} &: \mathsf{orn}\, (\llbracket \mathsf{o} \rrbracket_{\mathsf{orn}} : \mathsf{func}\, \mathsf{Nat})\, \pi_0 \\ o^{+} &\mapsto \lceil o \rceil \end{aligned}$$

and unfolds to

$$o^{+} \rightsquigarrow \lambda (n, \mathsf{in}\, (a, f)).\, \mathsf{delete}\, a\, (\Sigma\, C\, \lambda c.\, \Pi\, \lambda b.\, \mathsf{insert}\, (\mathsf{true} = \pi_0\, (f\, b))\, \lambda -.\, \mathsf{var}\, (\mathsf{inv}\, (\mathsf{suc}\, n, \pi_1\, (f\, b))))$$

In the subsequent iteration, these Σ codes in the reornament are in turn deleted by the re-reornament. On our example, the second reornamentation is defined by

$$\begin{aligned} o^{++} &: \mathsf{orn}\, (\llbracket o^{+} \rrbracket_{\mathsf{orn}} : \mathsf{Nat} \times \mu\, D\, *)\, \pi_0 \\ o^{++} &\mapsto \lceil o^{+} \rceil \end{aligned}$$

and unfolds to

$$\begin{aligned} o^{++} \rightsquigarrow \lambda ((n, \mathsf{in}\, (a, f)), \mathsf{in}\, (a', (c, f^{+}))).\, \mathsf{insert}\, (a' = a)\, \lambda -.\, \mathsf{delete}\, c\, (\Pi\, \lambda b.\, \Sigma\, \lambda q.\, \\ \mathsf{var}\, (\mathsf{inv}\, ((\mathsf{suc}\, n, \pi_1\, (f\, b)), f^{+}\, b)) \end{aligned}$$

where the Σ code duplicates the (computationally trivial) equation on x ($\mathsf{true} = x$) that was inserted in the previous step.

In the third iteration, there is nothing left in the code but equations and structural scaffoldings (in the form of var, 1, and Π codes): the resulting datatype is computationally trivial and is entirely determined by its indices. On our example, the third reornamentation is defined by

$$\begin{aligned} o^{+++} &: \mathsf{orn}\, (\llbracket o^{++} \rrbracket_{\mathsf{orn}} : \mathsf{func}\, (n : \mathsf{Nat}) \times \mu\, D\, * \times \mu\, \llbracket o \rrbracket_{\mathsf{orn}}\, n)\, \pi_0 \\ o^{+++} &\mapsto \lceil o^{++} \rceil \end{aligned}$$

and unfolds to

$$\begin{aligned} o^{+++} \rightsquigarrow \lambda (((n, \mathsf{in}\, (a, f)), \mathsf{in}\, (a, (c, f^{+}))), \mathsf{in}\, (c', f^{++})).\, \\ \Sigma\, \lambda q_1.\, \mathsf{insert}\, (c' = c)\, \lambda q_2.\, \Pi\, \lambda b.\, \mathsf{delete}\, \mathsf{refl}\, (\mathsf{var}\, (\mathsf{inv}\, (((\mathsf{suc}\, n, \pi_1\, (f\, b)), f^{+}\, b), \pi_1\, (f^{++}\, b)))) \end{aligned}$$

where the Σ code duplicates the (computationally trivial) equation on a ($a' = a$) that was inserted in the previous step.

Formally, we have that any two inhabitants of a triple reornament are provably equal:

META-THEOREM

Let $o : \mathsf{orn}\, D\, u$ be an ornament.
Let
1. $i : I$
2. $ds : \mu\, D\, (u\, i)$
3. $os : \mu\, [\![o]\!]_{\mathsf{orn}}\, i$
4. $os^+ : \mu\, [\![\lceil o \rceil]\!]_{\mathsf{orn}}\, (i, ds)$

be some indices.
For any pair
1. $xs : \mu\, [\![\lceil\lceil\lceil o \rceil\rceil\rceil]\!]_{\mathsf{orn}}\, (((i, ds), os), os^+)$
2. $ys : \mu\, [\![\lceil\lceil\lceil o \rceil\rceil\rceil]\!]_{\mathsf{orn}}\, (((i, ds), os), os^+)$

We have:

$$xs \quad \cong \quad ys$$

◊

In this section, we have adapted ornaments to our universe of datatypes. In doing so, we have introduced deletion ornaments, which rely on indexing to remove duplicated information from the datatypes. We shall see in Section 6 how this more careful definition can be turned to our advantage when we transport functions across ornaments.

5 A Universe of Function Types and their Ornaments

Ornaments provide us with a calculus of data-structures: from a given inductive family, we can ornament it to as many similarly-structured datatypes. The universe of ornaments is essentially an intensional characterisation of such structure-preserving transformation of datatype. Functional ornaments *build upon* ornaments (but not exclusively, as discussed in Remark 5.2), relying on them to capture the structural ties between the types of two functions, a base function and its lifting.

In that sense, the functional ornaments presented in this article are a generalisation of ornaments to function types. To describe them, we first need to be able to, intensionally and in type theory, manipulate function types. We thus define a universe of function types (Section 5.1). With it, we will be able to write generic programs over the class of functions captured by this universe. We define a functional ornament as a decoration over this universe (Section 5.2). The liftings implementing the functional ornament are related to the base function by a coherence property. To minimise the theorem-proving burden induced by coherence proofs, we expand our system with patches (Section 5.3): a patch is the type of the functions that satisfy the coherence property *by construction*. Finally, we show how we can project the lifting and its coherence certificate out of a patch (Section 5.4).

5.1 A universe of function types

5.1 Definition (Universe of types[43])**.** For clarity of exposition, we restrict our language of types to the bare minimum: a type can either be an exponential whose domain is an inductive type, or a product whose

[43] MODEL: `FunOrn.Functions`

first component is an inductive type, or the unit type – used as a termination marker:

DEFINITION

$$\begin{array}{l}\textbf{data}\ \mathsf{Type} : \mathrm{SET}_1\ \textbf{where}\\ \quad \mathsf{Type} \ni \mu\{(D : \mathsf{func}\, K) \cdot (k : K)\} \rightarrow (T : \mathsf{Type})\\ \qquad\quad \mid \mu\{(D : \mathsf{func}\, K) \cdot (k : K)\} \times (T : \mathsf{Type})\\ \qquad\quad \mid 1\end{array}$$

This universe codes the function space from some (maybe none) inductive types to some (maybe none) inductive types. Concretely, the codes are interpreted as follows:

DEFINITION

$$\begin{array}{lcl}[\![(T : \mathsf{Type})]\!]_{\mathsf{Type}} & : & \mathrm{SET}\\ [\![\mu\{D \cdot k\} \rightarrow T]\!]_{\mathsf{Type}} & \mapsto & \mu\, D\, k \rightarrow [\![T]\!]_{\mathsf{Type}}\\ [\![\mu\{D \cdot k\} \times T]\!]_{\mathsf{Type}} & \mapsto & \mu\, D\, k \times [\![T]\!]_{\mathsf{Type}}\\ [\![1]\!]_{\mathsf{Type}} & \mapsto & \mathbb{1}\end{array}$$

▽

5.2 Remark (Extensions). The constructions we develop next could be adapted to a more powerful universe – such as one supporting higher-order functions, non-inductive parameters, or including dependent quantifiers. However, this would needlessly complicate our exposition.

For instance, the treatment of non-inductive parameters would lead to further, but orthogonal, extensions of the functional ornaments; namely, inserting or deleting these quantifiers during functional ornamentation. To support higher-order functions, we would have to distinguish the variance of ornamentations, a technicality that we can simply overlook in a first-order system. ◊

5.3 Example (Coding $- < -$[44])**.** Written in the universe of function types, the type of the comparison function is

$$\begin{array}{l}\mathsf{type}< \ :\ \mathsf{Type}\\ \mathsf{type}< \mapsto \mu\{\mathsf{Nat\text{-}func} \cdot *\} \rightarrow \mu\{\mathsf{Nat\text{-}func} \cdot *\} \rightarrow \mu\{\mathsf{Bool\text{-}func} \cdot *\} \times 1\end{array}$$

The implementation of $- < -$ is essentially the same as earlier, excepted that it ought to return a pair of a Boolean and the inhabitant of the unit type. To reduce the syntactic noise introduced by this trivial multiplication by the unit, we assume that the type isomorphisms $A \times \mathbb{1} \cong \mathbb{1} \times A \cong A$ are definitionally true: we spare ourselves from writing pairs with unit, and projections out of such pairs. Again, we refer our reader to the companion Agda code for the non-simplified terms.

[44] MODEL: `FunOrn.Functions.Examples.Le`

To be explicit about the recursion pattern of this function, we make use of Epigram's *by* ($\Leftarrow$) gadget:

$$\begin{array}{lll} - & < - & : [\![\mathsf{type}{<}]\!]_{\mathsf{Type}} \\ m & < n & \Leftarrow \mathsf{Nat\text{-}elim}\, n \\ \quad m & < 0 & \mapsto \mathsf{false} \\ \quad m & < (\mathsf{suc}\, n) & \Leftarrow \mathsf{Nat\text{-}case}\, m \\ \quad\; 0 & < (\mathsf{suc}\, n) & \mapsto \mathsf{true} \\ \quad (\mathsf{suc}\, m) & < (\mathsf{suc}\, n) & \mapsto m < n \end{array}$$

That is, we first do induction on n and then, in the successor case, we proceed by case analysis over m.

△

5.4 Example (Coding $-+-$[45])**.** In the universe of function types, the type of addition is given by

$$\begin{array}{l} \mathsf{type}{+} \;:\; \mathsf{Type} \\ \mathsf{type}{+} \mapsto \mu\{\mathsf{Nat\text{-}func} \cdot * \} \rightarrow \mu\{\mathsf{Nat\text{-}func} \cdot * \} \rightarrow \mu\{\mathsf{Nat\text{-}func} \cdot * \} \times 1 \end{array}$$

Again, up to a trivial multiplication by $\mathbb{1}$, the implementation of $-+-$ is left unchanged:

$$\begin{array}{lll} - & + - & : [\![\mathsf{type}{+}]\!]_{\mathsf{Type}} \\ m & + n & \Leftarrow \mathsf{Nat\text{-}elim}\, m \\ \quad 0 & + n & \mapsto n \\ \quad (\mathsf{suc}\, m) & + n & \mapsto \mathsf{suc}\, (m + n) \end{array}$$

That is, it is defined by induction over m.

△

5.2 Functional ornament

It is now straightforward to define functional ornaments: we traverse the function type and ornament the inductive types as we go. Note that it is always possible to leave an object non-ornamented: we ornament by the identity (Example 4.9), which simply copies the original description.

5.5 Definition (Universe of functional ornaments [46])**.** Following this intuition, we define functional ornaments by the following grammar:

DEFINITION

$$\begin{array}{l} \textbf{data}\ \mathsf{FunOrn}\, (T : \mathsf{Type}) : \mathrm{SET}_1\ \textbf{where} \\ \quad \mathsf{FunOrn}\, (\mu\{D \cdot k\} \rightarrow T) \ni \forall u : I \rightarrow K.\, \mu^+\{(o : \mathsf{orn}\, D\, u) \cdot (i : u^{-1}\, k)\} \rightarrow (T^+ : \mathsf{FunOrn}\, T) \\ \quad \mathsf{FunOrn}\, (\mu\{D \cdot k\} \times T) \ni \forall u : I \rightarrow K.\, \mu^+\{(o : \mathsf{orn}\, D\, u) \cdot (i : u^{-1}\, k)\} \times (T^+ : \mathsf{FunOrn}\, T) \\ \quad \mathsf{FunOrn}\quad 1 \quad \ni 1 \end{array}$$

▽

[45] MODEL: `FunOrn.Functions.Examples.Plus`
[46] MODEL: `FunOrn.FunOrnament`

5.6 Definition (Lifting type [46])**.** We get the type of the liftings by interpreting the ornaments as we traverse the functional ornament:

DEFINITION

$$\begin{array}{lll}
[\![(T^+ : \mathsf{FunOrn}\ T)]\!]_{\mathsf{FunOrn}} & : & \mathrm{SET} \\
[\![\mu^+\{o \cdot \mathsf{inv}\, i\} \rightarrowtail T^+]\!]_{\mathsf{FunOrn}} & \mapsto & \mu\, [\![o]\!]_{\mathsf{orn}}\, i \to [\![T^+]\!]_{\mathsf{FunOrn}} \\
[\![\mu^+\{o \cdot \mathsf{inv}\, i\} \times T^+]\!]_{\mathsf{FunOrn}} & \mapsto & \mu\, [\![o]\!]_{\mathsf{orn}}\, i \times [\![T^+]\!]_{\mathsf{FunOrn}} \\
[\![1]\!]_{\mathsf{FunOrn}} & \mapsto & \mathbb{1}
\end{array}$$

▽

We want the ornamented function to be *coherent* with respect to the base function we started from: for a function $f : \mu\, D\, k \to \mu\, E\, l$, the ornamented function $f^+ : \mu\, [\![o_D]\!]_{\mathsf{orn}}\, i \to \mu\, [\![o_E]\!]_{\mathsf{orn}}\, j$ is said to be coherent with f if the following diagram commutes

$$\begin{array}{ccc}
\mu\, [\![o_D]\!]_{\mathsf{orn}}\, i & \xrightarrow{\quad f^+ \quad} & \mu\, [\![o_E]\!]_{\mathsf{orn}}\, j \\
\Big\downarrow{\scriptstyle \mathsf{forget}\, o_D} & & \Big\downarrow{\scriptstyle \mathsf{forget}\, o_E} \\
\mu\, D\, (u\, i) & \xrightarrow[\quad f \quad]{} & \mu\, E\, (v\, j)
\end{array}$$

or, equivalently in type theory:

$$(x^+ : \mu\, [\![o_D]\!]_{\mathsf{orn}}\, i) \to f\, (\mathsf{forget}\, o_D\, x^+) = \mathsf{forget}\, o_E\, (f^+\, x^+)$$

This captures our intuition that the lifted function f^+ behaves like the base function f, only that it also carries the extra-information introduced by the ornament o_D over to the ornament o_E. Coherence states that this extra-step does not interfere with its core operational behaviour, which is specified by f.

5.7 Definition (Coherence[47])**.** This definition of coherence generalises to any arity. We define it by induction over the code of functional ornaments:

DEFINITION

$$\begin{array}{l}
\mathsf{Coherence}\, (T^+ : \mathsf{FunOrn}\, T)\, (f : [\![T]\!]_{\mathsf{Type}})\, (f^+ : [\![T^+]\!]_{\mathsf{FunOrn}}) : \mathrm{SET} \\
\mathsf{Coherence}\, (\mu^+\{o \cdot \mathsf{inv}\, i\} \rightarrowtail T^+)\quad f \quad f^+ \quad \mapsto \\
\qquad\qquad (x^+ : \mu\, [\![o]\!]_{\mathsf{orn}}\, i) \to \mathsf{Coherence}\, T^+\, (f\, (\mathsf{forget}\, o\, x^+))\, (f^+ x^+) \\
\mathsf{Coherence}\, (\mu^+\{o \cdot \mathsf{inv}\, i\} \times T^+)\, (x, xs)\, (x^+, xs^+) \mapsto \\
\qquad\qquad \mathsf{forget}\, o\, x^+ = x \times \mathsf{Coherence}\, T^+\, xs\, xs^+ \\
\mathsf{Coherence}\quad 1 \quad * \quad * \quad \mapsto \mathbb{1}
\end{array}$$

▽

[47] MODEL: `FunOrn.FunOrnament`

5.8 Example (Ornamenting type$<$ to describe lookup[48])**.** In Section 2, we have identified the ornaments taking the type of $-<-$ to the type of lookup. We ornament Nat to List A (Example 4.7), and Bool to Maybe A (Example 4.5). From there, the functional ornament describing the type of the lookup function is as follows:

$$\begin{aligned}&\mathsf{typeLookup} \;:\; \mathsf{FunOrn}\,\mathsf{type}<\\&\mathsf{typeLookup} \mapsto \mu^{+}\{\mathsf{idO}\,\mathsf{Nat\text{-}func}\cdot \mathsf{inv}*\}\rightarrow\mu^{+}\{\mathsf{List\text{-}Orn}\,A\cdot \mathsf{inv}*\}\rightarrow\mu^{+}\{\mathsf{Maybe\text{-}Orn}\,A\cdot \mathsf{inv}*\}\times 1\end{aligned}$$

We leave it to the reader to verify that $[\![\mathsf{typeLookup}]\!]_{\mathsf{FunOrn}}$ unfolds to the type of the lookup function, up to a multiplication by $\mathbb{1}$. Also, unfolding the coherence condition gives the desired property:

$$\begin{aligned}&\mathsf{Coherence}\,\mathsf{typeLookup}\,(-<-) \rightsquigarrow \lambda f^{+} : [\![\mathsf{typeLookup}]\!]_{\mathsf{FunOrn}}\cdot\\&\quad (n:\mathsf{Nat})(xs:\mathsf{List}\,A)\rightarrow \mathsf{isJust}\,(f^{+}\,n\,xs) = n<\mathsf{length}\,xs\end{aligned}$$

△

5.9 Remark. This equation is not *specifying* the lookup function: it is only establishing a computational relation between $-<-$ and a candidate lifting f^{+}, for which lookup is a valid choice. However, one could be interested in other functions satisfying this coherence property and they would be handled by our system just as well: the notion of functional ornament (Definition 5.5), and its coherence property (Definition 5.7) still apply. For example, assuming that A is a monoid, a function that sums the elements of List A from 0 to the index n, or returns nothing if the index is out of bound is coherent with $-<-$.

◊

5.10 Example (Ornamenting type$+$ to describe $-{+\!\!+}-$[49])**.** The functional ornament of type$+$ relies solely on the ornamentation of Nat into List A:

$$\begin{aligned}&\mathsf{type}{+\!\!+} \;:\; \mathsf{FunOrn}\,\mathsf{type}+\\&\mathsf{type}{+\!\!+} \mapsto \mu^{+}\{\mathsf{List\text{-}Orn}\,A\cdot \mathsf{inv}*\}\rightarrow\mu^{+}\{\mathsf{List\text{-}Orn}\,A\cdot \mathsf{inv}*\}\rightarrow\mu^{+}\{\mathsf{List\text{-}Orn}\,A\cdot \mathsf{inv}*\}\times 1\end{aligned}$$

Again, we check that unfolding $[\![\mathsf{type}{+\!\!+}]\!]_{\mathsf{FunOrn}}$ gives the type of $-{+\!\!+}-$ while the coherence condition $\mathsf{Coherence}\,\mathsf{type}{+\!\!+}\,(-+-)$ correctly captures our requirement that appending lists preserves their lengths. As before, the list append function is not the only valid lifting: one could for example consider a function that reverses the first list and appends it to the second one.

△

5.3 Patches

The coherence of the lifting $f^{+} : [\![T^{+}]\!]_{\mathsf{FunOrn}}$ of a base function $f : [\![T]\!]_{\mathsf{Type}}$ is therefore captured by the coherence predicate $\mathsf{Coherence}\,T^{+}\,f$. To implement a lifting that is coherent, we might ask the user to first implement the lifting f^{+} and then prove its coherence. However, we find this process unsatisfactory: we fail to harness the power of dependent types when implementing f^{+}, this weakness being then paid off by tedious proof obligations. To overcome this limitation, we define the notion of Patch as the type of *all* the functions that are coherent by construction.

[48] MODEL: `FunOrn.FunOrnament.Examples.Lookup`
[49] MODEL: `FunOrn.FunOrnament.Examples.Append`

5.11 Remark. We are looking for an isomorphism here: we will define patches in such a way that they are in bijection with the liftings satisfying a coherence property. Put otherwise, we want that:

$$\mathsf{Patch}\,T\,T^{+}\,f \quad \cong \quad (f^{+} : [\![T^{+}]\!]_{\mathsf{FunOrn}}) \times \mathsf{Coherence}\,T^{+}\,f\,f^{+}$$

◊

In this article, we constructively exploit this bijection in the left-to-right direction: having implemented a patch f^{++} of type $\mathsf{Patch}\,T\;T^{+}\,f$, we show how we can "apply" it, and extract a lifting together with its coherence proof.

5.12 Example (Patching $-<-$)**.** Before giving the general construction of a Patch, let us first work through our running example. After having functionally ornamented $-<-$ with typeLookup, the lifting function f^{+} and coherence property can be represented by the following commuting diagram:

$$\begin{array}{ccccc}
\mathsf{Nat} & \times & \mathsf{List}\,A & \xrightarrow{\;f^{+}\;} & \mathsf{Maybe}\,A \\
{\scriptstyle \mathsf{id}}\,\| & & {\scriptstyle \mathsf{length}}\downarrow & & \downarrow{\scriptstyle \mathsf{isJust}} \\
\mathsf{Nat} & \times & \mathsf{Nat} & \xrightarrow[-<-]{} & \mathsf{Bool}
\end{array} \tag{5.13}$$

In type theory, this is written as

$$\begin{array}{l}
(f^{+} : \mathsf{Nat} \times \mathsf{List}\,A \to \mathsf{Maybe}\,A) \times \\
\quad (m : \mathsf{Nat})(as : \mathsf{List}\,A) \to \mathsf{isJust}\,(f^{+}\,m\,as) = m < \mathsf{length}\,as
\end{array}$$

Applying intensional choice, this is isomorphic to

$$\begin{array}{l}
\cong (m : \mathsf{Nat}) \times (n : \mathsf{Nat}) \times (as : \mathsf{List}\,A) \times \mathsf{length}\,as = n \to \\
\quad (ma : \mathsf{Maybe}\,A) \times \mathsf{isJust}\,ma = m < n
\end{array}$$

Now, by the characterisation of reornaments (Equation 4.20), we have that:

$$\begin{array}{rcll}
(as : \mathsf{List}\,A) \times \mathsf{length}\,as = n & \cong & \mathsf{Vec}\,A\,n & \text{and} \\
(ma : \mathsf{Maybe}\,A) \times \mathsf{isJust}\,ma = b & \cong & \mathsf{IMaybe}\,A\,b &
\end{array}$$

Applying these isomorphisms, we obtain the following type, which we call the Patch of the functional ornament typeLookup:

$$\cong (m : \mathsf{Nat}) \to (n : \mathsf{Nat}) \times (vs : \mathsf{Vec}\,A\,n) \to \mathsf{IMaybe}\,A\,(m < n)$$

This last type is thus isomorphic to the pair of a lifting and its coherence proof.

△

Intuitively, the Patch construction amounts to turning the vertical arrows of the commuting diagram (5.13) into the equivalent reornaments. In type-theoretic terms, it turns the pairs of datatypes and their algebraically defined constraints into the equivalent reornaments. The coherence property of reornaments (Equation 4.20) tells us that projecting the ornamented function down to its non-ornamented components gives back the base function. By turning the projection functions into inductive datatypes, we enforce

the coherence property directly by the index: we introduce a fresh index for the arguments (in Example 5.12, introducing m and n) and index the return types by the result of the non-ornamented function (in Example 5.12, indexing $\mathsf{IMaybe}\,A$ by the result $m<n$).

5.14 Remark (Terminology). The name of "patch" comes from the idea that the Patch type $\mathsf{Patch}\,T\,T^+\,f$ captures all those (coherent) functions that *extends* the base function f. In Section 5.4, we shall see how such a patch can be *applied* (Definition 5.22), *i.e.* we describe how a base function is patched by an inhabitant of a Patch type to build its coherent lifting.

◊

5.15 Definition (Patch type[50])**.** We define the Patch type generically by induction over the functional ornament. Upon an argument (*i.e.* a code $\mu^+\{o\cdot\mathsf{inv}\,i\}\!\rightarrow$), we introduce a fresh index and the reornament of o. Upon a result (*i.e.* a code $\mu^+\{o\cdot\mathsf{inv}\,i\}\!\times$), we ask for a reornament of o indexed by the result of the base function.

DEFINITION

$$\begin{array}{l}
\mathsf{Patch}\quad(T:\mathsf{Type})\quad(T^+:\mathsf{FunOrn}\,T)\quad(f:[\![T]\!]_{\mathsf{Type}})\ :\ \mathsf{SET}\\
\mathsf{Patch}\,(\mu\{D\cdot u\,i\}\!\rightarrow T)\,(\mu^+\{o\cdot\mathsf{inv}\,i\}\!\rightarrow T^+)\quad f\quad\mapsto\\
\qquad(x:\mu\,D\,(u\,i))\rightarrow\mu\,[\![\lceil o\rceil]\!]_{\mathsf{orn}}\,(i,x)\rightarrow\mathsf{Patch}\,T\,T^+\,(f\,x)\\
\mathsf{Patch}\,(\mu\{D\cdot u\,i\}\!\times T)\,(\mu^+\{o\cdot\mathsf{inv}\,i\}\!\times T^+)\quad(x,xs)\quad\mapsto\\
\qquad\mu\,[\![\lceil o\rceil]\!]_{\mathsf{orn}}\,(i,x)\times\mathsf{Patch}\,T\,T^+\,xs\\
\mathsf{Patch}\quad 1\quad 1\quad *\quad\mapsto\mathbb{1}
\end{array}$$

▽

5.16 Example (Patch of typeLookup[51])**.** The type of the coherent liftings of $-<-$ by typeLookup, as defined by the Patch of $-<-$ by typeLookup, unfolds to

$$(m:\mathsf{Nat})(m^+:\mu\,[\![\lceil\mathsf{idO\,Nat\text{-}func}\rceil]\!]_{\mathsf{orn}}\,m)\rightarrow(n:\mathsf{Nat})(vs:\mu\,[\![\lceil\mathsf{List}\,A\rceil]\!]_{\mathsf{orn}}\,n)\rightarrow\mu\,[\![\lceil\mathsf{Maybe}\,A\rceil]\!]_{\mathsf{orn}}\,(m<n)\times\mathbb{1}$$

△

5.17 Remark. $\mu\,[\![\lceil\mathsf{idO\,Nat\text{-}func}\rceil]\!]_{\mathsf{orn}}\,m$ is isomorphic to $\mathbb{1}$: all the content of the datatype has been forced – the recursive structure of the datatype is entirely determined by its index – and detagged – the choice of constructors is entirely determined by its index, leaving no actual data in it. Being computationally uninteresting, we ignore this argument. On the other hand, $\lceil\mathsf{List}\,A\rceil$ and $\lceil\mathsf{Maybe}\,A\rceil$ are, respectively, the previously introduced vectors (Example 4.30) and indexed option (Example 4.31) types.

◊

5.18 Example (Patch of type+[52])**.** Similarly, the Patch of $-+-$ by type+ unfolds to the type of the vector append function

$$(m:\mathsf{Nat})(xs:\mu\,[\![\lceil\mathsf{List}\,A\rceil]\!]_{\mathsf{orn}}\,m)\rightarrow(n:\mathsf{Nat})(ys:\mu\,[\![\lceil\mathsf{List}\,A\rceil]\!]_{\mathsf{orn}}\,n)\rightarrow\mu\,[\![\lceil\mathsf{List}\,A\rceil]\!]_{\mathsf{orn}}\,(m+n)\times\mathbb{1}$$

[50] MODEL: `FunOrn.Patch`
[51] MODEL: `FunOrn.Patch.Examples.Lookup`
[52] MODEL: `FunOrn.Patch.Examples.Append`

where, again, the datatype $\mu\,[\![\lceil \mathsf{List}\,A\rceil]\!]_{\mathsf{orn}}$ corresponds exactly to vectors.

△

5.19 Theorem. *Following our Remark 5.11, we have that a* Patch *is isomorphic to the pair of a lifting and its coherence proof:*

META-THEOREM

For any function type $T : \mathsf{Type}$*, any functional ornament* $T^+ : \mathsf{FunOrn}\,T$ *of* T*, and any base function* $f : [\![T]\!]_{\mathsf{Type}}$*, we have:*

$$\mathsf{Patch}\,T\,T^+\,f \quad \cong \quad (f^+ : [\![T^+]\!]_{\mathsf{FunOrn}}) \times \mathsf{Coherence}\,T^+\,f\,f^+$$

That is, our definition of the Patch *type enforces that its inhabitants are exactly those liftings that are coherent by construction.*

Proof. For clarity, we shall only write the proof for arity one. The generalisation to multiple input and output arities is straightforward but laboriously verbose. So, from a base function $f : \mu\,D\,k \to \mu\,E\,l$, we start with its lifting and the associated coherence property:

$$\begin{array}{l}(f^+ : \mu\,[\![o_D]\!]_{\mathsf{orn}}\,i \to \mu\,[\![o_E]\!]_{\mathsf{orn}}\,j) \times \\ \quad (ds^+ : \mu\,[\![o_D]\!]_{\mathsf{orn}}\,i) \to \mathsf{forget}\,o_E\,(f^+\,d^+) = f(\mathsf{forget}\,o_D\,ds^+)\end{array}$$

Applying intensional choice, we obtain the following isomorphic type:

$$\cong (ds : \mu\,D\,(u\,i)) \times (ds^+ : \mu\,[\![o_D]\!]_{\mathsf{orn}}\,i) \times \mathsf{forget}\,o_D\,ds^+ = ds \to (es^+ : \mu\,[\![o_E]\!]_{\mathsf{orn}}\,j) \times \mathsf{forget}\,o_E\,es^+ = f\,ds$$

Then, we can simply use the characterisation of reornaments (Equation 4.20) to turn every pair $(x^+ : \mu\,[\![o_X]\!]_{\mathsf{orn}}\,i) \times t = \mathsf{forget}\,o_X\,x^+$ into the isomorphic inductive type $\mu\,[\![\lceil o_X\rceil]\!]_{\mathsf{orn}}\,t$:

$$\cong (ds : \mu\,D\,(u\,i)) \times \mu\,[\![\lceil o_D\rceil]\!]_{\mathsf{orn}}\,ds \to \mu\,[\![\lceil o_E\rceil]\!]_{\mathsf{orn}}\,(f\,ds)$$

which corresponds to the Patch type of this functional ornament.

□

5.20 Remark (Computational interpretation). Constructively, we translate the left-to-right direction of this isomorphism into the pair of a patch function (Definition 5.22, which extracts the lifting) and a coherence proof (Definition 5.23, which establishes that such a lifting is coherent).

◊

5.21 Remark (When to index?). While these precisely indexed functions relieve us from the burden of theorem proving, this approach is not always applicable. For instance, if we were to implement a length-preserving list reversal function, our patching machinery would ask us to implement vrev:

$$\begin{array}{lll}\mathsf{vrev}\,(xs : \mathsf{Vec}\,A\,n) & : & \mathsf{Vec}\,A\,n \\ \mathsf{vrev}\quad \mathsf{nil} & \mapsto & \mathsf{nil} \\ \mathsf{vrev}\;\;(\mathsf{cons}\,a\,vs) & \mapsto & \{(\mathsf{vrev}\,vs) +\!\!+ (\mathsf{cons}\,a\,\mathsf{nil}) : \mathsf{Vec}\,A\,(1+n)\}\end{array}$$

To complete this goal calls for some proving in order to match up the types: we must appeal to the equational theory of addition. Here, the term we put in the hole has type $\mathsf{Vec}\,A\,(n+1)$ while the expected type is $\mathsf{Vec}\,A\,(1+n)$. The commutativity of addition is beyond the grasp of most type checkers, which are often limited to deciding definitional identities.

In the case of vrev, unless the type checker works up to equational theories, as done in CoqMT (Strub, 2010), the programmer is certainly better off using our machinery to generate the coherence condition (Section 5.2), implement the lifting, and write its coherence proof manually. While the patching machinery would still work, implementing a function realising the Patch specification would require some (cumbersome) rewritings of the types, thus littering the program with proofs.

This example gives a hint as to what can be seen as a "good" coherence property: because we want the type checker to do all the proving, the equations we rely on at the type level have to be definitionally true, either because our logic decides a rich definitional equality, or because we rely on operations that satisfy these identities by definition.

◊

5.4 Patching and coherence

At this stage, we can implement the ilookup function exactly as we did in Section 2. From there, we now want to obtain the lookup function and its coherence certificate. More generally, having implemented a function satisfying the Patch type, we want to extract the lifting and its coherence proof. Perhaps not surprisingly, we obtain this construction by looking at the meta-theorem of the previous section (Theorem 5.19) through our constructive glasses: indeed, since the Patch type is isomorphic to the class of liftings satisfying the coherence property, we effectively get a function taking every Patch to a lifting (Definition 5.22) and its coherence proof (Definition 5.23). More precisely, we obtain the lifting by generalising the reornament-induced forget functions to functional ornaments while we obtain the coherence proof by generalising the reornament-induced coherentOrn theorem.

5.22 Definition (Patching[53])**.** We call *patching* the action of projecting the coherent lifting from a Patch function. Again, it is defined by induction over the functional ornament. When ornamented arguments are introduced (*i.e.* with the code $\mu^+\{o \cdot \mathsf{inv}\, i\}\!\rightarrow$), we simply patch the body of the function. This is possible because from $x^+ : \mu\, [\![o_D]\!]_{\mathsf{orn}}$, we can forget the ornament to compute $f\,(\mathsf{forget}\, o_D\, x^+)$, and we can also make the reornament to compute $f^{++}\,\_\,(\mathsf{make}\,\lceil o \rceil\, x^+)$. When an ornamented result is to be returned (*i.e.* with the code $\mu^+\{o \cdot \mathsf{inv}\, i\}\times$), we simply forget the reornamentation computed by the coherent lifting:

DEFINITION

$$
\begin{aligned}
&\mathsf{patch}\,(T^+ : \mathsf{FunOrn}\,T)\,(f : [\![T]\!]_{\mathsf{Type}})\,(f^{++} : \mathsf{Patch}\,T\,T^+\,f) : [\![T^+]\!]_{\mathsf{FunOrn}}\\
&\mathsf{patch}\,(\mu^+\{o \cdot \mathsf{inv}\, i\}\!\rightarrow T^+)\quad f\qquad f^{++}\qquad \mapsto\\
&\qquad \lambda x^+.\,\mathsf{patch}\,T^+\,(f\,(\mathsf{forget}\, o\, x^+))\\
&\qquad\qquad\qquad (f^{++}\,(\mathsf{forget}\, o\, x^+)\,(\mathsf{make}\,\lceil o \rceil\, x^+))\\
&\mathsf{patch}\,(\mu^+\{o \cdot \mathsf{inv}\, i\}\times T^+)\,(x, xs)\,(x^{++}, xs^{++}) \mapsto\\
&\qquad (\mathsf{forget}\,\lceil o \rceil\, x^{++}, \mathsf{patch}\,T^+\, xs\, xs^{++})\\
&\mathsf{patch}\qquad 1\qquad\qquad *\qquad\quad *\qquad \mapsto *
\end{aligned}
$$

▽

5.23 Definition (Coherence of a patch[54])**.** Extracting the coherence proof follows a similar pattern. We introduce arguments as we go, just as we did with patch. When we reach a result, we have to prove the

[53] MODEL: `FunOrn.Patch.Apply`
[54] MODEL: `FunOrn.Patch.Coherence`

coherence of the result returned by the patched function, which is a straightforward application of the coherentOrn theorem:

DEFINITION

$$\begin{array}{l}
\mathsf{coherence}\,(T^+ : \mathsf{FunOrn}\,T)\,(f : [\![T]\!]_{\mathsf{Type}})\,(f^{++} : \mathsf{Patch}\,T\,T^+\,f) : \mathsf{Coherence}\,T^+\,f\,(\mathsf{patch}\,T^+\,f\,f^{++}) \\
\mathsf{coherence}\,(\mu^+\{o \cdot \mathsf{inv}\,i\} \rightarrow T^+) \quad f \quad f^{++} \quad \mapsto \\
\qquad \lambda x^+ . \mathsf{coherence}\,T^+\,(f\,(\mathsf{forget}\,o\,x^+)) \\
\qquad\qquad (f^{++}\,(\mathsf{forget}\,o\,x^+)\,(\mathsf{make}\,\lceil o \rceil\,x^+)) \\
\mathsf{coherence}\,(\mu^+\{o \cdot \mathsf{inv}\,i\} \times T^+)\,(x, xs)\,(x^+, xs^{++}) \mapsto \\
\qquad (\mathsf{coherentOrn}\,x^+, \mathsf{coherence}\,T^+\,xs\,xs^{++}) \\
\mathsf{coherence} \quad 1 \quad * \quad * \quad \mapsto *
\end{array}$$

▽

5.24 Example (Obtaining lookup and its coherence, for free[55])**.** This last step is a mere application of the patch and coherence functions. Hence, we define lookup as follows:

$$\begin{array}{l}
\mathsf{lookup} \;:\; [\![\mathsf{typeLookup}]\!]_{\mathsf{FunOrn}} \\
\mathsf{lookup} \mapsto \mathsf{patch}\,\mathsf{typeLookup}\,(- < -)\,\mathsf{ilookup}
\end{array}$$

And we get its coherence proof, here spelled in full (up to a multiplication by $\mathbb{1}$):

$$\begin{array}{l}
\mathsf{cohLookup}\,(n : \mathsf{Nat})\,(xs : \mathsf{List}\,A) \;:\; \mathsf{isJust}\,(\mathsf{lookup}\,n\,xs) = n < \mathsf{length}\,xs \\
\mathsf{cohLookup} \quad n \quad xs \quad \mapsto \mathsf{coherence}\,\mathsf{typeLookup}\,(- < -)\,\mathsf{ilookup}\,n\,xs
\end{array}$$

△

5.25 Remark (Code readability). The lookup function thus defined is rather daunting, especially for a potential user of that piece of code. However, we must bear in mind that lookup is in fact entirely specified by ilookup: there is no point in inspecting the definition of lookup. In a programming environment, we could imagine some syntactic sugar akin to our notation for ornaments. For example, we would state that lookup is a functional ornamentation of $- < -$. We would be lead to – transparently – implement ilookup in lieu of lookup.

◊

5.26 Example (Obtaining $- +\!\!+ -$ and its coherence, for free[56])**.** Assuming that we have implemented the coherent lifting vappend, we obtain concatenation of lists and its coherence proof by simply running our generic machinery:

$$\begin{array}{l}
+\!\!+ \;:\; [\![\mathsf{type}+\!\!+]\!]_{\mathsf{FunOrn}} \\
+\!\!+ \mapsto \mathsf{patch}\,\mathsf{type}+\!\!+\,(- + -)\,\mathsf{vappend}
\end{array}$$

$$\begin{array}{l}
\mathsf{coh}+\!\!+\,(xs : \mathsf{List}\,A)\,(ys : \mathsf{List}\,A) \;:\; \mathsf{length}\,(xs +\!\!+ ys) = (\mathsf{length}\,xs) + (\mathsf{length}\,ys) \\
\mathsf{coh}+\!\!+ \quad xs \quad ys \quad \mapsto \mathsf{coherence}\,\mathsf{type}+\!\!+\,(- + -)\,\mathsf{vappend}\,xs\,ys
\end{array}$$

△

[55] MODEL: `FunOrn.Lift.Examples.Lookup`
[56] MODEL: `FunOrn.Lift.Examples.Append`

Looking back at the pedestrian construction of Section 2, we can measure the progress we have made. In the pedestrian approach, we had to (manually) index our datatypes (by defining Vec and IMaybe, and their projections back to, respectively, the list and option types), index the type signature of lookup (obtaining the type of ilookup), project lookup out of ilookup, and write its coherence proof (defining cohLookup from ilookup and the projection functions).

Functional ornaments let us focus on ornamenting the individual datatypes (Example 5.8). The rest is automatically generated for us: the coherence condition is computed by the generic Coherence type (Example 5.8), the indexed type of lookup is computed by the Patch type (Example 5.16), lookup is obtained by applying the patch (Example 5.24), and its coherence an instance of the generic coherence lemma (Example 5.24).

This is not just convenient automation: a functional ornament establishes a strong connection between two functions. By pinning down this connection in this universe, we turn this knowledge into an effective object that can be manipulated and reasoned about within type theory.

We make use of this concreteness when we construct the Patch induced by a functional ornament: this is again a construction that is generic now, while we had to tediously (and perhaps painfully) construct it in Section 2. Similarly, we get patching and extraction of the coherence proof for free now, while we had to manually fiddle with several projection and injection functions.

We presented the Patch type as the type of the liftings coherent by construction. As we have seen, its construction and further projection down to a lifting is entirely automated, hence effortless. This is a significant step forward: we could either implement lookup and then prove it coherent, or we could go through the trouble of manually defining carefully indexed types and write a function correct by construction. Manually crafting these finely indexed types and functions takes up time and adds complexity to a code base. By automating these constructions, using finely indexed types is now just as economic (time-wise and complexity-wise) as proving the coherence after the fact. From a programming perspective, the second approach is much more appealing. In a word, we have made an appealing technique extremely cheap!

5.27 Remark (No meta-theory). We must reiterate that none of the above constructions involve extending the type theory: building upon our universe of datatypes, ornaments and functional ornaments are internalised as a few generic programs and inductive types. For systems such as Agda, Coq, or an ML with GADTs, we would need to extend the language – and therefore the meta-theory – to be able to reify inductive definitions, and provide an ornament mechanism. The fact that our constructions – such as the patching operations (Definition 5.22 and Definition 5.23), and the liftings (introduced in the next section, Definition 6.5, and Definition 6.17) type check in our model suggests that adding these objects at the meta-level is consistent with a pre-existing meta-theory.

◊

5.28 Remark (Efficiency considerations). The patching framework relies crucially on the duality between a reornament and its ornament presentation subject to a proof. While patching (Definition 5.22), we cross this isomorphism in both directions. In effect, this involves a traversal of each of the input datatypes and a traversal of each of the output datatypes. However, operationally, these traversal are identities: the only purpose of these terms is at the logical level, for the type checker to fix the types.

For example, the lookup function amounts to the following term:

$$\begin{aligned}\mathsf{lookup} &\triangleq \mathsf{patch}\,\mathsf{typeLookup}\,(-<-)\,\mathsf{ilookup}\\ &\rightsquigarrow \lambda m : \mathsf{Nat}.\,\lambda xs : \mathsf{List}\,A.\,\pi_0(\mathsf{forgetIMaybe}\,(\mathsf{ilookup}\,m\,(\mathsf{makeVec}\,xs)))\end{aligned}$$

```
_      <  _    : [[type<]]Type
m      <  n    ⇐ Nat-elim n
  m    <  0    ↦ false
  m    < (suc n) ⇐ Nat-case m
    0  < (suc n) ↦ true
    (suc m) < (suc n) ↦ m < n
```

```
ilookup (m : Nat) (vs : Vec A n) : IMaybe A (m < n)
ilookup      m      vs      ⇐ Vector-elim vs
  ilookup    m      nil     ↦ nothing
  ilookup    m      (cons a vs) ⇐ Nat-case m
    ilookup  0      (cons a vs) ↦ just a
    ilookup (suc m) (cons a vs) ↦ ilookup m vs
```

Fig. 6: Implementations of $- < -$ and ilookup

In this definition, we rely on makeVec, which traverses the (input) list to return a vector indexed by the list's length. Operationally, this is an identity. We also rely on forgetIMaybe, which traverses the (output) IMaybe type to project it back to a non-indexed option type. Again, operationally, this is an identity.

In future work, we would like to transform our library of smart constructors into a proper domain-specific language (DSL). This way, implementing a coherent lifting would amount to working in a DSL for which an optimising compiler could compute away – by partial evaluation (Brady & Hammond, 2010) – the computationally irrelevant operations.

◊

6 Lazy Programmers, Smart Constructors

In our journey from $- < -$ to lookup, we had to implement the ilookup function. It is instructive to put $- < -$ and ilookup side-by-side (Fig. 6). First, both functions follow the same recursion pattern: induction over *n*/*vs* followed by case analysis over *m*. Second, the returned constructors are related through the Maybe ornament: knowing that we have returned true or false when implementing $- < -$, we can deduce which of just or nothing will be used in ilookup. Interestingly, the only unknown, hence the only necessary input from the user, is the *a* in the just case: this is precisely the information that has been introduced by the Maybe ornament.

In this section, we are going to leverage our knowledge of the definition of the base function – such as $- < -$ – to guide the implementation of the coherent lifting – such as ilookup: instead of re-implementing ilookup by duplicating most of the code of $- < -$, the user indicates *what to transport* and only provides the *strictly-necessary* inputs. We are primarily interested in transporting two forms of structure:

Recursion pattern: if the base function is a catamorphism $(\!|\alpha|\!)$ and the user provides us with a *coherent algebra* α^{++} of α, we construct the coherent lifting $(\!|\alpha^{++}|\!)$ of $(\!|\alpha|\!)$;

Returned constructor: if the base function returns a constructor C and the user provides us with a *coherent extension* of C, we construct the coherent lifting of C.

We shall formalise what we understand by being a *coherent algebra* and a *coherent extension* below. The key idea is to identify the strictly-necessary inputs from the user, helped in that by the ornaments. It is then straightforward to build the lifted objects, automatically and generically.

6.1 Transporting recursion patterns

When transporting a function, we are very unlikely to change the recursion pattern of the base function. Indeed, the very reason why we *can* do this transformation is that the lifting uses exactly the same underlying structure to compute its results. Hence, most of the time, we could just ask the computer to use the recursion pattern induced by the base function: the only task left to the user will be to give an algebra.

6.1 Example (Lifting a catamorphism)**.** To understand how we transport recursion patterns, let us look again at the coherence property of liftings, but this time specialising to a function that is a catamorphism:

$$\begin{array}{ccc} \mu\,[\![o_D]\!]_{\mathsf{orn}}\,i & \xrightarrow{(\!|\beta|\!)} & \mu\,[\![o_E]\!]_{\mathsf{orn}}\,i \\ \Big\downarrow{\scriptstyle \mathsf{forget}\,o_D} & & \Big\downarrow{\scriptstyle \mathsf{forget}\,o_E} \\ \mu\,D\,(u\,i) & \xrightarrow[(\!|\alpha|\!)]{} & \mu\,E\,(u\,i) \end{array}$$

By the fold-fusion theorem (Bird & de Moor, 1997), it is sufficient to work on the algebras, where we have the following diagram:

$$\begin{array}{ccccc} [\![o_D]\!]_{\mathsf{orn}}\,(\mu\,[\![o_E]\!]_{\mathsf{orn}})\,i & & \xrightarrow{\qquad\beta\qquad} & & \mu\,[\![o_E]\!]_{\mathsf{orn}}\,i \\ \Big\downarrow{\scriptstyle [\![[\![o_D]\!]_{\mathsf{orn}}]\!]^{\rightarrow}\,(\mathsf{forget}\,o_E)} & & & & \Big\downarrow{\scriptstyle \mathsf{forget}\,o_E} \\ [\![o_D]\!]_{\mathsf{orn}}\,(\mu\,E \circ u)\,i & \xrightarrow[\mathsf{forgetNT}\,o_D]{} & [\![D]\!]\,(\mu\,E)\,(u\,i) & \xrightarrow[\alpha]{} & \mu\,E\,(u\,i) \end{array}$$

We can therefore reduce the problem of describing the commuting square of catamorphisms (*i.e.* the Patch of $(\!|\alpha|\!)$) to the one consisting in working directly with their algebras. In effect, we are going to characterise the algebras α^{++} whose catamorphism is coherent by construction (*i.e.* inhabits the Patch of $(\!|\alpha|\!)$).

△

6.2 Remark. We have established that if the square composed of the algebras commutes, then the square composed of their catamorphisms commutes. However, the converse does not hold: having that the square of catamorphisms commutes does not necessarily imply that the square of algebras commutes.

◊

6.3 Example (Lifting isSuc)**.** To illustrate our approach, let us work through a concrete example: we derive $\mathsf{hd} : \mathsf{List}\,A \to \mathsf{Maybe}\,A$ from $\mathsf{isSuc} : \mathsf{Nat} \to \mathsf{Bool}$ by transporting the algebra. For the sake of argument, we artificially define isSuc by a catamorphism:

$$\begin{array}{l} \mathsf{isSuc}\,(n : \mathsf{Nat}) \;:\; \mathsf{Bool} \\ \mathsf{isSuc}\quad n \quad \mapsto (\!|\mathsf{isSucAlg}|\!)\,n \quad \textbf{where} \\ \qquad \mathsf{isSucAlg}\,(xs : [\![\mathsf{Nat\text{-}func}]\!]\,(\lambda *.\,\mathsf{Bool})\,*) \;:\; \mathsf{Bool} \\ \qquad \mathsf{isSucAlg} \qquad (\text{'}0, *) \qquad \mapsto \mathsf{false} \\ \qquad \mathsf{isSucAlg} \qquad (\text{'}\mathsf{suc}, xs) \qquad \mapsto \mathsf{true} \end{array}$$

Our objective is thus to define the algebra for hd, which has the following type

$$\mathsf{hdAlg} : [\![\mathsf{List\text{-}func}\,A]\!]\,(\lambda *.\,\mathsf{Maybe}\,A)\,* \to \mathsf{Maybe}\,A$$

such that its catamorphism is coherent. By the fold-fusion theorem, it is sufficient for hdAlg to satisfy the following condition:

$$\begin{array}{l} (ms : [\![\mathsf{List\text{-}func}\,A]\!]\,(\lambda *.\,\mathsf{Maybe}\,A)\,*) \to \\ \quad \mathsf{isJust}\,(\mathsf{hdAlg}\,ms) = \mathsf{isSucAlg}\,(\mathsf{forgetNT}\,(\mathsf{List\text{-}Orn}\,A)\,([\![\mathsf{List\text{-}func}\,A]\!]^{\rightarrow}\,\mathsf{isJust}\,ms)) \end{array}$$

Following the same methodology we applied to define the Patch type, we can massage the type of hdAlg and its coherence condition to obtain an isomorphic definition enforcing the coherence by indexing. In this case, we obtain the type

$$\mathsf{lift}_{\mathsf{Alg}}\,\mathsf{hdAlg} \triangleq \forall n.\, [\![\mathsf{Vec\text{-}func}\,A]\!]\,(\lambda n'.\,\mathsf{IMaybe}\,A\,(\mathsf{isSuc}\,n'))\,n \to \mathsf{IMaybe}\,A\,(\mathsf{isSuc}\,n)$$

△

This construction generalises to any functional ornament.

6.4 Definition (Coherent algebra [57])**.** We define the *coherent algebras* over an algebra α to be the inhabitants of the type

DEFINITION

$$\begin{aligned}&\mathsf{lift}_{\mathsf{Alg}}\,(\alpha : \forall k : K.\, [\![D]\!]\,(\lambda - .\, [\![T]\!]_{\mathsf{Type}})\,k \to [\![T]\!]_{\mathsf{Type}})\,(o : \mathsf{orn}\,D\,u)\,(T^+ : \mathsf{FunOrn}\,T) : \mathsf{SET}\\ &\mathsf{lift}_{\mathsf{Alg}}\,\alpha\,o\,T^+ \mapsto \forall (i,t) : (i : I) \times \mu\,D\,(u\,i).\\ &\qquad\qquad [\![\lceil o \rceil]\!]_{\mathsf{orn}}\,(\lambda(i,t).\,\mathsf{Patch}\,T\,T^+\,((\!|\alpha|\!)\,t))\,(i,t) \to \mathsf{Patch}\,T\,T^+\,((\!|\alpha|\!)\,t)\end{aligned}$$

▽

6.5 Definition (Lifting of coherent algebra [57])**.** Constructively, we get that coherent algebras induce coherent liftings, by the catamorphism of the coherent algebra:

DEFINITION

$$\begin{aligned}&\mathsf{lift\text{-}fold}\,(\alpha : \forall k : K.\, [\![D]\!]\,(\lambda - .\, [\![T]\!]_{\mathsf{Type}})\,k \to [\![T]\!]_{\mathsf{Type}})\\ &\qquad (\alpha^{++} : \mathsf{lift}_{\mathsf{Alg}}\,\alpha\,o\,T^+) : \mathsf{Patch}\,(\mu\{D \cdot u\,i\} \rightarrowtail T)\,(\mu^+\{o \cdot \mathsf{inv}\,i\} \rightarrowtail T^+)\,(\!|\alpha|\!)\\ &\mathsf{lift\text{-}fold}\,\alpha\,\alpha^{++} \mapsto \lambda x.\,\lambda x^{++}.\,(\!|\alpha^{++}|\!)\,x^{++}\end{aligned}$$

▽

The treatment of induction is essentially the same, as hinted at by the fact that induction can be reduced to a catamorphism (Hermida & Jacobs, 1998). We first define the coherent inductive step and deduce an operation lifting induction principles:

[57] MODEL: `FunOrn.Lift.Fold`

6.6 Definition (Coherent inductive step [58])**.** We define the *coherent inductive step* over an inductive step α to be the inhabitants of the type

DEFINITION

$$\begin{aligned}
&\mathsf{lift_{IH}}\,(\alpha : (k:K)(xs:[\![D]\!]\,(\mu\,D)\,k) \to \Box_D\,(\lambda_.\,[\![T]\!]_{\mathsf{Type}})\,xs \to [\![T]\!]_{\mathsf{Type}})\\
&\qquad (o : \mathsf{orn}\,D\,u)(T^+ : \mathsf{FunOrn}\,T)\ :\ \textsc{Set}\\
&\mathsf{lift_{IH}}\,\alpha\,o\,T^+ \mapsto ((i,t) : (i:I) \times \mu\,D\,(u\,i)) \to (xs : [\![\lceil o \rceil]\!]_{\mathsf{orn}}\,(\mu\,[\![\lceil o \rceil]\!]_{\mathsf{orn}})\,(i,t)) \to\\
&\qquad\qquad \Box_{[\![\lceil o \rceil]\!]_{\mathsf{orn}}}\,(\lambda(i,t).\,\mathsf{Patch}\,T\,T^+\,(\mathsf{induction}\,\alpha\,t))\,xs \to\\
&\qquad\qquad \mathsf{Patch}\,T\,T^+\,(\mathsf{induction}\,\alpha\,t)
\end{aligned}$$

▽

6.7 Definition (Lifting of inductive step [58])**.** As for algebras, a coherent inductive step α^{++} induces a coherent lifting, by merely applying the induction:

DEFINITION

$$\begin{aligned}
&\mathsf{lift\text{-}ind}\,(\alpha : (k:K)(xs:[\![D]\!]\,(\mu\,D)\,k) \to \Box_D\,(\lambda_.\,[\![T]\!]_{\mathsf{Type}})\,xs \to [\![T]\!]_{\mathsf{Type}})\\
&\qquad (\alpha^{++} : \mathsf{lift_{IH}}\,\alpha\,o\,T^+)\ :\ \mathsf{Patch}\,(\mu\{D \cdot u\,i\} \rightarrowtail T)\,(\mu^+\{o \cdot \mathsf{inv}\,i\} \rightarrowtail T^+)\,(\mathsf{induction}\,\alpha)\\
&\mathsf{lift\text{-}ind}\,\alpha\,\alpha^{++} \mapsto \lambda x.\,\lambda x^{++}.\,\mathsf{induction}\,\alpha^{++}\,x^{++}
\end{aligned}$$

▽

6.8 Definition (Lifting of case analysis[59])**.** Lifting case analysis is trivial, since it is derivable from induction by stripping out the induction hypotheses (McBride *et al.*, 2004):

DEFINITION

$$\begin{aligned}
&\mathsf{lift\text{-}case}\,(\alpha : (k:K)(xs:[\![D]\!]\,(\mu\,D)\,k) \to [\![T]\!]_{\mathsf{Type}})\\
&\qquad (\alpha^{++} : \mathsf{lift_{IH}}\,(\lambda xs\ _.\,\alpha\,xs)\,o\,T^+)\ :\ \mathsf{Patch}\,(\mu\{D \cdot u\,i\} \rightarrowtail T)\\
&\qquad\qquad\qquad\qquad (\mu^+\{o \cdot \mathsf{inv}\,i\} \rightarrowtail T^+)\\
&\qquad\qquad\qquad\qquad (\mathsf{induction}\,(\lambda xs\ _.\,\alpha\,xs))\\
&\mathsf{lift\text{-}case}\,\alpha\,\alpha^{++} \mapsto \mathsf{lift\text{-}ind}\,(\lambda xs\ _.\,\alpha\,xs)\,(\lambda xs\ _.\,\alpha^{++}\,xs)
\end{aligned}$$

▽

6.9 Example (Transporting the recursion pattern of isSuc[60])**.** We can now apply our generic machinery to transport isSuc to hd: using a high-level notation, we write the command of Fig. 7(a) (p.52). This command instructs the system to generate the skeleton of the algebra, as shown in Fig. 7(b) (p.52). In the

[58] MODEL: `FunOrn.Lift.Induction`
[59] MODEL: `FunOrn.Lift.Case`
[60] MODEL: `FunOrn.Lift.Examples.Head`

low-level type theory, this corresponds to the following term:

```
ihd (vs : Vec A n)  :  IMaybe A (isSuc n)
ihd      vs        ↦ lift-fold isSucAlg ihdAlg vs
  where
    ihdAlg (vs : [[Vec-func A]] (λn′. IMaybe A (isSuc n′)) n) : IMaybe A (isSuc n)
    ihdAlg      ('nil, ∗)      ↦ {?}
    ihdAlg ('cons, (a, xs)) ↦ {?}
```

Note that we do not need to specify the arguments over which the catamorphism is lifted in Fig. 7(a): indeed, this information is provided by the base function. This is reflected by the elaborated code: the arguments of lift-fold are only the algebras. The resulting catamorphism is applied to the correct variables, by construction.

△

6.10 Remark (High-level notation). Formalising the elaboration process from the high-level notation to the low-level type theory is beyond the scope of this article. The reader will convince herself that the high-level notation contains all the information necessary to reconstruct a low-level term. Indeed, when lifting a recursion pattern, the goal – a Patch type – provides all the information concerning the functional ornament being constructed, including the algebra that is being lifted. Similarly, when lifting a constructor, the goal – still a Patch type – carries the functional ornament as well as the constructor being lifted. We shall use the high-level notation to succinctly describe our transformations, with the understanding that it builds a low-level term that type checks. We come back the practicality of such a notation in Remark 6.22.

◊

6.11 Example (Transporting the recursion pattern of $-<-$[61])**.** To implement ilookup, we use lift-ind to transport the induction over n:

```
ilookup : Patch type< typeLookup (− < −)
ilookup    m m⁺    n        vs      ⇐lift lift-ind
  ilookup m m⁺    0        nil     {?}
  ilookup m m⁺ (suc n) (cons a vs) {?}
```

Followed by a lift-case to transport the case analysis over m:

```
ilookup : Patch type< typeLookup (− < −)
ilookup        m      m⁺      n          vs        ⇐lift lift-ind
  ilookup      m      m⁺      0          nil       {?}
  ilookup      m      m⁺   (suc n) (cons a vs) ⇐lift lift-case
    ilookup    0      0       0          nil       {?}
    ilookup (suc m) (suc m⁺)   0          nil       {?}
```

The interactive nature of this construction is crucial: the user needs only to specify a lifting – symbolised by the $\overset{\text{lift}}{\Leftarrow}$ command – together with the action to be carried out, while the computer does all the heavy lifting and generates the resulting patterns.

△

[61] MODEL: `FunOrn.Lift.Examples.Lookup`

6.12 Example (Transporting the recursion pattern of $-+-$[62])**.** In order to implement the concatenation of vectors, we can also benefit from our generic machinery. We simply have to instruct the machine that we want to lift the case analysis used in the definition of $-+-$ and the computer comes back to us with the following goals:

```
vappend : Patch type+ type++ −+−
vappend      m        xs      n    ys    ⇐[lift] lift-ind
  vappend    0        nil     n    ys    {?}
  vappend (suc m) (cons a xs) n    ys    {?}
```

△

6.2 Transporting constructors

Just as the recursive structure, the returned values frequently mirror the original definition: we are often in a situation where the base function returns a given constructor and we would like to return its ornamented counterpart. Informing the computer that we simply want to lift that constructor, it should fill in the parts that are already determined and ask only for the missing information, *i.e.* the data newly introduced by the ornament.

Recall that, when constructing an inhabitant of a Patch type, we are working on the reornaments of the lifting. When returning a constructor-headed value, we are simply building an inhabitant of a reornament (Definition 5.15, case $\mu^+\{o \cdot \mathsf{inv}\, i\}\times$). By definition of reornaments (Section 4.5), all the information provided by the non-ornamented datatype – the index – is optimally used: every opportunity for deletion has been taken. In particular, none of the data provided by index – the non-ornamented data – needs to be duplicated within the reornamented datatype: only the extensions introduced by the ornament need to be provided.

This suggests a decomposition of the inhabitants of reornaments in two components:

- Giving the *extension* of the ornament, *i.e.* all the extra information introduced by the ornament;
- Giving the recursive arguments – dictated by its *structure* – of the reornamented datatype.

6.13 Example (Inhabiting the reornament of List-Orn)**.** Let us illustrate this decomposition on the reornament of List-Orn. The reornament of List-Orn is structured as follows:

- We retrieve the index, hence obtaining a number $n : \mathsf{Nat}$;
- By inspecting the ornament List-Orn, we obtain the *exact* information by which n is *extended* into a list: if $n = 0$, no supplementary information is needed; if $n = \mathsf{suc}\, n'$, we need to extend it with an $a : A$. We call this the extension – denoted Extension – of List-Orn at n:

$$\mathsf{Extension}\,(\mathsf{List\text{-}Orn}\,A\,*)\,(\text{'}0, *) \rightsquigarrow \mathbb{1}$$
$$\mathsf{Extension}\,(\mathsf{List\text{-}Orn}\,A\,*)\,(\text{'}\mathsf{suc}, n') \rightsquigarrow (a : A) \times \mathbb{1}$$

- By inspecting the ornament List-Orn again, and provided an extension of the index n, we obtain the recursive structure of the reornament at that index by extracting the *refined* indexing discipline: if $n = 0$, no argument is expected; if $n = \mathsf{suc}\, n'$, we expect the tail to be a vector of size n'. We call

[62] MODEL: `FunOrn.Lift.Examples.Append`

this the (recursive) structure – denoted Structure – of List-Orn at n:

$$\mathsf{Structure}\,(\mathsf{List\text{-}Orn}\,A\,*)\,(\text{'}0,*)\,* \rightsquigarrow \mathbb{1}$$
$$\mathsf{Structure}\,(\mathsf{List\text{-}Orn}\,A\,*)\,(\text{'}\mathsf{suc},n')\,(a,*) \rightsquigarrow \mu\,[\![\lceil \mathsf{List\text{-}Orn}\,A\rceil]\!]_{\mathsf{orn}}\,(*,n')$$

Provided an extension – an inhabitant e of $\mathsf{Extension}\,(\mathsf{List\text{-}Orn}\,A\,*)\,n$ – and its recursive arguments – an inhabitant of $\mathsf{Structure}\,(\mathsf{List\text{-}Orn}\,A\,*)\,n\,e$, we have all the necessary data to inhabit of reornament of List-Orn (*i.e.*, to inhabit a vector indexed by n).

△

This decomposition generalises to any reornament. We define the *extension* of an ornament (Definition 6.14), and the *structure* of its reornament (Definition 6.15). Given an extension and its structure, we then show how to inhabit a reornament (Definition 6.16). In what follows, we take $E : \mathsf{func}\,I$ to be a description and $o : \mathsf{orn}\,E\,u$ to be an ornament of E.

6.14 Definition (Reornament extension [63])**.** The extension of an ornament is given by its insert codes. Therefore, the Extension function merely collects these insertions (the insert case). It also makes sure that the indexing respects the previously deleted data through equations (the delete case). On a Σ case, no data is required: the information is already available from the index. Otherwise, it proceeds purely structurally (the var, 1, and Π cases).

DEFINITION

$$\begin{array}{lllll}
\multicolumn{5}{l}{\mathsf{Extension}\,(O : \mathsf{Orn}\,D\,u)\,(xs : [\![D]\!]\,(\mu\,E)) \;:\; \mathrm{SET}} \\
\multicolumn{5}{l}{\quad \textit{– Ask for freshly inserted data, it is the data missing from } xs\textit{:}} \\
\mathsf{Extension} & (\mathsf{insert}\,S\,D^+) & xs & \mapsto & (s:S) \times \mathsf{Extension}\,(D^+\,s)\,xs \\
\multicolumn{5}{l}{\quad \textit{– Do not duplicate the original data, it is already in } xs\textit{:}} \\
\mathsf{Extension} & (\mathsf{var}\,(\mathsf{inv}\,i)) & xs & \mapsto & \mathbb{1} \\
\mathsf{Extension} & 1 & * & \mapsto & \mathbb{1} \\
\mathsf{Extension} & (\Pi\,T^+) & f & \mapsto & (s:S) \to \mathsf{Extension}\,(T^+\,s)\,(f\,s) \\
\mathsf{Extension} & (\Sigma\,T^+) & (s,xs) & \mapsto & \mathsf{Extension}\,(T^+\,s)\,xs \\
\multicolumn{5}{l}{\quad \textit{– Deleted data must be consistent with the index:}} \\
\mathsf{Extension} & (\mathsf{delete}\,s\,T^+) & (s',xs) & \mapsto & (q : s = s') \times \mathsf{Extension}\,T^+\,xs
\end{array}$$

Note that while we copy the Π codes as a Π-type, we are not *duplicating* information: the codomain of the Π-type contains only new information, or is isomorphic to $\mathbb{1}$ (in which case, the whole Π-type is itself isomorphic to $\mathbb{1}$).

▽

6.15 Definition (Reornament structure [63])**.** We capture the recursive structure of the reornament by traversing the ornament while peeling off the index xs along the way. The crucial step is the variable case (case var): we ask for a reornament $\mu\,[\![\lceil o\rceil]\!]_{\mathsf{orn}}$ taken at the index specified by the ornament and the current index xs. The other cases are only creating the necessary scaffolding to cover all the recursive arguments.

[63] MODEL: `FunOrn.Lift.MkReorn`

DEFINITION

```
Structure (O : Orn D u) (xs : ⟦D⟧ (μ E)) (e : Extension O xs) : SET
  – Extract the next index from the ornament and the index:
Structure (var (inv i))    xs      *      ↦ μ ⟦⌈o⌉⟧orn (i , xs)
  – Duplicate (only) the recursive structure:
Structure (insert S D⁺)    xs      (s , e)  ↦ Structure (D⁺ s) xs e
Structure 1                *       *        ↦ 𝟙
Structure (Π T⁺)           f       e        ↦ (s : S) Structure (T⁺ s) (f s) (e s)
Structure (Σ T⁺)           (s , xs) e       ↦ Structure (T⁺ s) xs e
Structure (delete s T⁺)    (s , xs) (refl , e) ↦ Structure T⁺ xs e
```

▽

The decomposition of the reornament in terms of an extension and its recursive structure is formally expressed by the following isomorphism:

META-THEOREM

Let $D : \mathsf{IDesc}\, I$ be a description code and let $O : \mathsf{Orn}\, D\, u$ be an ornament code of D.
For all $xs : \llbracket D \rrbracket\, (\mu\, E)$, we have:

$$(e : \mathsf{Extension}\, O\, xs) \times \mathsf{Structure}\, O\, xs\, e \quad \cong \quad \llbracket \llbracket \lceil O \rceil \rrbracket_{\mathsf{orn}}\, xs \rrbracket_{\mathsf{orn}}\, (\mu\, o)$$

In practice, we are interested in the left-to-right direction of the isomorphism, which allows us to inhabit a reornament by focusing on the extension introduced by the ornament and its recursive arguments. Constructively, this translates into the mkReorn function below.

6.16 Definition (Inhabitation of reornaments[64])**.** For a given ornament $O : \mathsf{Orn}\, D\, u$ and some index $xs : \llbracket D \rrbracket\, (\mu\, E)$, we can therefore combine an extension of O at xs with its recursive arguments to inhabit the reornament of O. To do so, we proceed by case analysis over the ornament O: we find that the data required by the reornament is either provided by the extension (case insert), or the structure (case var). The other cases contain no data per se, only the recursive structure of the datatype.

DEFINITION

```
mkReorn (O : Orn D u) (xs : ⟦D⟧ (μ E))
        (e : Extension O xs) (a : Structure O xs e) : ⟦⌈O⌉ xs⟧orn (μ ⟦⌈o⌉⟧orn)
mkReorn (insert S D⁺)    xs      (s , e)  a ↦ (s , mkReorn (D⁺ s) e a)
mkReorn (var (inv i))    xs      *        a ↦ a
mkReorn 1                *       *        * ↦ *
mkReorn (Π T⁺)           f       e        a ↦ λ s . mkReorn (T⁺ s) (f s) (e s) (a s)
mkReorn (Σ T⁺)           (s , xs) e       a ↦ mkReorn (T⁺ s) xs e a
mkReorn (delete s T⁺)    (s , xs) (refl , e) a ↦ (refl , mkReorn T⁺ xs e a)
```

[64] MODEL: `FunOrn.Lift.MkReorn`

▽

6.17 Definition (Lifting of constructor[65])**.** This clear separation of concerns is a blessing for us: when lifting a constructor, we only have to provide the extension and the arguments of the datatype, nothing more. The implementation is as simple as:

DEFINITION

```
lift-constructor (e : Extension (o i) xs)                    – coherent extension
                 (a : Structure (o i) xs e)                  – recursive arguments
                 (t^++ : Patch T T^+ t)
                 : Patch (μ{D · u i}× T) (μ^+{o · inv i}× T^+) (in xs, t)
lift-constructor e a t^++ ↦ (in (mkReorn (o i) xs e a), t^++)
```

For convenience, we extend our high-level notation with a gadget for returning a lifted constructor. We shall therefore write

$$\ldots \overset{\text{lift}}{\mapsto} e[a]$$

to denote the low-level term

$$\text{lift-constructor}\; e\; a\; *$$

A few examples illustrating this notation follow (Example 6.18 and Example 6.19).

▽

6.18 Example (Transporting the constructors of isSuc[66])**.** Let us finish the implementation of hd from isSuc. Our task is simply to transport the true and false constructors along the Maybe ornament. In a high-level notation, we would write the command shown in Fig. 7(c) (p.52). The interactive system would then respond by generating the code of Fig. 7(d) (p.52). The unit goals are trivially solved. The only information the user has to provide is a value of type A, which is required by the just constructor.

△

6.19 Example (Transporting the constructors of $- < -$[67])**.** As for ilookup, we want to lift the constructors true and false to the Maybe ornament. In a high-level notation, this would be represented as follows:

```
ilookup : Patch type< typeLookup (− < −)
ilookup          m    m^+   n       vs          ⇐[lift] lift-ind
  ilookup        m    m^+   0       nil         ↦[lift] nothing ∗[∗]
  ilookup        m    m^+   (suc n) (cons a vs) ⇐[lift] lift-case
    ilookup      0    0     (suc n) (cons a vs) ↦[lift] just {a : A} [∗]
    ilookup (suc m) (suc m^+) (suc n) (cons a vs) {?}
```

As before, in an interactive setting, the user would instruct the machine to execute the command $\overset{\text{lift}}{\mapsto}$ and the computer would come back with the skeleton of the expected inputs. Finishing the implementation of

[65] MODEL: `FunOrn.Lift.Constructor`
[66] MODEL: `FunOrn.Lift.Examples.Head`
[67] MODEL: `FunOrn.Lift.Examples.Lookup`

ilookup is now a baby step away for the programmer:

```
ilookup : Patch type< typeLookup (− < −)
ilookup          m    m⁺    n        vs        ⇐[lift] lift-ind
  ilookup        m    m⁺    0        nil       ↦[lift] nothing ∗[∗]
  ilookup        m    m⁺    (suc n)  (cons a vs) ⇐[lift] lift-case
    ilookup      0    0     (suc n)  (cons a vs) ↦[lift] just a[∗]
    ilookup (suc m) (suc m⁺) (suc n) (cons a vs) ↦ ilookup m m⁺ n vs
```

This last step is out of reach of our framework: the recursive call is justified by the first induction over the vector *vs*, which gives us access to an higher-order induction hypothesis (taking natural numbers to the IMaybe type). We have to fully apply this induction hypothesis to *m* and *im* to perform the recursive call, an operation for which can offer no support.

△

6.20 Remark (Lifting *vs.* manual construction). Had we ignored the structural ties between $-<-$ and lookup, we would have constructed ilookup by duplicating its underlying structure:

```
ilookup : Patch type< typeLookup (− < −)
ilookup          m    m⁺    n        vs        ⇐ Vector-elim vs
  ilookup        m    m⁺    0        nil       ↦ nothing
  ilookup        m    m⁺    (suc n)  (cons a vs) ⇐ Nat-case m⁺
    ilookup      0    0     (suc n)  (cons a vs) ↦ just a
    ilookup (suc m) (suc m⁺) (suc n) (cons a vs) ↦ ilookup m m⁺ n vs
```

That is, we would have duplicated the induction over the second argument (… ⇐ Vec-elim *vs*), and the induction over the first argument (… ⇐ Nat-elim *m*). We would also have duplicated the returned constructor (… ↦ nothing and … ↦ just *a*).

Ignoring the structural ties has two consequences. First, this provides less opportunities for the system to guide the implementation of ilookup: the search space is less constrained. Second, having lost the connection between $-<-$ and ilookup, a modification of the former will not impact the latter. In the course of a development, the two implementations might diverge in incompatible ways.

◊

6.21 Example (Transporting the constructors of $-+-$[68])**.** We can also benefit from the automatic lifting of constructors to fill out the cons case of vector append. We instruct the system that we want to lift the suc constructor, which results in the following goals:

```
vappend : Patch type+ type++ − + −
vappend    m       xs    n   ys    ⇐[lift] lift-ind
vappend    0       nil   n   ys    {?}
vappend (suc m) (cons a xs) n ys    ↦[lift] cons {?:A} [ {?} ]
```

[68] MODEL: `FunOrn.Lift.Examples.Append`

Concluding the implementation of vappend is then left to the programmer:

```
vappend : Patch type+ type++ − + −
vappend   m      xs     n    ys    ⇐[lift] lift-ind
vappend   0      nil    n    ys    ↦ ys
vappend (suc m) (cons a xs) n ys    ↦[lift] cons a [vappend m xs n ys]
```

△

(a) Request lifting of algebra (user input):

```
ihd (vs : Vec A n)        :        IMaybe A (isSuc n)
ihd                 {⇐[lift] lift-fold}
```

(b) Result of lifting the algebra (system output):

```
ihd (vs : Vec A n) : IMaybe A (isSuc n)
ihd                ⇐[lift] lift-fold  where
  ihdAlg (vs : ⟦Vec-func⟧ (λn′. IMaybe A (isSuc n′)) n) : IMaybe A (isSuc n)
  ihdAlg                 'nil                  {?}
  ihdAlg             ('cons a xs)              {?}
```

(c) Request lifting of constructors (user input):

```
ihd (vs : Vec A n) : IMaybe A (isSuc n)
ihd                ⇐[lift] lift-fold  where
  ihdAlg (vs : ⟦Vec-func⟧ (λn′. IMaybe A (isSuc n′)) n) : IMaybe A (isSuc n)
  ihdAlg                 'nil                  {↦[lift]}
  ihdAlg             ('cons a xs)              {↦[lift]}
```

(d) Result of lifting constructors (system output)

```
ihd (vs : Vec A n) : IMaybe A (isSuc n)
ihd                ⇐[lift] lift-fold  where
  ihdAlg (vs : ⟦Vec A⟧ (λn′. IMaybe A (isSuc n′)) n) : IMaybe A (isSuc n)
  ihdAlg                 'nil                  ↦[lift] nothing {?:𝟙} [ {?:𝟙} ]
  ihdAlg             ('cons a xs)              ↦[lift] just {?:A} [ {?:𝟙} ]
```

(e) Type checked term (automatically generated from (d)):

```
ihd (vs : Vec A n) : IMaybe A (isSuc n)
ihd vs ↦ lift-fold isSucAlg ihdAlg  where
  ihdAlg (vs : ⟦Vec A⟧ (λn′. IMaybe A (isSuc n′)) n) : IMaybe A (isSuc n)
  ihdAlg     'nil      ↦ lift-constructor 'nil {?:𝟙} {?:𝟙} ∗
  ihdAlg ('cons a xs) ↦ lift-constructor ('suc n) {?:A} {?:𝟙} ∗
```

Fig. 7: Guided implementation of ihd

6.22 Remark (About an interactive system). To convey our message to the reader, we have used an (informal) notation, extending the Epigram programming gadgets with lifting-specific gadgets. We have not given much information about their implementability, or even hinted at a formal specification. It would certainly be interesting to elaborate on such a language extension. However, as for the notation for ornaments (Section 4.1), this notation was meant to convey high-level intuitions *to the reader*, and to keep us from flooding these pages with lambda terms.

In terms of implementation, elaborating this notation (or a less ambiguous version thereof) might reveal arduous. A more pragmatic alternative would be to implement a semi-decision procedure à la Agsy (Lindblad & Benke, 2004) that would attempt to automatically lift a function. In this setting, the lifting constructors we have defined in this section serve as a precise language in which to express the lifting problem, and over which to compute its solution. They thus narrow the search-space and guide the decision procedure.

◊

7 Related Work

Our work is an extension of the work of McBride (2013) on ornaments, originally introduced to organise datatypes according to their common structure. This gave rise to the notion of ornamental algebras – forgetting the extra information of an ornamented datatype – and algebraic ornaments – indexing a datatype according to an algebra. This, in turn, induced the notion of algebraic ornament by ornamental algebras, which is a key ingredient for our work. However, for simplicity of exposition, these ornaments had originally been defined on a less index-aware universe of datatypes. As a consequence, computation over indices was impossible and deletion of duplicated information was impossible. A corollary of this was that reornaments contained a lot of duplication, hence making the lifting of values from ornamented to reornamented datatypes extremely tedious.

Our presentation of algebraic ornament has been greatly improved by the categorical model developed by Atkey *et al.* (2012): the authors gave a conceptually clear treatment of algebraic ornament in a Lawvere fibration. At the technical level, the authors connected the definition of algebraic ornament with truth-preserving liftings, which are also used in the construction of induction principles, and op-reindexing, which models Σ-types in type theory. Whilst the authors did not explicitly address the issue of transporting functions across ornaments, much of the infrastructure was implicitly there: for instance, lifting of catamorphisms is a trivial specialisation of induction.

In their work on realizability and parametricity for Pure Type Systems, Bernardy and Lasson (2011) have shown how to build a logic from a programming language. In such a system, terms of type theory can be precisely segregated based on their computational and logical contribution. In particular, the idea that natural numbers realise lists of the corresponding length appears in this system under the guise of vectors, the reflection of the realizability predicate. The strength of the realizability interpretation is that it is naturally defined on functions: while McBride (2013) and Atkey *et al.* (2012) only consider ornaments on datatypes, Bernardy and Lasson's work is the first, to our knowledge, to capture a general notion of functions realising – *i.e.* ornamenting – other functions.

Bernardy and Moulin has further shown that this technique can be internalised in a type theory with color (Bernardy & Guilhem, 2013), for which the logical system and the programming language are a single entity. In this setting, the realizability predicate is specialised to an (internalised) parametricity result. This parametricity result gives "theorems for free" (Wadler, 1989), relating functions operating on colored types (akin to our functional ornaments) to functions operating on color-erased types (akin to our base types). On inductive types, colors allow a user to specify *restrictions* of types (dually to the *extension*

mechanism offered by ornaments), by filtering out some colors from a definition. This difference is mostly methodological. Our work is focused on creating more precise datatypes from less precise ones, and lifting functions from basic types to richer types: hence our focus on extensions. Being guided by erasure, Bernardy and Moulin focus on extracting functions on less precise types from the more informative ones: hence their focus on restrictions. In our setting, the erasure-based approach corresponds to the right-to-left direction of the Patch isomorphism (Theorem 5.19). The *refinement* mechanism offered by ornaments seems absent from the initial proposal of the calculus of colored constructions: this suggests a natural generalisation of the calculus with inductive families. Our base types and functional ornaments are however limited to a first-order, simply-typed setting, whilst the calculus of colored constructions defines erasure for Π-types and Σ-types: it would be interesting to extend our universe of function types and functional ornaments in that direction (following Remark 5.2).

Following the steps of Bernardy and Lasson, Ko and Gibbons (2011) adapted the realizability interpretation to McBride's universe of datatypes and explored the other direction of the Patch isomorphism (Theorem 5.19), using reornaments to generate coherence properties: they describe how one could take list append together with a proof that it is coherent with respect to addition and obtain the vector append function. Their approach would shift neatly to our index-aware setting, where the treatment of reornaments is streamlined by the availability of deletion.

However, we prefer to exploit the direction of the isomorphism which internalises coherence: we would rather use the full power of dependent types to avoid explicit proof. Hence, in our framework, we simultaneously induce list append and implicitly prove its coherence with addition just by defining vector append. Of course, which approach is appropriate depends on one's starting point. Moreover, our universe of function types takes a step beyond the related work by supporting the mechanised construction of liftings, leaving to the user the task of supplying a minimal patch. Our framework could easily be used to mechanise the realizability predicates of Bernardy and Lasson (2011), and Ko and Gibbons (2011).

8 Conclusion

In this article, we have adapted McBride's ornaments to our universe of datatypes, a cosmetic evolution of an earlier presentation (Chapman *et al.*, 2010) . This gave us the ability to compute over indices, hence introducing the deletion ornament. Deletion ornaments are a key ingredient for the internalisation of Brady's optimisation over inductive families. By applying these ideas, we obtained a simpler implementation of reornaments.

We then developed the notion of functional ornament as a generalisation of ornaments to functions: from a universe of function type, we define a functional ornament as the ornamentation of each of its inductive components. A function of the resulting type will be subject to a coherence property, akin to the ornamental forgetful map of ornaments. We have constructively presented this object by building a small universe of functional ornaments.

We have finally shown how to achieve code reuse by transporting functions along a functional ornament in such a way that the coherence property holds. Instead of asking the user to write cumbersome proofs, we defined a Patch type as the type of all the functions that satisfies the coherence property by construction. Hence, we make extensive use of the dependently-typed programming machinery: in this setting, the type checker, that is the computer, is working with us to construct a term, not waiting for us to produce a proof.

Having implemented a function correct by construction, one then gets, for free, the lifting and its coherence certificate. This is a straightforward application of the isomorphism between the Patch type

and the set of coherent functions. These projection functions have been implemented in type theory by simple generic programming over the universe of functional ornaments.

To further improve code reuse, we provide a few smart constructors to implement a Patch type: the idea is to use the structure of the base function to guide the implementation of the coherent lifting. Hence, if the base function uses a specific induction principle or returns a specific constructor, we make it possible for the user to specify that she wants to lift this structure one level up. This way, the function is not duplicated: only the new information, as determined by the ornament, is necessary.

To conclude, we believe that this is a first yet interesting step toward code reuse for dependently-typed programming systems. With ornaments, we were able to organise datatypes by their structure. With functional ornaments, we are now able to organise functions by their structure-preserving computational behaviour. For a large class of functional ornaments, the original program and its lifting share a similar recursion pattern and returned value. To take advantage of this structural similarity, we have developed some appealing automation to assist their implementation, without any proving required, hence making this approach even more accessible.

Future work Whilst we have deliberately chosen a simple universe of types, we plan to extend it in various directions. Supporting higher-order functions and adding type dependency (Π-types and Σ-types) is a necessary first step. Inspired by Bernardy and Lasson (2011), we would like to add a parametric quantifier: in the implementation of ilookup, we would mark the index A of List A as parametric so that in the cons a case, the a could automatically be carried over.

The universe of functional ornaments could be extended as well, especially once the universe of function types has been extended with dependent quantifiers. For instance, we want to consider the introduction and deletion of quantifiers, as we are currently doing on datatypes. Whilst we have only looked at least fixpoints in this article, we also want to generalise our universe with greatest fixpoints and the lifting of co-inductive definitions.

Besides enriching the universe, its systematic exploration offers exciting prospects. For pedagogical reasons, this article is confined to a handful of ornaments (natural numbers to lists, Booleans to the option type, *etc.*) and focuses on two functional ornaments (the comparison and the addition). A quick comparison between natural numbers and lists reveals a few other opportunities, such as the $-\,-$/drop, null/head, or pred/tl. Keeping with non-indexed data-structures, a whole zoo of ornaments is available for tree-like structures, depending on whether one stores the data at the leaves, nodes, or both. One could imagine writing the structural operations on the bare structure and to instantiate them for specific data-storage strategies by lifting. Finally, many gems are awaiting us in the indexed setting, the true raison d'être of ornaments. For example, we would like to exploit the fact that the simply-typed lambda-calculus (Example 3.10) is an ornament of the untyped one and lift its substitution operators from the one defined over the untyped calculus.

Acknowledgements We owe many thanks to the ICFP and JFP reviewers, their comments having significantly improved this article. We are also very grateful to Guillaume Allais, Stevan Andjelkovic and Peter Hancock for their meticulous reviews of this paper. We shall also thank Edwin Brady for suggesting the study of lookup functions and Andrea Vezzosi for spotting an issue in our definition of reornaments. Finally, this paper would have remained a draft without the help and encouragement of José Pedro Magalhães. The authors are supported by the Engineering and Physical Sciences Research Council, Grant EP/G034699/1.

References

Atkey, Robert, Johann, Patricia, & Ghani, Neil. (2012). Refining inductive types. *Logical methods in computer science*, **8**. doi:10.2168/LMCS-8(2:9)2012.

Benton, Nick, Hur, Chung-Kil, Kennedy, Andrew J., & Mcbride, Conor. (2012). Strongly typed term representations in coq. *Journal of automated reasoning*, **49**(2), 141–159. doi:10.1007/s10817-011-9219-0.

Bernardy, Jean-Philippe, & Guilhem, Moulin. (2013). Type-theory in color. *Pages 61–72 of: International conference on functional programming*. doi:10.1145/2500365.2500577.

Bernardy, Jean-Philippe, & Lasson, Marc. (2011). Realizability and parametricity in pure type systems. *Pages 108–122 of: Foundations of software science and computation structures*. doi:10.1007/978-3-642-19805-2_8.

Bird, Richard S., & de Moor, Oege. (1997). *Algebra of programming*. Prentice Hall.

Brady, Edwin, & Hammond, Kevin. (2010). Scrapping your inefficient engine: using partial evaluation to improve domain-specific language implementation. *Pages 297–308 of: International conference in functional programming*. doi:10.1145/1863543.1863587.

Brady, Edwin, McBride, Conor, & McKinna, James. (2003). Inductive families need not store their indices. *Types for proofs and programs*. doi:10.1007/978-3-540-24849-1_8.

Chapman, James, Dagand, Pierre-Évariste, McBride, Conor, & Morris, Peter. (2010). The gentle art of levitation. *Pages 3–14 of: International conference on functional programming*. doi:10.1145/1863543.1863547.

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

Dagand, Pierre-Evariste. (2013). *Reusability and dependent types*. Ph.D. thesis, University of Strathclyde.

Dagand, Pierre-Evariste, & McBride, Conor. (2012). Transporting functions across ornaments. *Pages 103–114 of: International conference on functional programming*. doi:10.1145/2364527.2364544.

Dagand, Pierre-Evariste, & McBride, Conor. (2013a). A categorical treatment of ornaments. *Logics in computer science*. doi:10.1109/LICS.2013.60.

Dagand, Pierre-Evariste, & McBride, Conor. (2013b). Elaborating inductive definitions. *Journées francophones des langages applicatifs*.

Dybjer, Peter. (1994). Inductive families. *Formal aspects of computing*, **6**(4), 440–465. doi:10.1007/BF01211308.

Freeman, Tim, & Pfenning, Frank. (1991). Refinement types for ML. *Pages 268–277 of: Programming language design and implementation*. doi:10.1145/113445.113468.

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

Gonthier, Georges, Mahboubi, Assia, & Tassi, Enrico. (2008). *A Small Scale Reflection Extension for the Coq system*. Research Report RR-6455. INRIA.

Hermida, Claudio, & Jacobs, Bart. (1998). Structural induction and coinduction in a fibrational setting. *Information and computation*, **145**(2), 107–152. doi:10.1006/inco.1998.2725.

Hofmann, M., & Streicher, T. (1994). The groupoid model refutes uniqueness of identity proofs. *Pages 208–212 of: Logic in computer science*. doi:10.1109/LICS.1994.316071.

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

Lindblad, Fredrik, & Benke, Marcin. (2004). A tool for automated theorem proving in Agda. *Pages 154–169 of: Types for proofs and programs*. doi:10.1007/11617990_10.

McBride, Conor. (1999). *Dependently typed functional programs and their proofs*. Ph.D. thesis, LFCS.

McBride, Conor. (2002). Elimination with a motive. *Page 727 of: Types for proofs and programs*. doi:10.1007/3-540-45842-5_13.

McBride, Conor. (2013). Ornamental algebras, algebraic ornaments. *Journal of functional programming*. To appear.

McBride, Conor, & McKinna, James. (2004). The view from the left. *Journal of functional programming*, **14**(1), 69–111. doi:10.1007/11780274_27.

McBride, Conor, Goguen, Healfdene, & McKinna, James. (2004). A few constructions on constructors. *Pages 186–200 of: Types for proofs and programs*. doi:10.1007/11617990_12.

Morris, Peter, Altenkirch, Thorsten, & Ghani, Neil. (2009). A universe of strictly positive families. *International journal of foundations of computer science*, **20**(1), 83–107. doi:10.1142/S0129054109006462.

Paulin-Mohring, Christine. (1989). *Extraction de programmes dans le Calcul des Constructions*. Ph.D. thesis, Université Paris VII.

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*. doi:10.1145/1596550.1596599.

Strub, Pierre-Yves. (2010). Coq modulo theory. *Pages 529–543 of: Computer science logic*. doi:10.1007/978-3-642-15205-4_40.

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*. doi:10.1145/2034773.2034811.

Wadler, Philip. (1989). Theorems for free! *Pages 347–359 of: Conference on functional programming languages and computer architecture*. doi:10.1145/99370.99404.

A Worked example: from comparison to lookup

A.1 The comparison function

(a) Specify the type of the comparison function:

```
type<  :  Type
type<  ↦  μ{Nat-func · ∗}⟶ μ{Nat-func · ∗}⟶ μ{Bool-func · ∗}× 1
```

(b) Implement the comparison function:

```
−          <  −      :  [[type<]]Type
m          <  n      ⇐  Nat-elim n
  m        <  0      ↦  false
  m        <  (suc n)  ⇐  Nat-case m
    0      <  (suc n)  ↦  true
    (suc m)  <  (suc n)  ↦  m < n
```

A.2 The functional ornament

(a) Specify the ornamented type of the lookup function:

```
typeLookup  :  FunOrn type<
typeLookup  ↦  μ⁺{idO Nat-func · inv ∗}⟶ μ⁺{List-Orn A · inv ∗}⟶ μ⁺{Maybe-Orn A · inv ∗}× 1
```

(b) Implement its Patch type:

```
ilookup : Patch type< typeLookup (− < −)
ilookup m m⁺ n vs {?}
```

(c) Lift the induction:

```
ilookup : Patch type< typeLookup (− < −)
ilookup   m m⁺   n        vs          ⇐ (lift) lift-ind
  ilookup m m⁺   0        nil         {?}
  ilookup m m⁺   (suc n)  (cons a vs) {?}
```

(d) Lift the case analysis:

```
ilookup : Patch type< typeLookup (− < −)
ilookup          m          m⁺          n          vs          ⇐ (lift) lift-ind
  ilookup        m          m⁺          0          nil         {?}
  ilookup        m          m⁺          (suc n)    (cons a vs) ⇐ (lift) lift-case
    ilookup      0          0           0          nil         {?}
    ilookup      (suc m)    (suc m⁺)    0          nil         {?}
```

(e) Lift the constructors:

```
ilookup : Patch type< typeLookup (− < −)
ilookup        m    m⁺    n        vs          ⇐[lift] lift-ind
  ilookup      m    m⁺    0        nil         ↦[lift] nothing ∗[∗]
  ilookup      m    m⁺    (suc n)  (cons a vs) ⇐[lift] lift-case
    ilookup    0    0     (suc n)  (cons a vs) ↦[lift] just {a : A} [∗]
    ilookup (suc m) (suc m⁺) (suc n) (cons a vs) {?}
```

(f) Finish the definition:

```
ilookup : Patch type< typeLookup (− < −)
ilookup        m    m⁺    n        vs          ⇐[lift] lift-ind
  ilookup      m    m⁺    0        nil         ↦[lift] nothing ∗[∗]
  ilookup      m    m⁺    (suc n)  (cons a vs) ⇐[lift] lift-case
    ilookup    0    0     (suc n)  (cons a vs) ↦[lift] just a[∗]
    ilookup (suc m) (suc m⁺) (suc n) (cons a vs) ↦ ilookup m m⁺ n vs
```

A.3 Extracting the lookup function

(a) Obtain the lookup function by applying the patch:

```
lookup : ⟦typeLookup⟧FunOrn
lookup ↦ patch typeLookup (− < −) ilookup
```

(b) Obtain a proof of its coherence:

```
cohLookup (n : Nat) (xs : List A) : isJust (lookup n xs) = n < length xs
cohLookup    n          xs        ↦ coherence typeLookup (− < −) ilookup n xs
```

B Definitions

B.1 Universe of descriptions

$$
\begin{array}{ll}
\textbf{data}\ \mathsf{IDesc}\,[I:\mathrm{SET}]:\mathrm{SET}_1\ \textbf{where} & [\![(D:\mathsf{IDesc}\,I)]\!]\,(X:I\to\mathrm{SET}):\mathrm{SET} \\
\quad \mathsf{IDesc}\,I \ni \mathsf{var}\,(i:I) & [\![\mathsf{var}\,i]\!]\quad X \mapsto X\,i \\
\qquad\quad \mid 1 & [\![1]\!]\qquad X \mapsto \mathbb{1} \\
\qquad\quad \mid \Pi\,(S:\mathrm{SET})\,(T:S\to\mathsf{IDesc}\,I) & [\![\Pi\,S\,T]\!]\,X \mapsto (s:S)\to[\![T\,s]\!]\,X \\
\qquad\quad \mid \Sigma\,(S:\mathrm{SET})\,(T:S\to\mathsf{IDesc}\,I) & [\![\Sigma\,S\,T]\!]\,X \mapsto (s:S)\times[\![T\,s]\!]\,X \\
 & \\
\mathsf{func}\,(I:\mathrm{SET}):\mathrm{SET}_1 & [\![(D:\mathsf{func}\,I)]\!]\,(X:I\to\mathrm{SET}):I\to\mathrm{SET} \\
\mathsf{func}\,I \mapsto I\to\mathsf{IDesc}\,I & [\![D]\!]\,X \mapsto \lambda i.\,[\![D\,i]\!]\,X
\end{array}
$$

Code and interpretation

$$
\begin{array}{l}
\textbf{data}\ \mu\,[D:\mathsf{func}\,I]\,(i:I):\mathrm{SET}\ \textbf{where} \\
\quad \mu\,D\,i \ni \mathsf{in}\,(xs:[\![D]\!]\,(\mu\,D)\,i) \\
 \\
\mathsf{induction}:\forall\,P:\forall\,i:I.\,\mu\,D\,i\to\mathrm{SET}. \\
\qquad\qquad (\alpha:(i:I)(xs:[\![D]\!]\,(\mu\,D)\,i)\to\Box_D\,P\,xs\to P\,(\mathsf{in}\,xs)) \\
\qquad\qquad (x:\mu\,D\,i)\to P\,x
\end{array}
$$

Least fixpoint and induction principle

B.2 Universe of ornaments

$$
\begin{array}{l}
\textbf{data}\ \mathsf{Orn}\,(D:\mathsf{IDesc}\,K)[u:I\to K]:\mathrm{SET}_1\ \textbf{where} \\
\quad \textit{– Extend with S:} \\
\quad \mathsf{Orn}\quad D\quad u\quad \ni\quad \mathsf{insert}\,(S:\mathrm{SET})(D^+:S\to\mathsf{Orn}\,D\,u) \qquad [\![(O:\mathsf{Orn}\,D\,u)]\!]_{\mathsf{orn}}\ :\ \mathsf{IDesc}\,I \\
\quad \textit{– Refine index:} \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad [\![\mathsf{insert}\,S\,D^+]\!]_{\mathsf{orn}}\ \mapsto \Sigma\,S\,\lambda s.\,[\![D^+\,s]\!]_{\mathsf{orn}} \\
\quad \mathsf{Orn}\ (\mathsf{var}\,k)\ u\quad \ni\quad \mathsf{var}\,(i:u^{-1}\,k) \qquad\qquad\qquad\qquad\qquad\quad [\![\mathsf{var}\,(\mathsf{inv}\,i)]\!]_{\mathsf{orn}}\ \mapsto \mathsf{var}\,i \\
\quad \textit{– Copy the original:} \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad [\![1]\!]_{\mathsf{orn}}\ \mapsto 1 \\
\quad \mathsf{Orn}\quad 1\quad u\quad \ni\quad 1 \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad [\![\Pi\,T^+]\!]_{\mathsf{orn}}\ \mapsto \Pi\,S\,\lambda s.\,[\![T^+\,s]\!]_{\mathsf{orn}} \\
\quad \mathsf{Orn}\,(\Pi\,S\,T)\,u\quad \ni\quad \Pi\,(T^+:(s:S)\to\mathsf{Orn}\,(T\,s)\,u) \qquad\qquad\qquad [\![\Sigma\,T^+]\!]_{\mathsf{orn}}\ \mapsto \Sigma\,S\,\lambda s.\,[\![T^+\,s]\!]_{\mathsf{orn}} \\
\quad \mathsf{Orn}\,(\Sigma\,S\,T)\,u\quad \ni\quad \Sigma\,(T^+:(s:S)\to\mathsf{Orn}\,(T\,s)\,u) \qquad\qquad\qquad [\![\mathsf{delete}\,s\,T^+]\!]_{\mathsf{orn}}\ \mapsto [\![T^+]\!]_{\mathsf{orn}} \\
\quad \textit{– Delete S:} \\
\qquad\qquad\qquad\quad |\quad \mathsf{delete}\,(s:S)(T^+:\mathsf{Orn}\,(T\,s)\,u) \\
\\
\mathsf{orn}\,(D:\mathsf{func}\,K)\,(u:I\to K)\ :\ \mathrm{SET}_1 \qquad\qquad [\![(o:\mathsf{orn}\,D\,u)]\!]_{\mathsf{orn}}\ :\ \mathsf{func}\,I \\
\mathsf{orn}\,D\,u\mapsto (i:I)\to\mathsf{Orn}\,(D\,(u\,i))\,u \qquad\qquad [\![o]\!]_{\mathsf{orn}}\mapsto\lambda i.\,[\![o\,i]\!]_{\mathsf{orn}}
\end{array}
$$

Code and interpretation

$$
\begin{array}{l}
\mathsf{forgetNT}:(o:\mathsf{orn}\,D\,u)\to[\![o]\!]\ (X\circ u)\,i\to[\![D]\!]\,X\,(u\,i) \\
\mathsf{forgetAlg}:(o:\mathsf{orn}\,D\,u)\to[\![o]\!]_{\mathsf{orn}}\ (\mu\,D\circ u)\,i\to\mu\,D\,(u\,i) \\
\mathsf{forget}:(o:\mathsf{orn}\,D\,u)\to\mu\,[\![o]\!]_{\mathsf{orn}}\,i\to\mu\,D\,(u\,i)
\end{array}
$$

Ornamental algebra

$$
\begin{array}{l}
(D:\mathsf{func}\,K)^{(\alpha:[\![D]\!]\,X\dot{\to}X)}:\mathsf{orn}\,D\,(\pi_0:(k:K)\times X\,k\to K) \\
\mathsf{coherentOrn}:(t^\alpha:\mu\,[\![D^\alpha]\!]_{\mathsf{orn}}(k,x))\to(\!|\alpha|\!)\,(\mathsf{forget}\,D^\alpha\,t^\alpha)=x \\
\mathsf{make}\,(D:\mathsf{func}\,I)^{(\alpha:\forall i.\,[\![D]\!]\,X\,i\to X\,i)}:(t:\mu\,D\,k)\to\mu\,[\![D^\alpha]\!]_{\mathsf{orn}}(k,(\!|\alpha|\!)\,t)
\end{array}
$$

Algebraic ornament

$$
\lceil(o:\mathsf{orn}\,D\,u)\rceil:\mathsf{orn}\,[\![o]\!]_{\mathsf{orn}}\,(\pi_0:(k:K)\times X\,k\to K)
$$

Reornament

B.3 Universe of function types

$$
\begin{array}{l}
\textbf{data}\ \mathsf{Type} : \mathrm{SET}_1\ \textbf{where} \\
\quad \mathsf{Type} \ni \mu\{(D : \mathsf{func}\, K) \cdot (k : K)\} \rightarrow (T : \mathsf{Type}) \\
\qquad\quad \mid \mu\{(D : \mathsf{func}\, K) \cdot (k : K)\} \times (T : \mathsf{Type}) \\
\qquad\quad \mid 1 \\
\\
[\![(T : \mathsf{Type})]\!]_{\mathsf{Type}} \quad : \quad \mathrm{SET} \\
[\![\mu\{D \cdot k\} \rightarrow T]\!]_{\mathsf{Type}} \mapsto \mu\, D\, k \rightarrow [\![T]\!]_{\mathsf{Type}} \\
[\![\mu\{D \cdot k\} \times T]\!]_{\mathsf{Type}} \mapsto \mu\, D\, k \times [\![T]\!]_{\mathsf{Type}} \\
[\![1]\!]_{\mathsf{Type}} \qquad\qquad\ \mapsto \mathbb{1}
\end{array}
$$

Code and interpretation

B.4 Universe of functional ornaments

$$
\begin{array}{l}
\textbf{data}\ \mathsf{FunOrn}\,(T : \mathsf{Type}) : \mathrm{SET}_1\ \textbf{where} \\
\quad \mathsf{FunOrn}\,(\mu\{D \cdot k\} \rightarrow T) \ni \forall u : I \rightarrow K.\, \mu^+\{(o : \mathsf{orn}\, D\, u) \cdot (i : u^{-1}\, k)\} \rightarrow (T^+ : \mathsf{FunOrn}\, T) \\
\quad \mathsf{FunOrn}\,(\mu\{D \cdot k\} \times T) \ni \forall u : I \rightarrow K.\, \mu^+\{(o : \mathsf{orn}\, D\, u) \cdot (i : u^{-1}\, k)\} \times (T^+ : \mathsf{FunOrn}\, T) \\
\quad \mathsf{FunOrn} \qquad 1 \qquad\quad \ni 1 \\
\\
[\![(T^+ : \mathsf{FunOrn}\, T)]\!]_{\mathsf{FunOrn}} \quad : \quad \mathrm{SET} \\
[\![\mu^+\{o \cdot \mathsf{inv}\, i\} \rightarrow T^+]\!]_{\mathsf{FunOrn}} \mapsto \mu\, [\![o]\!]_{\mathsf{orn}}\, i \rightarrow [\![T^+]\!]_{\mathsf{FunOrn}} \\
[\![\mu^+\{o \cdot \mathsf{inv}\, i\} \times T^+]\!]_{\mathsf{FunOrn}} \mapsto \mu\, [\![o]\!]_{\mathsf{orn}}\, i \times [\![T^+]\!]_{\mathsf{FunOrn}} \\
[\![1]\!]_{\mathsf{FunOrn}} \qquad\qquad\qquad\ \mapsto \mathbb{1}
\end{array}
$$

Code and interpretation

$$
\begin{array}{l}
\mathsf{Coherence}\,(T^+ : \mathsf{FunOrn}\, T)\,(f : [\![T]\!]_{\mathsf{Type}})\,(f^+ : [\![T^+]\!]_{\mathsf{FunOrn}}) : \mathrm{SET} \\
\mathsf{Coherence}\,(\mu^+\{o \cdot \mathsf{inv}\, i\} \rightarrow T^+) \quad f \qquad f^+ \qquad \mapsto \\
\qquad\qquad (x^+ : \mu\, [\![o]\!]_{\mathsf{orn}}\, i) \rightarrow \mathsf{Coherence}\, T^+\,(f\,(\mathsf{forget}\, o\, x^+))\,(f^+ x^+) \\
\mathsf{Coherence}\,(\mu^+\{o \cdot \mathsf{inv}\, i\} \times T^+)\,(x, xs)\,(x^+, xs^+) \mapsto \\
\qquad\qquad \mathsf{forget}\, o\, x^+ = x \times \mathsf{Coherence}\, T^+\, xs\, xs^+ \\
\mathsf{Coherence} \qquad 1 \qquad\quad * \qquad\quad * \qquad \mapsto \mathbb{1}
\end{array}
$$

Coherence

B.5 Patches

```
Patch    (T : Type)      (T⁺ : FunOrn T)   (f : ⟦T⟧Type)  :  SET
Patch (μ{D · u i}⟶ T) (μ⁺{o · inv i}⟶ T⁺)      f       ↦
      (x : μ D (u i)) → μ ⟦⌈o⌉⟧orn (i, x) → Patch T T⁺ (f x)
Patch (μ{D · u i}× T) (μ⁺{o · inv i}× T⁺)     (x, xs)   ↦
      μ ⟦⌈o⌉⟧orn (i, x) × Patch T T⁺ xs
Patch        1                 1                 *      ↦ 𝟙
```

Patch

```
patch (T⁺ : FunOrn T) (f : ⟦T⟧Type) (f⁺⁺ : Patch T T⁺ f) : ⟦T⁺⟧FunOrn
patch (μ⁺{o · inv i}⟶ T⁺)   f       f⁺⁺      ↦
      λx⁺. patch T⁺ (f (forget o x⁺))
                    (f⁺⁺ (forget o x⁺) (make ⌈o⌉ x⁺))
patch (μ⁺{o · inv i}× T⁺) (x, xs) (x⁺⁺, xs⁺⁺) ↦
      (forget ⌈o⌉ x⁺⁺, patch T⁺ xs xs⁺⁺)
patch        1              *         *      ↦ *

coherence (T⁺ : FunOrn T) (f : ⟦T⟧Type) (f⁺⁺ : Patch T T⁺ f) : Coherence T⁺ f (patch T⁺ f f⁺⁺)
coherence (μ⁺{o · inv i}⟶ T⁺)   f       f⁺⁺      ↦
      λx⁺. coherence T⁺ (f (forget o x⁺))
                        (f⁺⁺ (forget o x⁺) (make ⌈o⌉ x⁺))
coherence (μ⁺{o · inv i}× T⁺) (x, xs) (x⁺, xs⁺⁺) ↦
      (coherentOrn x⁺, coherence T⁺ xs xs⁺⁺)
coherence        1              *         *      ↦ *
```

Patching and its coherence proof

B.6 Transporting recursion patterns

$$
\begin{array}{l}
\mathsf{lift}_{\mathsf{Alg}}\,(\alpha : \forall k : K.\, [\![D]\!]\,(\lambda - .\, [\![T]\!]_{\mathsf{Type}})\,k \to [\![T]\!]_{\mathsf{Type}})\,(o : \mathsf{orn}\,D\,u)\,(T^+ : \mathsf{FunOrn}\,T) : \textsc{Set} \\
\mathsf{lift}_{\mathsf{Alg}}\,\alpha\,o\,T^+ \mapsto \forall\,(i,t) : (i : I) \times \mu\,D\,(u\,i)). \\
\qquad\qquad [\![\lceil o \rceil]\!]_{\mathsf{orn}}\,(\lambda(i,t).\,\mathsf{Patch}\,T\,T^+\,((\!|\alpha|\!)\,t))\,(i,t) \to \mathsf{Patch}\,T\,T^+\,((\!|\alpha|\!)\,t) \\
\\
\mathsf{lift\text{-}fold}\,(\alpha : \forall k : K.\, [\![D]\!]\,(\lambda - .\, [\![T]\!]_{\mathsf{Type}})\,k \to [\![T]\!]_{\mathsf{Type}}) \\
\qquad (\alpha^{++} : \mathsf{lift}_{\mathsf{Alg}}\,\alpha\,o\,T^+) : \mathsf{Patch}\,(\mu\{D \cdot u\,i\} \rightarrowtail T)\,(\mu^+\{o \cdot \mathsf{inv}\,i\} \rightarrowtail T^+)\,(\!|\alpha|\!) \\
\mathsf{lift\text{-}fold}\,\alpha\,\alpha^{++} \mapsto \lambda x.\,\lambda x^{++}.\,(\!|\alpha^{++}|\!)\,x^{++}
\end{array}
$$

Coherent algebra and its lifting

$$
\begin{array}{l}
\mathsf{lift}_{\mathsf{IH}}\,(\alpha : (k : K)(xs : [\![D]\!]\,(\mu\,D)\,k) \to \square_D\,(\lambda - .\, [\![T]\!]_{\mathsf{Type}})\,xs \to [\![T]\!]_{\mathsf{Type}}) \\
\qquad (o : \mathsf{orn}\,D\,u)(T^+ : \mathsf{FunOrn}\,T) : \textsc{Set} \\
\mathsf{lift}_{\mathsf{IH}}\,\alpha\,o\,T^+ \mapsto ((i,t) : (i : I) \times \mu\,D\,(u\,i)) \to (xs : [\![\lceil o \rceil]\!]_{\mathsf{orn}}\,(\mu\,[\![\lceil o \rceil]\!]_{\mathsf{orn}})\,(i,t)) \to \\
\qquad\qquad \square_{[\![\lceil o \rceil]\!]_{\mathsf{orn}}}\,(\lambda(i,t).\,\mathsf{Patch}\,T\,T^+\,(\mathsf{induction}\,\alpha\,t))\,xs \to \\
\qquad\qquad \mathsf{Patch}\,T\,T^+\,(\mathsf{induction}\,\alpha\,t) \\
\\
\mathsf{lift\text{-}ind}\,(\alpha : (k : K)(xs : [\![D]\!]\,(\mu\,D)\,k) \to \square_D\,(\lambda - .\, [\![T]\!]_{\mathsf{Type}})\,xs \to [\![T]\!]_{\mathsf{Type}}) \\
\qquad (\alpha^{++} : \mathsf{lift}_{\mathsf{IH}}\,\alpha\,o\,T^+) : \mathsf{Patch}\,(\mu\{D \cdot u\,i\} \rightarrowtail T)\,(\mu^+\{o \cdot \mathsf{inv}\,i\} \rightarrowtail T^+)\,(\mathsf{induction}\,\alpha) \\
\mathsf{lift\text{-}ind}\,\alpha\,\alpha^{++} \mapsto \lambda x.\,\lambda x^{++}.\,\mathsf{induction}\,\alpha^{++}\,x^{++}
\end{array}
$$

Coherent inductive step and its lifting

$$
\begin{array}{ll}
\mathsf{lift\text{-}constructor}\,(e : \mathsf{Extension}\,(o\,i)\,xs) & \text{– coherent extension} \\
\qquad (a : \mathsf{Structure}\,(o\,i)\,xs\,e) & \text{– recursive arguments} \\
\qquad (t^{++} : \mathsf{Patch}\,T\,T^+\,t) & \\
\qquad : \mathsf{Patch}\,(\mu\{D \cdot u\,i\} \times T)\,(\mu^+\{o \cdot \mathsf{inv}\,i\} \times T^+)\,(\mathsf{in}\,xs, t) & \\
\mathsf{lift\text{-}constructor}\,e\,a\,t^{++} \mapsto (\mathsf{in}\,(\mathsf{mkReorn}\,(o\,i)\,xs\,e\,a), t^{++}) &
\end{array}
$$

Constructor lifting