Transporting Functions across Ornaments

Pierre-Evariste Dagand Conor McBride

Mathematically Structured Programming group
University of Strathclyde
{dagand,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 effort of code reuse among similarly structured data. In this paper, we exorcise our datatypes by adapting 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.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming

Keywords Dependent types, Datatype, Ornament

1. Introduction

Imagine designing a library for a 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 optimised set of operations. If, tempted by a snake, 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 the ML days, 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, the programmer can enforce invariants on top of the data-structures. In such a system, programmers strive to express the correctness of programs in their types: a well typed program is correct *by construction*, the proof of correctness being reduced to type-checking.

A simple yet powerful recipe to obtain these richer datatypes is to *index* the data-structure. These datatypes have originally been studied in the context of 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 [Xi et al. 2003], a subset of inductive families for which type inference is decidable. Refinement types [Freeman and Pfenning 1991; Swamy et al. 2011] are another technique to equip data-structures with rich invariants. Atkey et al. [2011] have shown how refinement types relate to inductive families, and Bernardy and Lasson [2011] establish a connection with realisability.

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 [2012] developed *ornaments*, describing how one datatype can be enriched into others *with the same structure*. Such structure-preserving transformations take two forms: one can *extend* the initial type with more information – such as obtaining Maybe_A from Bool or List_A from Nat:

$$\begin{array}{lcl}
\textbf{data}\ \mathsf{Bool} : \textsc{Set}\ \textbf{where} & & \textbf{data}\ \mathsf{Nat} : \textsc{Set}\ \textbf{where}\\
\quad \mathsf{Bool}\ \ni\ \mathsf{true} & & \quad \mathsf{Nat}\ \ni\ 0\\
\qquad\qquad \mid\ \mathsf{false} & & \qquad\qquad \mid\ \mathsf{suc}\,(n : \mathsf{Nat})\\
\\
\quad \Downarrow \mathsf{Maybe\text{-}Orn} & & \quad \Downarrow \mathsf{List\text{-}Orn}\\
\textbf{data}\ \mathsf{Maybe}\ [A : \textsc{Set}] : \textsc{Set}\ \textbf{where} & & \textbf{data}\ \mathsf{List}\ [A : \textsc{Set}] : \textsc{Set}\ \textbf{where}\\
\quad \mathsf{Maybe}_A\ \ni\ \mathsf{just}\,(a : A) & & \quad \mathsf{List}_A\ \ni\ \mathsf{nil}\\
\qquad\qquad \mid\ \mathsf{nothing} & & \qquad\qquad \mid\ \mathsf{cons}\,(a : A)(as : \mathsf{List}_A)
\end{array}$$

Or one can *refine* the indexing of the initial type by a finer discipline – e.g., obtaining Fin by indexing Nat with a bound n:

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

One can also do both at the same time – such as extending Nat into a List_A while refining the index to match the length of the list:

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

ICFP'12, September 9–15, 2012, Copenhagen, Denmark.

Note that we declare datatype parameters $[A : \mathrm{SET}]$ in brackets and datatype indices $(n : \mathsf{Nat})$ in parentheses. We make equational constraints on the latter only when needed, and explicitly.

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. Whilst we now 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 List_A as an ornament of Nat, it seems natural to transport some structure-preserving function of Nat to List_A, such as moving from addition of natural numbers to concatenation of lists:

$$\begin{array}{lclcl}
(m : \mathsf{Nat}) & + & (n : \mathsf{Nat}) & : & \mathsf{Nat} \\
0 & + & n & \mapsto & n \\
(\mathsf{suc}\ m) & + & n & \mapsto & \mathsf{suc}\ (m + n)
\end{array}$$

$$\Downarrow$$

$$\begin{array}{lclcl}
(xs : \mathsf{List}_A) & +\!\!+ & (ys : \mathsf{List}_A) & : & \mathsf{List}_A \\
\mathsf{nil} & +\!\!+ & ys & \mapsto & ys \\
(\mathsf{cons}\ a\ xs) & +\!\!+ & ys & \mapsto & \mathsf{cons}\ a\ (xs +\!\!+ ys)
\end{array}$$

Or moving from subtraction of natural numbers to dropping the prefix of a list:

$$\begin{array}{lclcl}
(m : \mathsf{Nat}) & - & (n : \mathsf{Nat}) & : & \mathsf{Nat} \\
0 & - & n & \mapsto & 0 \\
m & - & 0 & \mapsto & m \\
(\mathsf{suc}\ m) & - & (\mathsf{suc}\ n) & \mapsto & m - n
\end{array}$$

$$\Downarrow$$

$$\begin{array}{lllcl}
\mathsf{drop} & (xs : \mathsf{List}_A) & (n : \mathsf{Nat}) & : & \mathsf{List}_A \\
\mathsf{drop} & \mathsf{nil} & n & \mapsto & \mathsf{nil} \\
\mathsf{drop} & xs & 0 & \mapsto & xs \\
\mathsf{drop} & (\mathsf{cons}\ a\ xs) & (\mathsf{suc}\ n) & \mapsto & \mathsf{drop}\ xs\ n
\end{array}$$

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 adapt ornaments to our universe of datatypes [Chapman et al. 2010] in Section 3. This presentation benefits greatly from our ability to inspect indices when defining datatypes. This allows us to consider ornaments which *delete* index-determined information, yielding a key simplification in the construction of an algebraic ornament from an ornamental algebra ;
- We describe how functions can be transported through functional ornaments: 'deletion' allows us a contrasting approach to Ko and Gibbons [2011], internalising proof obligations. First, we manually work through an example in Section 2. Then, we formalise the concept of functional ornament by a universe construction in Section 4. Based on this universe, we establish the connection between a base function (such as _ + _ and _ − _) 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 constructions ;
- In Section 5, we provide further support to drive the computer into lifting functions semi-automatically. As we can see from our examples above, the lifted functions often follow the same recursion pattern and return similar constructors: with a few generic constructions, we shall remove further clutter and code duplication from our libraries.

$$\begin{array}{lclcl}
(m : \mathsf{Nat}) & < & (n : \mathsf{Nat}) & : & \mathsf{Bool} \\
m & < & 0 & \mapsto & \mathsf{false} \\
0 & < & \mathsf{suc}\ n & \mapsto & \mathsf{true} \\
\mathsf{suc}\ m & < & \mathsf{suc}\ n & \mapsto & m < n
\end{array}$$

$$\Downarrow ?$$

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

Figure 1. Implementation of _ < _ and lookup

This paper is an exercise in constructive mathematics: upon identifying an isomorphism, we shall look at it with our constructive glasses and obtain an effective procedure that lets us cross the isomorphism. In this paper, we put a strong emphasis on the programming aspect: we shall only hint at the isomorphisms through concrete examples and let the reader consult the companion technical report for the actual mathematical proofs.

We shall write our code in a syntax inspired by the Epigram [McBride and McKinna 2004] programming language. In particular, we 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. We shall also make ample use of mathematical notations and symbols in the programming language itself (in particular, mixfix operators), hence appealing to our reader's eye for mathematics, rather than to the intricate details of a particular formal syntax. Like ML, unbound variables in type definitions are universally quantified, further abating syntactic noise. The syntax of datatype definitions draws upon the ML tradition as well: its novelty will be presented by way of examples in Section 3. All the constructions presented in this paper have been modelled in Agda, using only standard inductive definitions and two levels of universe. The formalisation and technical report are available on Dagand's website.

2. From _ < _ to lookup, manually

There is an astonishing resemblance between the comparison function _ < _ on natural numbers and the list lookup function (Fig. 1). The similarity is not merely at the level of types but also in their implementation: their definitions follow the same pattern of recursion (first, case analysis on the second element; then induction on the first element) and they both return a failure value (false and nothing respectively) in the first case analysis and a success value (true and just respectively) 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:

$$\begin{tikzcd}
\_<\_ & : & \mathsf{Nat} \arrow[r, phantom, "\to"] \arrow[d, Rightarrow, "\mathsf{idO}_{\mathsf{Nat}}"'] & \mathsf{Nat} \arrow[r, phantom, "\to"] \arrow[d, Rightarrow, "\mathsf{List\text{-}Orn}"'] & \mathsf{Bool} \arrow[d, Rightarrow, "\mathsf{Maybe\text{-}Orn}"'] \\
\mathsf{lookup} & : & \mathsf{Nat} \arrow[r, phantom, "\to"] & \mathsf{List}_A \arrow[r, phantom, "\to"] & \mathsf{Maybe}_A
\end{tikzcd}$$

Note that the first argument is ornamented to itself, or put differently, it has been ornamented by the identity ornament.

Each of these ornaments come with a forgetful map, computed from the ornamental algebra:

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

Using these forgetful map, the relation, at the computational level, between $\_<\_$ and lookup is uniquely established by their ornamentation. This relation is captured by the *coherence* property: $\forall n\!:\!\mathsf{Nat}.\forall xs\!:\!\mathsf{List}_A.\ \mathsf{isJust}\,(\mathsf{lookup}\ n\ xs) \equiv n < \mathsf{length}\ xs$.

Let us settle the vocabulary at this stage. 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* (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. 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 different coherence properties.

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: one would implement lookup and prove the coherence as a theorem. This works rather well in a system like Coq [The Coq Development Team] as it offers a powerful theorem proving environment. It does not work so well in a system like Agda [Norell 2007] that does not offer tactics to its users, forcing them to write explicit proof terms. It would not work at all in Haskell with GADTs, which has no notion of proof.

However, we are not satisfied by this laborious approach: if we have dependent types, why should we use them only for *proofs*, as an afterthought? We should rather write a lookup function *correct by construction*: by implementing a more precisely indexed version of lookup, the user can drive the index-level computations to unfold, hence making the type-checker verify the necessary invariants. We believe that this is how it should be: computers should replace proofs by computation; humans should drive computers. The other way around – where humans are coerced into computing for computers – may seem surreal, yet it corresponds to the current situation in most proof systems.

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

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

Where IMaybe_A is Maybe_A indexed by its truth as computed by isJust. It is defined as follows[1]:

$$\begin{array}{l}
\mathbf{data}\ \mathsf{IMaybe}\ [A\!:\!\mathrm{SET}](b\!:\!\mathsf{Bool})\!:\!\mathrm{SET}\ \mathbf{where} \\
\quad \mathsf{IMaybe}_A\ \mathsf{true} \ni \mathsf{just}\,(a\!:\!A) \\
\quad \mathsf{IMaybe}_A\ \mathsf{false} \ni \mathsf{nothing}
\end{array}$$

This comes with the following forgetful map:

$$\begin{array}{llll}
\mathsf{forgetIMaybe} & (mba\!:\!\mathsf{IMaybe}_A\ b) & : & (ma\!:\!\mathsf{Maybe}_A) \times \mathsf{isJust}\ ma \equiv b \\
\mathsf{forgetIMaybe} & (\mathsf{just}\ a) & \mapsto & (\mathsf{just}\ a, \mathsf{refl}) \\
\mathsf{forgetIMaybe} & \mathsf{nothing} & \mapsto & (\mathsf{nothing}, \mathsf{refl})
\end{array}$$

The rationale behind ilookup is to *index* the types of lookup by their unornamented version, i.e. the arguments and result of $\_<\_$. Hence, we can make sure that the result computed by ilookup respects the output of $\_<\_$ on the unornamented indices: the result is 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. Expounding further our vocabulary, we call *coherent liftings* these finely indexed functions that are correct by construction.

Ko and Gibbons [2011] use ornaments to specify the coherence requirements for functional liftings, but we work the other way around, using ornaments to internalise coherence requirements. From ilookup, we can extract both lookup and its proof of correctness *without having written any proof term ourselves*:

$$\begin{array}{llll}
\mathsf{lookup} & (m\!:\!\mathsf{Nat}) & (xs\!:\!\mathsf{List}_A) & : \ \mathsf{Maybe}_A \\
\mathsf{lookup} & m & xs & \mapsto \\
\multicolumn{4}{l}{\quad \pi_0(\mathsf{forgetIMaybe}\,(\mathsf{ilookup}\ m\ (\mathsf{makeVec}\ xs)))}
\end{array}$$

$$\begin{array}{llll}
\mathsf{cohLookup} & (n\!:\!\mathsf{Nat}) & (xs\!:\!\mathsf{List}_A) & : \\
\multicolumn{4}{l}{\quad \mathsf{isJust}\,(\mathsf{lookup}\ n\ xs) \equiv n < \mathsf{length}\ xs} \\
\mathsf{cohLookup} & m & xs & \mapsto \\
\multicolumn{4}{l}{\quad \pi_1(\mathsf{forgetIMaybe}\,(\mathsf{ilookup}\ m\ (\mathsf{makeVec}\ xs)))}
\end{array}$$

where $\mathsf{makeVec} : (xs : \mathsf{List}_A) \to \mathsf{Vec}_A\ (\mathsf{length}\ xs)$ simply turns a list into a vector of the corresponding length.

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 List_A and Bool to Maybe_A in order to describe the desired lifting, here $\mathsf{lookup} : \mathsf{Nat} \to \mathsf{List}_A \to \mathsf{Maybe}_A$ satisfying $\forall n\!:\!\mathsf{Nat}.\forall xs\!:\!\mathsf{List}_A.\ \mathsf{isJust}\,(\mathsf{lookup}\ n\ xs) \equiv n < \mathsf{length}\ xs$
- 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 proving any theorem

This manual unfolding of the lifting is instructive: it involves a lot of constructions on datatypes (here, the datatypes List_A and 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 the next Section, we shall build the machinery to describe these constructions and obtain them *within* the type theory itself.

3. A universe of datatypes and their ornaments

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 datatypes 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 datatype definitions, as they belong to the meta-theory.

3.1 A closed theory of datatypes

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

Note that our results are not restricted to this setting where datatype definitions are 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 Haskell with GADTs. Working with an internalised presentation, we can simply avoid these two levels of logic and work in the logic provided by the type theory itself.

[1] Note that we have overloaded the constructors of Maybe and IMaybe: for a bi-directional type-checker, there is no ambiguity as constructors are checked against their type.

$$\begin{array}{l}
\textbf{data}\ \mathsf{IDesc}\ [I:\textsc{Set}]:\textsc{Set}_1\ \textbf{where} \\
\quad \mathsf{IDesc}\ I\ \ni\ \text{'}\mathsf{var}\,(i:I) \\
\qquad\qquad \mid\ \text{'}1 \\
\qquad\qquad \mid\ \text{'}\Pi\,(S:\textsc{Set})\ (T:S\to\mathsf{IDesc}\,I) \\
\qquad\qquad \mid\ \text{'}\Sigma\,(S:\textsc{Set})\ (T:S\to\mathsf{IDesc}\,I)
\end{array}$$

$$\begin{array}{lll}
[\![(D:\mathsf{IDesc}\,I)]\!]\ (X:I\to\textsc{Set}) & : & \textsc{Set} \\
[\![\text{'}\mathsf{var}\,i]\!]\ X & \mapsto & X\ i \\
[\![\text{'}1]\!]\ X & \mapsto & \mathbb{1} \\
[\![\text{'}\Pi\,S\ T]\!]\ X & \mapsto & (s:S)\to[\![T\ s]\!]\ X \\
[\![\text{'}\Sigma\,S\ T]\!]\ X & \mapsto & (s:S)\times[\![T\ s]\!]\ X
\end{array}$$

Figure 2. Universe of inductive families

For the sake of completeness, let us recall a few definitions and results from our previous work. As in previous work, our requirements on the type theory are minimal: we will need Σ-, Π-types, and at least two universes. For convenience, we require a type of finite sets, which lets us build collections of labels[2]. We also need a pre-existing notion of propositional equality, upon which we make no assumption. We internalise the inductive families by a universe construction (Fig. 2): an indexed datatype is described by a function from its index to codes. The codes are then interpreted to build the fix-point:

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

For readability purposes, we use an informal notation to declare datatypes. This notation is strongly inspired by Agda's datatype declarations. Note that these definitions can always be turned into IDesc codes: when defining a datatype T, we will denote T-Desc the code it elaborates to. Similarly, we denote T-elim and T-case the induction principle and case analysis operators associated with T. For instance, Nat-case corresponds to case analysis over natural numbers (either 0 or suc) while Nat-elim corresponds to standard induction on natural numbers. These operations can be implemented by generic programming, along the lines of McBride et al. [2004]. Formalising the elaboration of datatypes definitions down to code is beyond the scope of this paper. However, it is simple enough to be understood with a few examples. Three key ideas are at play.

First, non-indexed datatypes definitions follow the ML tradition: we name the datatype and then comes a choice of constructors. For example, List and Brouwer ordinals would be written and elaborated as follows:

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

$$\wr$$

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

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

$$\wr$$

$$\begin{array}{l}
\mathsf{Ord\text{-}Desc}\ (x:\mathbb{1})\ :\ \mathsf{IDesc}\,\mathbb{1} \\
\mathsf{Ord\text{-}Desc}\ *\ \mapsto\ \text{'}\Sigma\left\{\begin{array}{l}\text{'}0\\ \text{'}\mathsf{suc}\\ \text{'}\mathsf{lim}\end{array}\right\}\left\{\begin{array}{ll}\text{'}0 & \mapsto \text{'}1\\ \text{'}\mathsf{suc} & \mapsto \text{'}\mathsf{var}\,*\\ \text{'}\mathsf{lim} & \mapsto \text{'}\Pi\,\mathsf{Nat}\ \lambda\_.\ \text{'}\mathsf{var}\,*\end{array}\right\}
\end{array}$$

Secondly, indexed datatypes can be defined following the Agda convention: indices are *constrained* to some particular value. For example, Vec could be defined by constraining the index to be 0 in the nil case and suc n' for some n' : Nat in the cons case:

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

$$\wr$$

$$\begin{array}{l}
\mathsf{Vec\text{-}Desc}\ (A:\textsc{Set})\ (n:\mathsf{Nat})\ :\ \mathsf{IDesc}\,\mathsf{Nat} \\
\mathsf{Vec\text{-}Desc}\ A\ n\ \mapsto \\
\quad \text{'}\Sigma\left\{\begin{array}{l}\text{'}\mathsf{nil}\\ \text{'}\mathsf{cons}\\ \end{array}\right\}\left\{\begin{array}{ll}\text{'}\mathsf{vnil} & \mapsto \text{'}\Sigma\,(n\equiv 0)\ \lambda\_.\ \text{'}1\\ \text{'}\mathsf{vcons} & \mapsto \text{'}\Sigma\,\mathsf{Nat}\ \lambda n'.\ \text{'}\Sigma\,(n\equiv\mathsf{suc}\,n')\lambda\_.\\ & \qquad\quad \text{'}\Sigma\ A\ \lambda\_.\ \text{'}\mathsf{var}\,n'\end{array}\right\}
\end{array}$$

The elaboration naturally 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 constructors have been chosen, we restrict n to its valid value(s): 0 in the first case and suc n' for some n' in the second case. Hence the placement of the equality constraints in the above definition: 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.

Thirdly, we can compute over indices: here, we make use of the crucial property that a datatype definition is a *function* from index to IDesc codes. Hence, our notation should reflect this ability to define datatypes as functions on their index. For instance, inspired by Brady et al. [2004], an alternative presentation of vector would match on the index to determine the constructor to be presented, hence removing the need for constraints:

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

$$\wr$$

$$\begin{array}{l}
\mathsf{Vec\text{-}Desc}\ (A:\textsc{Set})\ (n:\mathsf{Nat})\ :\ \mathsf{IDesc}\,\mathsf{Nat} \\
\mathsf{Vec\text{-}Desc}\ A\ n\ \mapsto\ \mathsf{Nat\text{-}case}\ n\ (\lambda\_.\ \mathsf{IDesc}\,\mathsf{Nat}) \\
\qquad\qquad\qquad\qquad\qquad\ \text{'}1 \\
\qquad\qquad\qquad\qquad\qquad\ (\lambda n.\ \text{'}\Sigma\ A\ \lambda\_.\ \text{'}\mathsf{var}\,n)
\end{array}$$

In order to be fully explicit about computations, we use here the Epigram [McBride and McKinna 2004] *by* ($\Leftarrow$) programming gadget, which let us appeal to any elimination principle with a syntax close to pattern-matching. However, standard pattern-matching constructions [Coquand 1992; Norell 2007] would work just as well. Again, we shall write pattern-matching definitions when the recursion pattern is unremarkable.

Our syntax departs radically from the one adopted by Coq, Agda, and GADTs in Haskell. It is crucial to understand that this is but reflecting the actual semantics of inductive families: we can *compute* over indices, not merely constrain them to be what we would like. 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.

3.2 Ornaments

Originally, McBride [2012] presented the notion of ornament for a universe where the indices a constructor targets could be enforced *only* by equality constraints. As a consequence, in that simpler setting, computing types from indices was impossible. We shall now adapt the original definition to our setting.

Just as the original definition, an ornament is defined upon a base datatype – specified by a function $D : I \mapsto \mathsf{IDesc}\,I$ – and indices are refined up to a reindexing function $re : J \to I$. The

[2] We denote finite sets of tagged elements by $\{\text{'x}, \text{'y}, \text{'z}, \ldots\}$. Their elimination principle consists of an exhaustive case enumeration and is denoted by $\{\text{'x} \mapsto vx, \text{'y} \mapsto vy, \text{'z} \mapsto vz, \ldots\}$. If the tags are vertically aligned, we shall skip the separating comma.

difference in our setting is that, just as the code of datatypes can be computed from the indices, we want the ornament to be computable from its J-index. Hence, an ornament is a function from $j : J$ to ornament codes describing the ornamentation of $D\ (re\ j)$:

$$\begin{array}{l}\mathsf{orn}(re_I : J \to I)(re_O : P \to O)(D : O \to \mathsf{IDesc}\, I) \;:\; \mathrm{S}\small{\mathrm{ET}}_1\\ \mathsf{orn}\; re_I\; re_O\; D \;\mapsto\; (p : P) \to \mathsf{Orn}\; re_I\; (D\; (re_O\; p))\end{array}$$

As for the ornament codes themselves, they are similar to the original definition: we shall be able to *copy* the base datatype, *extend* it by inserting sets, or *refine* the indexing subject to the relation imposed by re. However, we also have the J-index in our context: following Brady's insight that *inductive families need not store their indices* [Brady et al. 2004], we could as well *delete* parts of a datatype definition as long as we can recover this information from the index. Hence, we obtain the following code[3]:

$$\begin{array}{llll}
\multicolumn{4}{l}{\mathbf{data}\ \mathsf{Orn}\ [re : J \to I](D : \mathsf{IDesc}\, I) : \mathrm{SET}_1\ \mathbf{where}}\\
\multicolumn{4}{l}{\quad \textit{– Extend with } S\textit{:}}\\
\quad \mathsf{Orn}\ re & D & \ni & \mathsf{insert}\,(S : \mathrm{SET})(D^{+} : S \to \mathsf{Orn}\ re\ D)\\
\multicolumn{4}{l}{\quad \textit{– Refine index:}}\\
\quad \mathsf{Orn}\ re & (\text{'}\mathsf{var}\ i) & \ni & \text{'}\mathsf{var}\,(j : re^{\,-1}\ i)\\
\multicolumn{4}{l}{\quad \textit{– Copy the original:}}\\
\quad \mathsf{Orn}\ re & \text{'}1 & \ni & \text{'}1\\
\quad \mathsf{Orn}\ re & (\text{'}\Pi\, S\ T) & \ni & \text{'}\Pi\,(T^{+} : (s : S) \to \mathsf{Orn}\ re\ (T\ s))\\
\quad \mathsf{Orn}\ re & (\text{'}\Sigma\, S\ T) & \ni & \text{'}\Sigma\,(T^{+} : (s : S) \to \mathsf{Orn}\ re\ (T\ s))\\
\multicolumn{4}{l}{\quad \textit{– Delete } S\textit{:}}\\
 & & | & \mathsf{delete}\,(\mathsf{replace} : S)(T^{+} : \mathsf{Orn}\ re\ (T\ \mathsf{replace}))
\end{array}$$

Note that the recursive structure of the original data-type – as specified by $\text{'}\Pi$ – is preserved by the ornament: we have thus ensured, by construction, that the source datatype and its ornament have the same recursive structure. Being able to insert or delete $\text{'}\Pi$-quantifiers would defeat our purpose by making ambiguous the connection between the source datatype and its ornamented form.

Given an ornament, we can interpret it as the datatype it describes. The implementation consists in traversing the ornament code, introducing a $\text{'}\Sigma$ when inserting new data and computing the ornament at the replaced value when deleting some redundant data:

$$\begin{array}{l}
\begin{array}{llll}
[\![(o : \mathsf{orn}\ re_I\ re_O\ D)]\!]_{\mathsf{orn}}\ (p : P) & : & \mathsf{IDesc}\ J & \\
[\![o]\!]_{\mathsf{orn}}\qquad\qquad\qquad\quad p & \mapsto & \mathsf{intOrn}\,(D\ (re_O\ p))\ (o\ p) & \mathbf{where}
\end{array}\\
\quad \mathsf{intOrn}(D : \mathsf{IDesc}\, I)(O : \mathsf{Orn}\ re\ D) : \mathsf{IDesc}\ J\\
\quad \mathsf{intOrn}\quad D \quad (\mathsf{insert}\ S\ D^{+}) \;\mapsto\; \text{'}\Sigma\, S\ \lambda s.\ \mathsf{intOrn}\ D\ (D^{+}\ s)\\
\quad \mathsf{intOrn}\quad (\text{'}\mathsf{var}\,(re\ j)) \quad (\text{'}\mathsf{var}\,(\mathsf{inv}\ j)) \;\mapsto\; \text{'}\mathsf{var}\ j\\
\quad \mathsf{intOrn}\quad \text{'}1 \quad \text{'}1 \;\mapsto\; \text{'}1\\
\quad \mathsf{intOrn}\quad (\text{'}\Pi\, S\ T) \quad (\text{'}\Pi\ T^{+}) \;\mapsto\; \text{'}\Pi\, S\ \lambda s.\ \mathsf{intOrn}\ (T\ s)\ (T^{+}\ s)\\
\quad \mathsf{intOrn}\quad (\text{'}\Sigma\, S\ T) \quad (\text{'}\Sigma\ T^{+}) \;\mapsto\; \text{'}\Sigma\, S\ \lambda s.\ \mathsf{intOrn}\ (T\ s)\ (T^{+}\ s)\\
\quad \mathsf{intOrn}\quad (\text{'}\Sigma\, S\ T) \quad (\mathsf{delete}\ \mathsf{replace}\ T^{+}) \;\mapsto\\
\qquad\quad \mathsf{intOrn}\ (T\ \mathsf{replace})\ (T^{+}\ \mathsf{replace})
\end{array}$$

Note that in the delete case, no $\text{'}\Sigma$ code is generated: the set S has been deleted from the original datatype. The witness of this existential is instead provided by replace.

Once again, we adopt an informal notation to describe ornaments conveniently. The idea is to simply mirror our **data** definition, adding **from** which datatype the ornament is defined. When specifying a constructor, we can then extend it with a new element using $[s : S]$ or delete an element originally named s by giving its value with $[s \triangleq \mathsf{value}]$. Some typical examples of extension are presented in Figure 3.

While the definition Vec in Figure 3 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:

$$\begin{array}{llll}
\multicolumn{4}{l}{\mathbf{data}\ \mathsf{Vec}\ [A : \mathrm{SET}](n : \mathsf{Nat})\ \mathbf{from}\ \mathsf{List}_A\ \mathbf{where}}\\
\quad \mathsf{Vec}_A & 0 & \ni & \mathsf{nil}\\
\quad \mathsf{Vec}_A & (\mathsf{suc}\ n') & \ni & \mathsf{cons}\,(a : A)(vs : \mathsf{Vec}_A\, n')
\end{array}$$

$$\begin{array}{llll}
\multicolumn{4}{l}{\mathbf{data}\ \mathsf{List}\ [A : \mathrm{SET}]\ \mathbf{from}\ \mathsf{Nat}\ \mathbf{where}}\\
\quad \mathsf{List}_A & & \ni & \mathsf{nil}\\
 & & | & \mathsf{cons}\,[a : A](as : \mathsf{List}_A)\\
 & & & \\
\multicolumn{4}{l}{\mathbf{data}\ \mathsf{Vec}\ [A : \mathrm{SET}](n : \mathsf{Nat})\ \mathbf{from}\ \mathsf{List}_A\ \mathbf{where}}\\
\quad \mathsf{Vec}_A & n & \ni & \mathsf{nil}\,[q : n \equiv 0]\\
 & & | & \mathsf{cons}\,[n' : \mathsf{Nat}][q : n \equiv \mathsf{suc}\ n'](a : A)(vs : \mathsf{Vec}_A\, n')\\
 & & & \\
\multicolumn{4}{l}{\mathbf{data}\ \mathsf{Fin}\ (n : \mathsf{Nat})\ \mathbf{from}\ \mathsf{Nat}\ \mathbf{where}}\\
\quad \mathsf{Fin} & n & \ni & \mathsf{f0}\,[n' : \mathsf{Nat}][q : n \equiv \mathsf{suc}\ n']\\
 & & | & \mathsf{fsuc}\,[n' : \mathsf{Nat}][q : n \equiv \mathsf{suc}\ n'](fn : \mathsf{Fin}\ n')
\end{array}$$

Figure 3. Examples of ornament

Note that such a definition was unavailable in the basic presentation [McBride 2012]. Brady et al. [2004] call this operation *detagging*: the constructors of the datatype are determined by the index. The definition of Fin given in Figure 3 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 deleting the resulting, obvious proof. Hence, Fin can be further ornamented to the optimised Fin', which makes crucial use of deletion:

$$\begin{array}{lllll}
\multicolumn{5}{l}{\mathbf{data}\ \mathsf{Fin'}\ (n : \mathsf{Nat})\ \mathbf{from}\ \mathsf{Fin}\ \mathbf{where}}\\
\quad \mathsf{Fin'} & 0 & \ni & [b : \mathbb{0}] & \qquad\qquad \textit{– no constructor}\\
\quad \mathsf{Fin'} & (\mathsf{suc}\ n) & \ni & \mathsf{f0}\,[n' \triangleq n][q \triangleq \mathsf{refl}] & \\
 & & | & \mathsf{fsuc}\,[n' \triangleq n][q \triangleq \mathsf{refl}](fn : \mathsf{Fin'}\ n') &
\end{array}$$

Again, this definition was previously unavailable to us. Besides, we are making crucial use of the deletion ornament to avoid duplication. Brady et al. [2004] call this operation *forcing*: the content of the constructors – here n' and the constraint – are retrieved from the index, instead of being needlessly duplicated.

Just as the datatype declaration syntax was elaborated to IDesc codes, this high-level syntax is elaborated to ornament codes. A formal description of the translation is beyond the scope of this paper. Note that we require the order of constructors to be preserved, as their name might change from the original to the ornamented version. From the definition of an ornamented type T, we will assume the existence of its corresponding ornament code T-Orn.

As described by McBride [2012], every ornament induces an *ornamental algebra*: intuitively, an algebra that forgets the extra data, hence mapping the ornamented datatype back to its unornamented form. From an ornament $O : \mathsf{orn}\ re\, D$, there is a natural transformation from the ornamented functor down to the unornamented one, which we denote:

$$O\text{-}\mathsf{forgetNat} : (X : I \to \mathrm{SET})(j : J) \to [\![[\![O]\!]_{\mathsf{orn}} j]\!] (X \circ re) \to [\![D(re j)]\!] X$$

Applied with $\mu\, D$ for X and post-composed with in, this natural transformation induces the ornamental algebra:

$$O\text{-}\mathsf{forgetAlg} : (j : J) \to [\![[\![O]\!]_{\mathsf{orn}}\ j]\!]\ (\mu\, D \circ re) \to \mu\, D\ (re\ j)$$

In turn, this algebra induces an ornamental forgetful map denoted:

$$O\text{-}\mathsf{forget} : (j : J) \to \mu\, [\![O]\!]_{\mathsf{orn}}\ j \to \mu\, D\ (re\ j)$$

We do not re-implement these functions here: it is straightforward to update the original definitions to our setting.

3.2.1 Algebraic ornaments

An important class of datatypes is constructed by *algebraic ornamentation* over a base datatype. The idea of an algebraic ornament is to index an inductive type by the result of a fold over the original data. From the code $D : I \to \mathsf{IDesc}\, I$ and an algebra $\alpha : (i : I) \to [\![D\ i]\!]\ X \to X\ i$, there is an ornament that defines a code $D^{\alpha} : (i : I) \times X\ i \to \mathsf{IDesc}\,(i : I) \times X\ i$ with the property that:

$$\mu\, D^{\alpha}\ (i, x) \cong (t : \mu\, D\ i) \times (\!|\alpha|\!)\ t \equiv x$$

We shall indiscriminately use D^{α} to refer to the ornament and the resulting datatype. Seen as a refinement type, the correctness

[3] The inverse image of a function is defined by:

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathbf{data}\ ({}^{-1})\ [f : A \to B](b : B) : \mathrm{SET}\ \mathbf{where}}\\
\quad f^{\,-1}\ (b = f\ a) & \ni & \mathsf{inv}\,(a : A)
\end{array}$$

property states that $\mu\, D^{\alpha}\ (i, x) \cong \{t \in \mu\, D\ i \mid (\!|\alpha|\!)\ t = x\}$. The type theoretic construction of D^{α} is described by McBride [2012]. We shall not reiterate it here, the implementation being essentially the same. A categorical presentation is also given in Atkey et al. [2011] that explores the connection with refinement types.

Constructively, the correctness property gives us two (mutually inverse) functions. The direction $\mu\, D^{\alpha}(i, x) \to (t\,{:}\,\mu\, D i) \times (\!|\alpha|\!)\ t \equiv x$ relies on the generic D^{α}-forget function to compute the first component of the pair and gives us the following theorem:

$$\mathsf{coherentOrn} : \forall t^{\alpha} : \mu\, D^{\alpha}(i, x).\ (\!|\alpha|\!)\ (D^{\alpha}\text{-}\mathsf{forget}\ t^{\alpha}) \equiv x$$

This corresponds to the Recomputation theorem of McBride [2012]. We shall not reprove it here, the construction being similar. In the other direction, the isomorphism gives us a function of type:

$$(t\,{:}\,\mu\, D\ i) \times (\!|\alpha|\!)\ t \equiv x \to \mu\, D^{\alpha}(i, x)$$

Put in full and simplifying the equation, this corresponds to the function $D^{\alpha}\text{-}\mathsf{make} : (t\,{:}\,\mu\, D\ i) \to \mu\, D^{\alpha}(i, (\!|\alpha|\!)\, t)$. This corresponds to the remember function of McBride [2012]. Again, we will assume this construction here.

A typical use-case of algebraic ornaments is the implementation of semantic-preserving operations on syntax trees [McBride 2012]. For example, let us consider arithmetic expressions, which semantics is given by interpretation in Nat:

$$\begin{array}{lll@{\qquad}llcl}
\multicolumn{3}{l}{\textbf{data}\ \mathsf{Expr} : \textsc{Set}\ \textbf{where}} & \alpha_{\mathsf{eval}} & (es : [\![\mathsf{Expr\text{-}Desc}]\!]\ \mathsf{Nat}) & : & \mathsf{Nat} \\
\quad \mathsf{Expr} & \ni & \mathsf{const}\,(n : \mathsf{Nat}) & \alpha_{\mathsf{eval}} & (\mathsf{const}\,n) & \mapsto & n \\
 & \mid & \mathsf{add}\,(d\,e : \mathsf{Expr}) & \alpha_{\mathsf{eval}} & (\mathsf{add}\ m\ n) & \mapsto & m{+}n
\end{array}$$

Using the algebra α_{eval}, we construct the algebraic ornament of Expr and obtain expressions indexed by their semantics:

$$\begin{array}{lll}
\multicolumn{3}{l}{\textbf{data}\ \mathsf{Expr}^{\alpha_{\mathsf{eval}}}\ (k : \mathsf{Nat}) : \textsc{Set}\ \textbf{where}} \\
\quad \mathsf{Expr}^{\alpha_{\mathsf{eval}}}\ (k = n) & \ni & \mathsf{const}\,(n : \mathsf{Nat}) \\
\quad \mathsf{Expr}^{\alpha_{\mathsf{eval}}}\ (k = m{+}n) & \ni & \mathsf{add}\,(m n : \mathsf{Nat}) \\
 & & \qquad (d : \mathsf{Expr}^{\alpha_{\mathsf{eval}}}\ m)(e : \mathsf{Expr}^{\alpha_{\mathsf{eval}}}\ n)
\end{array}$$

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

$$\begin{array}{llcl}
\mathsf{optimize\text{-}0{+}} & (e : \mathsf{Expr}^{\alpha_{\mathsf{eval}}}\ n) & : & \mathsf{Expr}^{\alpha_{\mathsf{eval}}}\ n \\
\mathsf{optimize\text{-}0{+}} & (\mathsf{const}\,n) & \mapsto & \mathsf{const}\,n \\
\mathsf{optimize\text{-}0{+}} & (\mathsf{add}\,0\ n\ (\mathsf{const}\,0)\ e) & \mapsto & \mathsf{optimize\text{-}0{+}}\ e \\
\mathsf{optimize\text{-}0{+}} & (\mathsf{add}\ m\ n\ d\ e) & \mapsto & \mathsf{add}\ m\ n\ d\ e
\end{array}$$

If the type-checker accepts our definition, we have that, by construction, the operation preserves the semantics. We can then prune the semantics from the types using the coherentOrn theorem and retrieve the transformation on raw syntax trees.

3.2.2 Reornaments

In this paper, we are interested in a special sub-class of algebraic ornaments. As we have seen, every ornament O induces an ornamental algebra O-forgetAlg, which forgets the extra information introduced by the ornament. Hence, given a datatype D and an ornament O_D of D, we can algebraically ornament $[\![O_D]\!]_{\mathsf{orn}}$ using the ornamental algebra O_D-forgetAlg. The resulting ornament is denoted D^{O_D}. McBride [2012] calls this object the *algebraic ornament by the ornamental algebra*. For brevity, we call it the *reornament* of O_D. Again, we shall overload D^{O_D} to denote both the ornament and the resulting datatype. A standard example of reornament is Vec: it is the reornament of List-Orn. Put otherwise, a vector is the algebraic ornament of List by the algebra computing its length, i.e. the ornamental algebra from List to Nat.

Reornaments can be implemented straightforwardly by unfolding their definition: 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.

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 Vec reornament would be as follows:

- We retrieve the index, hence obtaining n ;
- By inspecting the ornament List-Orn, we obtain *exactly* the information by which n is *extended* into a list: if $n = 0$, no supplementary information is needed and if $n = \mathsf{suc}\, n'$, we need to extend it with an $a : A$. We call this the Extension of n ;
- By inspecting the ornament List-Orn again, we obtain the recursive structure of the reornament by *deleting* the data already fully determined by the index and its extension, and *refining* the indexing discipline: the tail of a vector of size $\mathsf{suc}\, n'$ is a vector of size n'. The recursive structure is denoted by Structure.

A reornament is thus the Extension of its index followed by the recursive structure as defined by Structure[4]. Based on this intuition, we define the associated reornament at index $t = \mathsf{in}\, xs : \mu\, D$ by, first, inserting the valid extensions of t with Extension, then, building the recursive structure using Structure:

$$\begin{array}{lcl}
\mathsf{reornament}\ (O : \mathsf{orn}\ re\ D) & : & \mathsf{orn}\ \pi_0\ [\![O]\!]_{\mathsf{orn}} \\
\mathsf{reornament}\ O & \mapsto & \lambda(j, \mathsf{in}\ xs).\ \mathsf{insert}\ (\mathsf{Extension}\ (O\ j)\ xs)\ \lambda e. \\
 & & \qquad\qquad \mathsf{Structure}\ (O\ j)\ xs\ e
\end{array}$$

Applied to the reornament of List-Orn, this construction gives the fully Brady-optimised – detagged and forced – version of Vec, here written in full:

$$\begin{array}{lcl}
\multicolumn{3}{l}{\textbf{data}\ \mathsf{Vec}\ [A : \textsc{Set}](n : \mathsf{Nat}) : \textsc{Set}\ \textbf{where}} \\
\quad \mathsf{Vec}_A\ \ 0 & \ni & \mathsf{nil} \\
\quad \mathsf{Vec}_A\ (\mathsf{suc}\ n) & \ni & \mathsf{cons}\,(a : A)(vs : \mathsf{Vec}_A\ n)
\end{array}$$

Note that our ability to *compute* over the index is crucial for this construction to work. Also, it is isomorphic to the datatype one would have obtained with the algebraic ornament of the ornamental algebra. Consequently, the correctness property of algebraic ornaments is still valid here: constructively, we get the coherentOrn theorem in one direction and the $*$-make function in the other.

In this Section, we have adapted the notion of ornament to our universe of datatypes. In doing so, we have introduced the concept of a deletion ornament, using the indexing to remove duplicated information in the datatypes. This has proved useful to simplify the definition of reornaments. We shall see how this can be turned to our advantage when we transport functions across ornaments.

4. A universe of functions and their ornaments

We are now going to generalise the notion of ornament to functions. In order to do this, we first need to be able, in type theory, to manipulate functions and especially their types. Hence, we define a universe of functions. With it, we will be able to write generic programs over the class of functions captured by our universe.

Using this technology, we define a functional ornament as a decoration over the universe of functions. 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*: a patch is the type of the functions that satisfy the coherence property *by construction*. Finally, and still writing generic programs, we show how we can automatically project the lifting and its coherence certificate out of a patch.

[4] For space reasons, we shall refer the reader to the companion technical report for the type-theoretic definition of Extension and Structure. Their exact definition is not necessary for the understanding of this paper.

$$\begin{array}{lll}
\textbf{data}\ \mathsf{FunOrn}\ (T : \mathsf{Type}) : \textsc{Set}_1\ \textbf{where} & & \\
\quad \mathsf{FunOrn}\ (\mu\{D\ i\}\!\to T) & \ni & \mu^{+}\{(O : \mathsf{orn}\ \mathit{re}\ D)\ (j : \mathit{re}^{-1}\ i)\}\!\to (T^{+} : \mathsf{FunOrn}\ T) \\
\quad \mathsf{FunOrn}\ (\mu\{D\ i\}\!\times T) & \ni & \mu^{+}\{(O : \mathsf{orn}\ \mathit{re}\ D)\ (j : \mathit{re}^{-1}\ i)\}\!\times (T^{+} : \mathsf{FunOrn}\ T) \\
\quad \mathsf{FunOrn}\quad 1 & \ni & 1
\end{array}$$

(a) Code

$$\begin{array}{lll}
\llbracket (T^{+} : \mathsf{FunOrn}\ T) \rrbracket_{\mathsf{FunOrn}} & : & \textsc{Set} \\
\llbracket \mu^{+}\{O\ (\mathsf{inv}\ j)\}\!\to T^{+} \rrbracket_{\mathsf{FunOrn}} & \mapsto & \mu\ \llbracket O \rrbracket_{\mathsf{orn}}\ j \to \llbracket T^{+} \rrbracket_{\mathsf{FunOrn}} \\
\llbracket \mu^{+}\{O\ (\mathsf{inv}\ j)\}\!\times T^{+} \rrbracket_{\mathsf{FunOrn}} & \mapsto & \mu\ \llbracket O \rrbracket_{\mathsf{orn}}\ j \times \llbracket T^{+} \rrbracket_{\mathsf{FunOrn}} \\
\llbracket 1 \rrbracket_{\mathsf{FunOrn}} & \mapsto & \mathbb{1}
\end{array}$$

(b) Interpretation

Figure 4. Universe of functional ornaments

4.1 A universe of functions

For clarity of exposition, we restrict our language of types to the bare minimum: a type can either be an exponential which domain is an inductive object, or a product which first component is an inductive object, or the unit type – used as a termination symbol:

$$\begin{array}{ll}
\textbf{data}\ \mathsf{Type} : \textsc{Set}_1\ \textbf{where} & \\
\quad \mathsf{Type} \ni & \mu\{(D : I \to \mathsf{IDesc}\ I)\ (i : I)\}\!\to (T : \mathsf{Type}) \\
\quad\quad\ \ \mid & \mu\{(D : I \to \mathsf{IDesc}\ I)\ (i : I)\}\!\times (T : \mathsf{Type}) \\
\quad\quad\ \ \mid & 1
\end{array}$$

Hence, 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:

$$\begin{array}{lll}
\llbracket (T : \mathsf{Type}) \rrbracket_{\mathsf{Type}} & : & \textsc{Set} \\
\llbracket \mu\{D\ i\}\!\to T \rrbracket_{\mathsf{Type}} & \mapsto & \mu\ D\ i \to \llbracket T \rrbracket_{\mathsf{Type}} \\
\llbracket \mu\{D\ i\}\!\times T \rrbracket_{\mathsf{Type}} & \mapsto & \mu\ D\ i \times \llbracket T \rrbracket_{\mathsf{Type}} \\
\llbracket 1 \rrbracket_{\mathsf{Type}} & \mapsto & \mathbb{1}
\end{array}$$

The constructions we develop below could be extended to a more powerful universe – such as one supporting non-inductive sets or having dependent functions and pairs. However, this would needlessly complicate our exposition.

Example 1 (Coding $\_ < \_$)**.** Written in the universe of function types, the type of $\_ < \_$ is:

$$\begin{array}{lll}
\mathsf{type}\!< & : & \mathsf{Type} \\
\mathsf{type}\!< & \mapsto & \mu\{\mathsf{Nat\text{-}Desc} *\}\!\to \mu\{\mathsf{Nat\text{-}Desc} *\}\!\to \mu\{\mathsf{Bool\text{-}Desc} *\}\!\times 1
\end{array}$$

The implementation of $\_ < \_$ is essentially the same as earlier, excepted that it must now return a pair of a boolean and an inhabitant of the unit type. To be explicit about the recursion pattern of this function, we make use of Epigram's *by* ($\Leftarrow$) construct:

$$\begin{array}{rcll}
\_ & < & \_ & : \llbracket \mathsf{type}\!< \rrbracket_{\mathsf{Type}} \\
m & < & n & \Leftarrow \mathsf{Nat\text{-}case}\ n \\
m & < & 0 & \mapsto (\mathsf{false}, *) \\
m & < & \mathsf{suc}\ n & \Leftarrow \mathsf{Nat\text{-}elim}\ m \\
0 & < & \mathsf{suc}\ n & \mapsto (\mathsf{true}, *) \\
\mathsf{suc}\ m & < & \mathsf{suc}\ n & \mapsto m < n
\end{array}$$

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

Example 2 (Coding $\_ + \_$)**.** Written in the universe of function types, the type of $\_ + \_$ is:

$$\begin{array}{lll}
\mathsf{type}\!+ & : & \mathsf{Type} \\
\mathsf{type}\!+ & \mapsto & \mu\{\mathsf{Nat\text{-}Desc} *\}\!\to \mu\{\mathsf{Nat\text{-}Desc} *\}\!\to \mu\{\mathsf{Nat\text{-}Desc} *\}\!\times 1
\end{array}$$

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

4.2 Functional ornament

From the universe of function types, it is now straightforward to define the notion of functional ornament: we traverse the type code and ornament the inductive types as we go. Note that it is always possible to leave an object unornamented: we ornament by the identity that simply copies the original definition. Hence, we obtain the definition given in Fig. 4(a). From a functional ornament, we get the type of the liftings by interpreting each ornaments (Fig. 4(b)). This defines the universe of functional ornaments.

We will want our ornamented function to be *coherent* with the base function we started with: for a function $f : \mu\,D \to \mu\,E$, the ornamented function $f^{+} : \mu\,\llbracket O_D \rrbracket_{\mathsf{orn}} \to \mu\,\llbracket O_E \rrbracket_{\mathsf{orn}}$ is said to be coherent with f if it satisfies the following equation:

$$\forall x^{+} : \mu\,\llbracket O_D \rrbracket_{\mathsf{orn}} .\, f\ (O_D\text{-}\mathsf{forget}\ x^{+}) \equiv O_E\text{-}\mathsf{forget}\ (f^{+}\ x^{+})$$

To generalise the definition of coherence to any arity, we proceed by induction over the universe of functional ornaments:

$$\begin{array}{llllll}
\multicolumn{6}{l}{\mathsf{Coherence}(T^{+} : \mathsf{FunOrn}\ T)(f : \llbracket T \rrbracket_{\mathsf{Type}})(f^{+} : \llbracket T^{+} \rrbracket_{\mathsf{FunOrn}}) : \textsc{Set}} \\
\mathsf{Coherence} & (\mu^{+}\{O\ (\mathsf{inv}\ j)\}\!\to T^{+}) & f & f^{+} & \mapsto & \\
\multicolumn{6}{l}{\quad\quad \forall x^{+} : \mu\,\llbracket O \rrbracket_{\mathsf{orn}}\ j . \mathsf{Coherence}\ T^{+}\ (f\ (\mathsf{forgetOrn}\ x^{+}))\ (f^{+} x^{+})} \\
\mathsf{Coherence} & (\mu^{+}\{O\ (\mathsf{inv}\ j)\}\!\times T^{+}) & (x, xs) & (x^{+}, xs^{+}) & \mapsto & \\
\multicolumn{6}{l}{\quad\quad x \equiv \mathsf{forgetOrn}\ x^{+} \times \mathsf{Coherence}\ T^{+}\ xs\ xs^{+}} \\
\mathsf{Coherence} & 1 & * & * & \mapsto & \mathbb{1}
\end{array}$$

Example 3 (Ornamenting $\mathsf{type}\!<$ to describe lookup)**.** In Section 2, we have identified the ornaments involved to transport the type of $\_ < \_$ to obtain the type of lookup. From there, we give the functional ornament describing the type of the lookup function:

$$\begin{array}{lll}
\mathsf{typeLookup} & : & \mathsf{FunOrn}\ \mathsf{type}\!< \\
\mathsf{typeLookup} & \mapsto & \mu^{+}\{\mathsf{idO}_{\mathsf{Nat}} *\}\!\to \\
 & & \quad \mu^{+}\{\mathsf{List\text{-}Orn} *\}\!\to \\
 & & \quad\quad \mu^{+}\{\mathsf{Maybe\text{-}Orn} *\}\!\times 1
\end{array}$$

The user can verify that $\llbracket \mathsf{typeLookup} \rrbracket_{\mathsf{FunOrn}}$ gives us the type of the lookup function, up to multiplication by $\mathbb{1}$. Also, computing $\mathsf{Coherence}\ \mathsf{typeLookup}\ (\_ < \_)$ gives the expected result:

$$\begin{array}{l}
\lambda f^{+} : \llbracket \mathsf{typeLookup} \rrbracket_{\mathsf{FunOrn}} . \\
\quad \forall n : \mathsf{Nat} . \forall xs : \mathsf{List}_A . \mathsf{isJust}\ (f^{+}\ n\ xs) \equiv n < \mathsf{length}\ xs
\end{array}$$

Note that 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.

Example 4 (Ornamenting $\mathsf{type}\!+$ to describe $\_ +\!\!+ \_$)**.** The functional ornament of $\mathsf{type}\!+$ makes only use of the ornamentation of Nat into List_A:

$$\begin{array}{lll}
\mathsf{type}\!+\!\!+ & : & \mathsf{FunOrn}\ \mathsf{type}\!+ \\
\mathsf{type}\!+\!\!+ & \mapsto & \mu^{+}\{\mathsf{List\text{-}Orn} *\}\!\to \\
 & & \quad \mu^{+}\{\mathsf{List\text{-}Orn} *\}\!\to \\
 & & \quad\quad \mu^{+}\{\mathsf{List\text{-}Orn} *\}\!\times 1
\end{array}$$

Again, computing $\llbracket \mathsf{type}\!+\!\!+ \rrbracket_{\mathsf{FunOrn}}$ indeed gives us the type of $\_ +\!\!+ \_$ while $\mathsf{Coherence}\ \mathsf{type}\!+\!\!+\ (\_ + \_)$ correctly captures our requirement that list append preserves the length of its arguments. As before, the list append function is not the only valid lifting: one could for example consider a function that reverts the first list and appends it to the second one.

4.3 Patches

By definition of a functional ornament, the lifting of a base function $f : \llbracket T \rrbracket_{\mathsf{Type}}$ is a function f^{+} of type $\llbracket T^{+} \rrbracket_{\mathsf{FunOrn}}$ satisfying the coherence property $\mathsf{Coherence}\ T^{+}\ f$. To implement a lifting that is coherent, we might ask the user to first implement the lifting f^{+} and then prove it coherent. 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.

Note that we are looking for an equivalence here: we will define patches so that they are in bijection with liftings satisfying a coherence property, informally:

$$\mathsf{Patch}\, T\, T^{+}\, f \cong (f^{+} : [\![T^{+}]\!]_{\mathsf{FunOrn}}) \times \mathsf{Coherence}\, T^{+}\, f\, f^{+} \qquad (1)$$

In this paper, we constructively use this bijection in the left to right direction: having implemented a patch f^{++} of type $\mathsf{Patch}\; T\; T^{+} f$, we will show, in the next Section, how we can extract a lifting together with its coherence proof.

Before giving the generic construction of the Patch object, let us first work through the $\_ < \_$ example. After having functionally ornamented $\_ < \_$ with typeLookup, the lifting function f^{+} and coherence property can be represented by the following pair:

$$(f^{+} : \mathsf{Nat} \times \mathsf{List}_A \to \mathsf{Maybe}_A) \times$$
$$\forall m : \mathsf{Nat}. \forall as : \mathsf{List}_A .\, m < \mathsf{List\text{-}forget}\; as \equiv \mathsf{Maybe\text{-}forget}\; (f^{+}\; m\; as)$$

Applying dependent choice, this is equivalent to:

$$\cong (m : \mathsf{Nat}) \times (n : \mathsf{Nat}) \times (as : \mathsf{List}_A) \times \mathsf{List\text{-}forget}\; as \equiv n \to$$
$$(ma : \mathsf{Maybe}_A) \times \mathsf{Maybe\text{-}forget}\; ma \equiv m < n$$

Now, by definition of reornaments, we have that:

$$(as : \mathsf{List}_A) \times \mathsf{List\text{-}forget}\; as \equiv n \cong \mathsf{Vec}_A\; n \qquad \text{and}$$
$$(ma : \mathsf{Maybe}_A) \times \mathsf{Maybe\text{-}forget}\; ma \equiv b \cong \mathsf{IMaybe}_A\; b$$

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

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

Which is thus equivalent to a pair of a lifting and its coherence.

Intuitively, the Patch construction consists in turning the pairs of data and their algebraically defined constraint into equivalent reornaments. The coherence property of reornaments tells us that projecting the ornamented function down to its unornamented 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 (here, introducing m and n) and index the return types by the result of the unornamented function (here, indexing IMaybe_A by $m < n$).

To build this type generically, we simply proceed by induction over the functional ornament. Upon an argument (i.e. a $\mu^{+}\{O\;\}\!\!\to$), we introduce a fresh index and the reornament of O. Upon a result (i.e. a $\mu^{+}\{O\;\}\!\times$), we ask for a reornament of O indexed by the result of the base function.

$$\begin{array}{llllll}
\multicolumn{6}{l}{\mathsf{Patch}(T : \mathsf{Type})(T^{+} : \mathsf{FunOrn}\; T)(f : [\![T]\!]_{\mathsf{Type}}) \;:\; \textsc{Set}} \\
\mathsf{Patch} & (\mu\{D\,(re\,j)\,\}\!\!\to T) & (\mu^{+}\{O\,(\mathsf{inv}\,j)\,\}\!\!\to T^{+}) & f & \mapsto \\
\multicolumn{6}{l}{\qquad (x : \mu\, D\; (re\; j)) \to \mu\, D^{O}\; (j, x) \to \mathsf{Patch}\; T\; T^{+}\; (f\; x)} \\
\mathsf{Patch} & (\mu\{D\,(re\,j)\,\}\!\times T) & (\mu^{+}\{O\,(\mathsf{inv}\,j)\,\}\!\times T^{+}) & (x, xs) & \mapsto \\
\multicolumn{6}{l}{\qquad \mu\, D^{O}\; (j, x) \times \mathsf{Patch}\; T\; T^{+}\; xs} \\
\mathsf{Patch} & 1 & 1 & * & \mapsto \mathbb{1}
\end{array}$$

Example 5 (Patch of typeLookup)**.** The type of the coherent liftings of $\_ < \_$ by typeLookup, as defined by the Patch of $\_ < \_$ by typeLookup, computes to:

$$(m : \mathsf{Nat}) \to (m^{+} : \mu\, \mathsf{Nat}^{\mathsf{idO}_{\mathsf{Nat}}}\; m) \to$$
$$(n : \mathsf{Nat}) \to (vs : \mu\, \mathsf{Nat}^{\mathsf{List}_A}\; n) \to \mu\, \mathsf{Bool}^{\mathsf{Maybe}_A}\; (m < n) \times \mathbb{1}$$

Note that $\mu\, \mathsf{Nat}^{\mathsf{idO}_{\mathsf{Nat}}}\; n$ 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. Hence, we discard this argument as computationally uninteresting. On the other hand, $\mathsf{Nat}^{\mathsf{List}_A}$ and $\mathsf{Bool}^{\mathsf{Maybe}_A}$ are, respectively, the previously introduced Vec_A and IMaybe_A types.

Example 6 (Patch of type+)**.** Similarly, the Patch of $\_ + \_$ by type+ computes to the type of the vector append function:

$$(m : \mathsf{Nat}) \to (xs : \mathsf{Nat}^{\mathsf{List}_A}\; m) \to$$
$$(n : \mathsf{Nat}) \to (ys : \mathsf{Nat}^{\mathsf{List}_A}\; m) \to \mathsf{Nat}^{\mathsf{List}_A}\; (m + n) \times \mathbb{1}$$

Discussion: While these precisely indexed functions remove the burden of theorem proving, this solution is not relevant in all situations. For instance, if we were to implement a length-preserving list reversal, our patching machinery would ask us to implement $\mathsf{vrev} : \mathsf{Vec}_A\; n \to \mathsf{Vec}_A\; n$ that will inevitably require some proving to match up the types: we must appeal to the equational theory of addition – in this case, $n + 1 \equiv \mathsf{suc}\; n$ – and this is beyond the grasp of our type-checker, which can only decide definitional identities. 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 4.2) and implement the lifting and its coherence proof manually, rather than using patches. However, 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 need to be definitionally true, either because our logic has a rich definitional equality, or because we rely on operations that satisfy these identities by definition.

4.4 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 isomorphism (1) of the previous Section through our constructive glasses: indeed, as the Patch type is isomorphic to the set of liftings satisfying the coherence property, we effectively get a function taking every Patch to a lifting and its coherence proof. 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.

We call *patching* the action of projecting the coherent lifting from a Patch function. Again, it is defined by mere induction over the functional ornament. When ornamented arguments are introduced (i.e. with $\mu^{+}\{O\;\}\!\!\to$), 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 $f\; (\mathsf{forgetOrn}\; x^{+})$ and we can also make the reornament to compute $f^{++}\; \_\; (\mathsf{makeAlgOrn}\; x^{+})$. When an ornamented result is to be returned, we simply forget the reornamentation computed by the coherent lifting:

$$\begin{array}{llllll}
\multicolumn{6}{l}{\mathsf{patch}\; (T^{+} : \mathsf{FunOrn}\; T)(f : [\![T]\!]_{\mathsf{Type}})(p : \mathsf{Patch}\; T\; T^{+}\; f) \;:} \\
\multicolumn{6}{l}{\qquad\qquad [\![T^{+}]\!]_{\mathsf{FunOrn}}} \\
\mathsf{patch} & (\mu^{+}\{O\,(\mathsf{inv}\,j)\,\}\!\!\to T^{+}) & f & f^{++} & \mapsto \\
\multicolumn{6}{l}{\quad \lambda x^{+}.\, \mathsf{patch}\; (f\; (\mathsf{forgetOrn}\; x^{+}))} \\
\multicolumn{6}{l}{\qquad\qquad (f^{++}\; (\mathsf{forgetOrn}\; x^{+})\; (\mathsf{makeAlgOrn}\; x^{+}))} \\
\mathsf{patch} & (\mu^{+}\{O\,(\mathsf{inv}\,j)\,\}\!\times T^{+}) & (x, xs) & (x^{++}, xs^{++}) & \mapsto \\
\multicolumn{6}{l}{\quad (\mathsf{forgetOrn}\; x^{++}, \mathsf{patch}\; T^{+}\; xs\; xs^{++})} \\
\mathsf{patch} & 1 & * & * & \mapsto *
\end{array}$$

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

$$\begin{array}{llllll}
\multicolumn{6}{l}{\mathsf{coherence}\; (T^{+} : \mathsf{FunOrn}\; T)(f : [\![T]\!]_{\mathsf{Type}})(p : \mathsf{Patch}\; T\; T^{+}\; f) \;:} \\
\multicolumn{6}{l}{\qquad\qquad \mathsf{Coherence}\; T^{+}\; f\; (\mathsf{patch}\; T^{+}\; f\; p)} \\
\mathsf{coherence} & (\mu^{+}\{O\,(\mathsf{inv}\,j)\,\}\!\!\to T^{+}) & f & p & \mapsto \\
\multicolumn{6}{l}{\quad \lambda x^{+}.\, \mathsf{coherence}\, T^{+}\; (f\; (\mathsf{forgetOrn}\; x^{+}))} \\
\multicolumn{6}{l}{\qquad\qquad (p\; (\mathsf{forgetOrn}\; x^{+})\; (\mathsf{makeAlgOrn}\; x^{+}))} \\
\mathsf{coherence} & (\mu^{+}\{O\,(\mathsf{inv}\,j)\,\}\!\times T^{+}) & (x, xs) & (x^{+}, p) & \mapsto \\
\multicolumn{6}{l}{\quad (\mathsf{coherentOrn}\; x^{+}, \mathsf{coherence}\; T^{+}\; xs\; p)} \\
\mathsf{coherence} & 1 & * & * & \mapsto *
\end{array}$$

```
_        <   _     :  ⟦type<⟧_Type          ilookup      (m : Nat) (vs : Vec_A n)  :  IMaybe_A (m < n)
m        <   n     ⇐  Nat-case n            ilookup       m           vs          ⇐  Vector-case vs
  m      <   0     ↦  (false, ∗)              ilookup     m           nil         ↦  nothing
  m      < suc n   ⇐  Nat-elim m              ilookup     m       (cons a vs)     ⇐  Nat-elim m
    0    < suc n   ↦  (true, ∗)                 ilookup   0       (cons a vs)     ↦  just a
    suc m < suc n  ↦  m < n                     ilookup (suc m)   (cons a vs)     ↦  ilookup m vs
```

Figure 5. Implementations of $\_ < \_$ and ilookup

Example 7 (Obtaining lookup and its coherence certificate, for free)**.** This last step is a mere application of the patch and coherence functions. Hence, we define lookup as follows:

```
lookup  :  ⟦typeLookup⟧_FunOrn
lookup  ↦  patch typeLookup (_<_) ilookup
```

And we get its coherence proof, here spelled in full:

```
cohLookup (n : Nat) (xs : List_A)  :
      Maybe-forget (π₀(lookup n xs)) ≡ π₀(n < List-forget xs)
cohLookup    n          xs        ↦
   coherence typeLookup (_<_) ilookup n xs
```

Example 8 (Obtaining $\_ +\!\!+ \_$ and its coherence certificate, for free)**.** Assuming that we have implemented the coherent lifting vappend, we obtain concatenation of lists and its coherence proof by simply running our generic machinery:

```
++  :  ⟦type++⟧_FunOrn
++  ↦  patch type++ (_+_) vappend

coh++ (xs : List_A) (ys : List_A)  :
 List-forget (π₀(xs ++ ys)) ≡ π₀((List-forget xs) +(List-forget ys))
coh++      xs           ys       ↦
   coherence type++ (_+_) vappend xs ys
```

Looking back at the manual construction in Section 2, we can measure the progress we have made: while we had to duplicate entirely the type signature of lookup and its coherence proof, we can now write down a functional ornament and these are generated for us. This is not just convenient: by giving a functional ornament, we establish a strong connection between two functions. By pinning down this connection with the universe of functional ornaments, we turn this knowledge into an effective object that can be manipulated and reasoned about within the type theory. We make use of this concrete object 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 as the type of the liftings coherent by construction. As we have seen, its construction and further projection down to a lifting is now 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. We have now made this second alternative just as accessible as the first one. And, from a programming perspective, the second approach is much more appealing. In a word, we have made an appealing technique extremely cheap!

Finally, we shall reiterate that none of the above constructions involve extending the type theory: using our universe of datatypes, functional ornaments are internalised as a few generic programs and inductive types. For systems such as Agda, Coq, or Haskell with GADTs, this technology would need to be provided at the meta-level. However, the fact that our constructions type-check in our system suggests that adding these constructions at the meta-level is consistent with a pre-existing meta-theory.

5. Lazy programmers, clever 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. 5). First, both functions follow the same recursion pattern: case analysis over n/vs followed by induction 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: it 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 duplicate* and only provides *strictly necessary* inputs. We are primarily interested in transporting two forms of structure:

- **Recursion pattern:** if the base function is a fold $(\!|\alpha|\!)$ and the user provides us with a *coherent algebra* $\hat{\beta}$ of α, we automatically construct the coherent lifting $(\!|\hat{\beta}|\!)$ of $(\!|\alpha|\!)$;
- **Returned constructor:** if the base function returns a constructor C and the user provides us with a *coherent extension* $\hat{C}$ of C, we automatically 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, automatically and generically, build the lifted folds and values.

5.1 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 transportation is that the lifting uses exactly the same structure to compute its results. Hence, in the majority of the cases, we could just ask the computer to use the induction principle induced by the base one: the only task left to the user will be to give the algebra. For clarity of exposition, we restrict ourselves to transporting folds. However, the treatment of induction is essentially the same, as hinted by the fact that induction can be reduced to folds [Fumex et al. 2011].

To illustrate this approach, we work through a concrete example: we derive hd : $\mathsf{List}_A \to \mathsf{Maybe}_A$ from isSuc : Nat → Bool by transporting the algebra. For the sake of argument, we artificially define isSuc by a fold:

```
isSuc (n : Nat)   :  Bool
isSuc    n        ↦  (|α_isSuc|) n    where
   α_isSuc (xs : ⟦Nat-Desc⟧ Bool)  :  Bool
   α_isSuc          '0             ↦  false
   α_isSuc       ('suc xs)         ↦  true
```

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

$$\alpha_{\mathsf{hd}} : [\![\mathsf{List\text{-}Desc}]\!]\ \mathsf{Maybe}_A \to \mathsf{Maybe}_A$$

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

$$\begin{array}{llll}
\mathsf{ihd} & (vs : \mathsf{Vec}_A\ n) & : & \mathsf{IMaybe}_A\ \mathsf{isSuc}\ n \\
\mathsf{ihd} & & \overset{\mathsf{lift}}{\Leftarrow} & \mathsf{lift\text{-}fold} \\
\{?\} & & &
\end{array}$$

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

$$\begin{array}{llll}
\mathsf{ihd}\ (vs : \mathsf{Vec}_A\ n) & : & \mathsf{IMaybe}_A\ (\mathsf{isSuc}\ n) & \\
\mathsf{ihd} & \overset{\mathsf{lift}}{\Leftarrow} & \mathsf{lift\text{-}fold}\quad \mathbf{where} & \\
\quad \alpha_{\mathsf{ihd}}\ (vs : [\![\mathsf{Vec\text{-}Desc}]\!]\ (\lambda n'.\ \mathsf{IMaybe}_A\ (\mathsf{isSuc}\ n'))\ n) & : & \mathsf{IMaybe}_A\ (\mathsf{isSuc}\ n) & \\
\quad \alpha_{\mathsf{ihd}}\qquad\qquad\qquad\qquad\qquad ’\mathsf{nil} & \{?\} & & \\
\quad \alpha_{\mathsf{ihd}}\qquad\qquad\qquad\quad (’\mathsf{cons}\ a\ xs) & \{?\} & &
\end{array}$$

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

$$\begin{array}{lll}
\mathsf{ihd}\ (vs : \mathsf{Vec}_A\ n) & : & \mathsf{IMaybe}_A\ (\mathsf{isSuc}\ n) \\
\mathsf{ihd} & \overset{\mathsf{lift}}{\Leftarrow} & \mathsf{lift\text{-}fold}\quad \mathbf{where} \\
\quad \alpha_{\mathsf{ihd}}\ (vs : [\![\mathsf{Vec\text{-}Desc}]\!]\ (\lambda n'.\ \mathsf{IMaybe}_A\ (\mathsf{isSuc}\ n'))\ n) & : & \mathsf{IMaybe}_A\ (\mathsf{isSuc}\ n) \\
\quad \alpha_{\mathsf{ihd}}\qquad\qquad\qquad\qquad\qquad ’\mathsf{nil} & \overset{\mathsf{lift}}{\mapsto} & \{?\} \\
\quad \alpha_{\mathsf{ihd}}\qquad\qquad\qquad\quad (’\mathsf{cons}\ a\ xs) & \overset{\mathsf{lift}}{\mapsto} & \{?\}
\end{array}$$

(d) Result of lifting constructors:
(system output)

$$\begin{array}{lll}
\mathsf{ihd}\ (vs : \mathsf{Vec}_A\ n) & : & \mathsf{IMaybe}_A\ (\mathsf{isSuc}\ n) \\
\mathsf{ihd} & \overset{\mathsf{lift}}{\Leftarrow} & \mathsf{lift\text{-}fold}\quad \mathbf{where} \\
\quad \alpha_{\mathsf{ihd}}\ (vs : [\![\mathsf{Vec\text{-}Desc}]\!]\ (\lambda n'.\ \mathsf{IMaybe}_A\ (\mathsf{isSuc}\ n'))\ n) & : & \mathsf{IMaybe}_A\ (\mathsf{isSuc}\ n) \\
\quad \alpha_{\mathsf{ihd}}\qquad\qquad\qquad\qquad\qquad ’\mathsf{nil} & \overset{\mathsf{lift}}{\mapsto} & \mathsf{nothing}\ \{? : \mathbb{1}\}\ [\ \{? : \mathbb{1}\}\ ] \\
\quad \alpha_{\mathsf{ihd}}\qquad\qquad\qquad\quad (’\mathsf{cons}\ a\ xs) & \overset{\mathsf{lift}}{\mapsto} & \mathsf{just}\ \{? : A\}\ [\ \{? : \mathbb{1}\}\ ]
\end{array}$$

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

$$\begin{array}{lll}
\mathsf{ihd}\ (vs : \mathsf{Vec}_A\ n) & : & \mathsf{IMaybe}_A\ (\mathsf{isSuc}\ n) \\
\mathsf{ihd}\qquad vs & \mapsto & \mathsf{lift\text{-}fold}\ \alpha_{\mathsf{isSuc}}\ \alpha_{\mathsf{ihd}}\quad \mathbf{where} \\
\quad \alpha_{\mathsf{ihd}}\ (vs : [\![\mathsf{Vec\text{-}Desc}]\!]\ (\lambda n'.\ \mathsf{IMaybe}_A\ (\mathsf{isSuc}\ n'))\ n) & : & \mathsf{IMaybe}_A\ (\mathsf{isSuc}\ n) \\
\quad \alpha_{\mathsf{ihd}}\qquad\qquad\qquad\qquad\qquad ’\mathsf{nil} & \mapsto & \mathsf{lift\text{-}constructor}\ ’\mathsf{nil}\ \{? : \mathbb{1}\}\ \ \{? : \mathbb{1}\}\ * \\
\quad \alpha_{\mathsf{ihd}}\qquad\qquad\qquad\quad (’\mathsf{cons}\ a\ xs) & \mapsto & \mathsf{lift\text{-}constructor}\ (’\mathsf{suc}\ n)\ \{? : A\}\ \ \{? : \mathbb{1}\}\ *
\end{array}$$

Figure 6. Guided implementation of ihd

such that its fold is coherent. By the fold-fusion theorem [Bird and de Moor 1997], it is sufficient (but not necessary) for α_{hd} to satisfy the following condition:

$$\begin{array}{l}
\forall ms : [\![\mathsf{List\text{-}Desc}]\!]\ \mathsf{Maybe}_A. \\
\quad \mathsf{isJust}\ (\alpha_{\mathsf{hd}}\ ms) \equiv \alpha_{\mathsf{isSuc}}\ (\mathsf{List\text{-}forgetNat}([\![\mathsf{List\text{-}Desc}]\!]\ \mathsf{isJust}\ ms))
\end{array}$$

Following the same methodology we applied to define the Patch type, we can massage the type of α_{hd} and its coherence condition to obtain an equivalent definition enforcing the coherence by indexing. In this case, the natural candidate is:

$$\alpha_{\mathsf{ihd}} : [\![\mathsf{Vec\text{-}Desc}]\!]\ (\lambda n'.\ \mathsf{IMaybe}_A\ (\mathsf{isSuc}\ n'))\ n \to \mathsf{IMaybe}_A\ (\mathsf{isSuc}\ n)$$

This construction generalises to any functional ornament. That is, from an algebra

$$\alpha : (i : I) \to [\![D\ i]\!]\ (\lambda\_.\ [\![T]\!]_{\mathsf{Type}}) \to [\![T]\!]_{\mathsf{Type}}$$

together with an ornament $O_D : \mathsf{orn}\ reD$ and a functional ornament $T^+ : \mathsf{FunOrn}\ T$, the type of coherent algebras for α is:

$$\begin{array}{l}
\hat{\beta} : (j : J)(t : \mu\, D\ (re\ j)) \to \\
\qquad [\![D^O\ (j, t)]\!]\ (\lambda(j, t).\ \mathsf{Patch}\ T\ (\llparenthesis\alpha\rrparenthesis\ t)\ T^+) \to \\
\qquad\quad \mathsf{Patch}\ T\ (\llparenthesis\alpha\rrparenthesis\ t)\ T^+
\end{array}$$

It can formally be proved that algebras of this type capture exactly the algebras satisfying the coherence condition. Constructively, we get that such a coherent algebra induces a coherent lifting, by a mere fold of the coherent algebra:

$$\begin{array}{l}
\mathsf{lift\text{-}fold}\ (\alpha : (i : I) \to [\![D\ i]\!]\ (\lambda\_.\ [\![T]\!]_{\mathsf{Type}}) \to [\![T]\!]_{\mathsf{Type}}) \\
\qquad\quad (\hat{\beta} :\ \ (j : J)(t : \mu\, D\ (re\ j)) \to \\
\qquad\qquad\qquad [\![D^O\ (j, t)]\!]\ (\lambda(j, t).\ \mathsf{Patch}\ T\ (\llparenthesis\alpha\rrparenthesis\ t)\ T^+) \to \\
\qquad\qquad\qquad \mathsf{Patch}\ T\ (\llparenthesis\alpha\rrparenthesis\ t)\ T^+) \\
\qquad\quad :\ \mathsf{Patch}\ (\mu\{D\ (re\ j)\ \} \mapsto T)\ \llparenthesis\alpha\rrparenthesis\ (\mu^+\{O\ j\ \} \mapsto T^+) \\
\mathsf{lift\text{-}fold}\ \alpha\ \hat{\beta}\ \mapsto\ \lambda x.\ \lambda x^{++}.\ \llparenthesis\hat{\beta}\rrparenthesis\ x^{++}
\end{array}$$

Generalising this idea, we can similarly lift induction: we denote lift-ind the corresponding clever constructor. Lifting case analysis is now simple, as case analysis is derivable from induction by stripping out the induction hypotheses [McBride et al. 2004].

Example 9 (Transporting the recursion pattern of isSuc)**.** We can now apply our generic machinery to transport isSuc to hd: in a high-level notation, we would write the command of Fig. 6(a). To this command, an interactive system would respond by automatically generating the algebra, as shown in Fig. 6(b). In the low-level type theory, this would elaborate to the following term:

$$\begin{array}{lll}
\mathsf{ihd}\ (vs : \mathsf{Vec}_A\ n) & : & \mathsf{IMaybe}_A\ (\mathsf{isSuc}\ n) \\
\mathsf{ihd}\qquad vs & \mapsto & \mathsf{lift\text{-}fold}\ \alpha_{\mathsf{isSuc}}\ \alpha_{\mathsf{ihd}}\quad \mathbf{where} \\
\quad \alpha_{\mathsf{ihd}}\ (vs : [\![\mathsf{Vec\text{-}Desc}]\!]\ (\lambda n'.\ \mathsf{IMaybe}_A\ (\mathsf{isSuc}\ n'))\ n) & : & \\
\qquad\qquad \mathsf{IMaybe}_A\ (\mathsf{isSuc}\ n) & & \\
\quad \alpha_{\mathsf{ihd}}\qquad\qquad\qquad\qquad\qquad ’\mathsf{nil} & \mapsto & \{?\} \\
\quad \alpha_{\mathsf{ihd}}\qquad\qquad\qquad\quad (’\mathsf{cons}\ a\ xs) & \mapsto & \{?\}
\end{array}$$

Once again, it is beyond the scope of this paper to formalise the elaboration process from the high-level notation to the low-level type theory. The reader will convince himself that the high-level notation contains all the information necessary to conduct this task. We shall now freely use the high-level syntax, with the understanding that it builds a low-level term that type-checks.

Example 10 (Transporting the recursion pattern of $\_ < \_$)**.** To implement ilookup, we use lift-case to transport the case analysis on n and lift-ind to transport the induction over m. In a high-level notation, this interaction results in:

$$\begin{array}{llllll}
\mathsf{ilookup}\ \ :\ \ \mathsf{Patch}\ \mathsf{type}{<}\ \mathsf{typeLookup}\ \_ < \_ & & & & & \\
\mathsf{ilookup} & m & m^m & n & vs & \overset{\mathsf{lift}}{\Leftarrow}\ \mathsf{lift\text{-}case} \\
\quad \mathsf{ilookup} & m & m^m & 0 & \mathsf{nil} & \{?\} \\
\quad \mathsf{ilookup} & m & m^m & (\mathsf{suc}\ n) & (\mathsf{cons}\ a\ vs) & \overset{\mathsf{lift}}{\Leftarrow}\ \mathsf{lift\text{-}ind} \\
\qquad \mathsf{ilookup} & 0 & 0 & 0 & \mathsf{nil} & \{?\} \\
\qquad \mathsf{ilookup} & (\mathsf{suc}\ m) & (\mathsf{suc}\ m^m) & 0 & \mathsf{nil} & \{?\}
\end{array}$$

5.2 Transporting constructors

Just as the recursive structure, the returned values often simply mirror the original definition: we are 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 the constructor, it should fill in the parts that are already determined by the original constructor and ask only for the missing information, i.e. the data freshly introduced by the ornament.

Remember that, when implementing the coherent lifting, we are working on the reornaments of the lifting type. Hence, when returning a constructor-headed value, we are building an inhabitant of a reornament. When defining reornaments in Section 3.2.2, we have shown that, thanks to deletion ornaments, a reornament can be decomposed in two components:

- first, the extension that contains all the extra information introduced by the ornament ;
- second, the recursive structure of the refined datatype, which defines the type of the arguments of the constructor

And no additional information is required: all the information provided by indexing with the unornamented datatype is optimally used in the definition of the reornament. Thus, there is absolutely no duplication of information.

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. In terms of implementation, this is as simple as:

$$\begin{array}{lll}
\mathsf{lift\text{-}constructor}\ (xs : \llbracket D\ (re\ j) \rrbracket\ \mu\, D) & & \\
\qquad (e : \mathsf{Extension}\ (O\ j)\ xs) & & \textit{– coherent extension} \\
\qquad (a : \llbracket \llbracket \mathsf{Structure}\ O\ xs\ e \rrbracket_{\mathsf{orn}} \rrbracket\ (\mu\, D^{O})) & & \textit{– arguments} \\
\qquad (t^{++} : \mathsf{Patch}\ T\ t\ T^{+}) & & \\
\qquad : \mathsf{Patch}\ (\mu\{ D\ (re\ j)\ \}\!\times\ T) & & \\
\qquad\qquad (\mathsf{in}\ xs, t) & & \\
\qquad\qquad (\mu^{+}\{ O\ j\ \}\!\times\ T^{+}) & & \\
\mathsf{lift\text{-}constructor}\ xs\ e\ a\ t^{++}\ \mapsto\ \left(\mathsf{in}\ (e, a), t^{++}\right) & &
\end{array}$$

Example 11 (Transporting the constructors of isSuc)**.** 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. 6(c). The interactive system would then respond by generating the code of Fig. 6(d). The $\mathbb{1}$ goals are trivially solved, probably automatically by the system. The only information the user has to provide is a value of type A returned by the just constructor.

Example 12 (Transporting the constructors of $\_ < \_$)**.** In the implementation of ilookup, we want to lift the returned true and false to the Maybe ornament. In a high-level notation, this would be represented as follows:

$$\begin{array}{llllll}
\multicolumn{6}{l}{\mathsf{ilookup} : \mathsf{Patch}\ \mathsf{type}{<}\ \mathsf{typeLookup}\ \_ < \_} \\
\mathsf{ilookup} & m & m^{m} & n & vs & \overset{\mathsf{lift}}{\Leftarrow}\ \mathsf{lift\text{-}case} \\
\ \mathsf{ilookup} & m & m^{m} & \mathsf{0} & \mathsf{nil} & \overset{\mathsf{lift}}{\mapsto}\ \mathsf{nothing}\ {*}[{*}] \\
\ \mathsf{ilookup} & m & m^{m} & (\mathsf{suc}\ n) & (\mathsf{cons}\ a\ vs) & \overset{\mathsf{lift}}{\Leftarrow}\ \mathsf{lift\text{-}ind} \\
\quad \mathsf{ilookup} & \mathsf{0} & \mathsf{0} & (\mathsf{suc}\ n) & (\mathsf{cons}\ a\ vs) & \overset{\mathsf{lift}}{\mapsto}\ \mathsf{just}\ \boxed{\{?:A\}}\ [{*}] \\
\quad \mathsf{ilookup} & (\mathsf{suc}\ m) & (\mathsf{suc}\ m^{m}) & (\mathsf{suc}\ n) & (\mathsf{cons}\ a\ vs) & \boxed{\{?\}}
\end{array}$$

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

6. Related work

Our work is an extension of the work of McBride [2012] 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, therefore, deletion of duplicated information was impossible. A corollary of this was that reornaments contained a lot of duplication, hence making the lifting of value from ornamented to reornamented datatype extremely tedious.

Our presentation of algebraic ornament has been greatly improved by the categorical model developed by Atkey et al. [2011]: 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 folds is a trivial specialisation of induction. Also, the characterisation of the fix-point of an algebraic ornament as op-reindexing of the fold is a key ingredient to understanding index-level computations and assimilate them at the term level.

In their work on realisability and parametricity for Pure Type Systems, Bernardy and Lasson [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 contribution and their 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 realisability predicate. The strength of the realisability interpretation is that it is naturally defined on functions: while McBride [2012] and Atkey et al. [2011] only consider ornaments on datatypes, their work is the first, to our knowledge, to capture a general notion of functions realising – i.e. ornamenting – other functions.

Following the steps of Bernardy, Ko and Gibbons [2011] adapted the realisability interpretation to McBride's universe of datatypes and explored the other direction of the Patch equivalence, 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 equivalence 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 functions 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 realisability predicate constructions of Bernardy and Lasson [2011], Ko and Gibbons [2011].

7. Conclusion

In this paper, we have developed the notion of functional ornament and shown how one can achieve code reuse by transporting functions along a functional ornament. To this end, we have adapted McBride's ornaments to our universe of datatypes [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 [Brady et al. 2004] over inductive families. In particular, this gave us a simpler implementation of reornaments.

We then generalised 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.

Having functional ornaments, this raises the question of transporting a function to its ornamented version in such way that the coherence property holds. Instead of asking our 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 offered by the environment: 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 equivalence 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 two clever 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 element 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. Besides, we have developed some appealing automation to assist the implementation of functional ornaments, without any proving required, hence making this approach even more accessible.

7.1 Future work

Whilst we have deliberately chosen a simple universe of functions, we plan to extend it in various directions. Adding type dependency (Π- and Σ-types) but also non inductive sets 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 Vec_A and IMaybe_A as parametric so that in the $\mathsf{cons}\, a$ case, the a could automatically be carried over.

The universe of functional ornaments could be extended as well, especially once the universe of functions 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 fixed points in this paper, we also want to generalise our universe with greatest fixed points and the lifting of co-inductive definitions.

Further, our framework relies crucially on the duality between a reornament and its ornament presentation subject to a proof. We cross this isomorphism in both directions when we project the lifting from the coherent lifting. In practice, this involves a traversal of each of the input datatypes and a traversal of each of the output datatypes. However, computationally, these traversal are identities: the only purpose of these terms is at the logical level, for the type-checker to fix the types. We are looking at transforming our library of clever constructor into a proper domain-specific language (DSL). This way, implementing a coherent lifting would consists in working in a DSL for which an optimising compiler could compute away the computationally irrelevant operations.

Finally, much work remains to be done on the front of usability: for convenience, we have presented some informal notations for datatypes, their ornaments and an extension of Epigram programming facility with liftings. A formal treatment of these syntaxes and of their elaboration to the low-level type theory is underway: we are confident that a sufficiently abstract semantics can be given to these syntaxes by giving a relational specification of the elaboration process, in the style of Harper and Stone [2000] for Standard ML.

Acknowledgements We owe many thanks to the anonymous reviewer, their comments having significantly improved this paper. 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

R. Atkey, P. Johann, and N. Ghani. When is a type refinement an inductive type? In *FOSSACS*, volume 6604 of *Lecture Notes in Computer Science*, pages 72–87. Springer, 2011.

J.-P. Bernardy and M. Lasson. Realizability and parametricity in pure type systems. In *FOSSACS*, volume 6604 of *Lecture Notes in Computer Science*, pages 108–122. Springer, 2011.

R. S. Bird and O. de Moor. *Algebra of programming*. Prentice Hall, 1997.

E. Brady, C. McBride, and J. McKinna. Inductive families need not store their indices. In *Types for Proofs and Programs*, pages 115–129. 2004.

J. Chapman, P.-E. Dagand, C. McBride, and P. Morris. The gentle art of levitation. *SIGPLAN Not.*, 45:3–14, September 2010.

T. Coquand. Pattern matching with dependent types. In *Types for Proofs and Programs*, 1992.

P. Dybjer. Inductive families. *Formal Asp. Comput.*, 6(4):440–465, 1994.

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

C. Fumex, N. Ghani, and P. Johann. Indexed induction and coinduction, fibrationally. In *CALCO*, pages 176–191, 2011.

R. Harper and C. Stone. A Type-Theoretic interpretation of standard ML. In *Proof, Language, and Interaction: essays in honour of Robin Milner*, 2000.

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

C. McBride. Ornamental algebras, algebraic ornaments. *Journal of Functional Programming, to appear*, 2012.

C. McBride and J. McKinna. The view from the left. *J. Funct. Program.*, 14(1):69–111, 2004.

C. McBride, H. Goguen, and J. McKinna. A few constructions on constructors. In *TYPES*, pages 186–200, 2004.

P. Morris, T. Altenkirch, and N. Ghani. A universe of strictly positive families. *Int. J. Found. Comput. Sci.*, 20(1):83–107, 2009.

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

P.-Y. Strub. Coq modulo theory. In *CSL*, pages 529–543, 2010.

N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan, and J. Yang. Secure distributed programming with value-dependent types. In *ICFP*, pages 266–278. ACM, 2011.

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

H. Xi, C. Chen, and G. Chen. Guarded recursive datatype constructors. In *POPL*, 2003.