Modularising Inductive Families

Hsiang-Shang Ko   Jeremy Gibbons
Department of Computer Science, University of Oxford
{Hsiang-Shang.Ko, Jeremy.Gibbons}@cs.ox.ac.uk

Abstract

Dependently typed programmers are encouraged to use inductive families to integrate constraints with data construction. Different constraints are used in different contexts, leading to different versions of datatypes for the same data structure. Modular implementation of common operations for these structurally similar datatypes has been a longstanding problem. We propose a datatype-generic solution based on McBride's datatype ornaments [11], exploiting an isomorphism whose interpretation borrows ideas from realisability. Relevant properties of the operations are separately proven for each constraint, and after the programmer selects several constraints to impose on a basic datatype and synthesises an inductive family incorporating those constraints, the operations can be routinely upgraded to work with the synthesised inductive family.

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

General Terms Design, Languages, Theory

Keywords Dependently Typed Programming, Inductive Families, Datatype-Generic Programming

1. Introduction

Dependently typed programmers are encouraged to use *inductive families*, i.e., datatypes with fancy indices, to integrate various constraints with data construction. Correctness proofs are built into and manipulated simultaneously with the data, and in ideal cases correct programs can be written in blissful ignorance of the proofs. We might characterise this approach as *internalist*, suggesting that data constraints are internalised. In contrast, the more traditional approach which favours using only basic datatypes and expressing constraints through separate predicates on those datatypes might be described as *externalist*.

The internalist approach easily leads to an explosion in differently indexed versions of the same data structure. For example, as well as ordinary lists, in different contexts we may need vectors (lists indexed with their length), sorted lists, or sorted vectors, ending up with four slightly different but structurally similar datatypes. The problem, then, is how the common operations are implemented for these different versions of the datatype. Current practice is to completely reimplement the operations for each version, causing serious code duplication and dreadful reusability. The externalist approach, in contrast, responds to this problem very well. We would have only one basic list type, with one predicate stating that a list has a certain length and another predicate asserting that a list is sorted. The list type is upgraded to the vector type, the sorted list type, or the sorted vector type by simply pairing the list type with the sortedness predicate, the length predicate, or the pointwise conjunction of the two predicates. The common operations are implemented for ordinary lists only, and their properties regarding ordering or length are separately proven and invoked when needed. Can we somehow introduce this beneficial compositionality to internalism as well? Yes, we can! There is an isomorphism between externalist and internalist datatypes to be exploited.

To illustrate, let us go through a small case study about upgrading a function on natural numbers. The internalists use the following datatype to characterise the *finite numbers*, which are natural numbers bounded above by a certain number.

$$
\begin{array}{l}
\mathbf{data}\ \mathsf{Fin} : \mathsf{Nat} \to \mathsf{Set}\ \mathbf{where} \\
\quad \mathsf{fzero}\ :\ \{m : \mathsf{Nat}\} \to \mathsf{Fin}\ (\mathsf{suc}\ m) \\
\quad \mathsf{fsuc}\ \ :\ \{m : \mathsf{Nat}\} \to \mathsf{Fin}\ m \to \mathsf{Fin}\ (\mathsf{suc}\ m)
\end{array}
$$

We can be explicit about how we regard finite numbers as natural numbers by defining a forgetful map.

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\mathit{forget}_{\mathsf{F}} : \forall\ \{m\} \to \mathsf{Fin}\ m \to \mathsf{Nat}} \\
\mathit{forget}_{\mathsf{F}}\ \mathsf{fzero} & = & \mathsf{zero} \\
\mathit{forget}_{\mathsf{F}}\ (\mathsf{fsuc}\ i) & = & \mathsf{suc}\ (\mathit{forget}_{\mathsf{F}}\ i)
\end{array}
$$

To represent the same type, externalists would first define a greater-than relation for natural numbers,

$$
\begin{array}{l}
\mathbf{data}\ \_{>}\_ : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Set}\ \mathbf{where} \\
\quad \mathsf{base}\ :\ \{m : \mathsf{Nat}\} \to \mathsf{suc}\ m > \mathsf{zero} \\
\quad \mathsf{step}\ :\ \{m\ n : \mathsf{Nat}\} \to m > n \to \mathsf{suc}\ m > \mathsf{suc}\ n\ ,
\end{array}
$$

and then use the dependent pair type $\Sigma\ \mathsf{Nat}\ (\lambda n \mapsto m > n)$, an object of which is a natural number n paired with a proof that $m > n$. We have an isomorphism between the two types,

$$\mathsf{Fin}\ m\ \cong\ \Sigma\ \mathsf{Nat}\ (\lambda n \mapsto m > n)\ ,$$

witnessed by

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\Re_{\mathsf{F}} : \forall\ \{m\} \to (i : \mathsf{Fin}\ m) \to m > \mathit{forget}_{\mathsf{F}}\ i} \\
\Re_{\mathsf{F}}\ \ \mathsf{fzero} & = & \mathsf{base} \\
\Re_{\mathsf{F}}\ \ (\mathsf{fsuc}\ i) & = & \mathsf{step}\ (\Re_{\mathsf{F}}\ i)
\end{array}
$$

and

$$
\begin{array}{llcl}
\multicolumn{4}{l}{\Re_{\mathsf{F}}^{-1} : \forall\ \{m\} \to (n : \mathsf{Nat}) \to m > n \to \mathsf{Fin}\ m} \\
\Re_{\mathsf{F}}^{-1}\ \ .\mathsf{zero} & \mathsf{base} & = & \mathsf{fzero} \\
\Re_{\mathsf{F}}^{-1}\ \ .(\mathsf{suc}\ \_) & (\mathsf{step}\ gt) & = & \mathsf{fsuc}\ (\Re_{\mathsf{F}}^{-1}\ \_\ gt)\ .
\end{array}
$$

Now suppose that we have some function $f' : \mathsf{Nat} \to \mathsf{Nat}$, and additionally that we can prove externally that f' preserves upper bounds (in other words, is non-increasing):

$$f'\textit{-bound} : \forall\ \{m\ n\} \to m > n \to m > f'\ n\ .$$

WGP'11, September 18, 2011, Tokyo, Japan.

Then we can upgrade f' to work with finite numbers by exploiting the isomorphism:

$$\begin{aligned} & f_{\mathsf{F}} : \forall\ \{m\} \to \mathsf{Fin}\ m \to \mathsf{Fin}\ m \\ & f_{\mathsf{F}}\ i = \mathfrak{R}_{\mathsf{F}}^{-1}\ (f'\ (\mathit{forget}_{\mathsf{F}}\ i))\ (f'\text{-}\mathit{bound}\ (\mathfrak{R}_{\mathsf{F}}\ i))\ . \end{aligned}$$

The input finite number $i : \mathsf{Fin}\ m$ is split into the underlying natural number $\mathit{forget}_{\mathsf{F}}\ i : \mathsf{Nat}$ and a corresponding proof $\mathfrak{R}_{\mathsf{F}}\ i : m > \mathit{forget}\ i$. The natural number is then processed by f' and the proof by $f'\text{-}\mathit{bound}$, before the results are integrated back into a finite number by way of $\mathfrak{R}_{\mathsf{F}}^{-1}$.

Further suppose that we need parity information about f'. The externalists would define a function to compute the parity of a natural number,

$$\begin{aligned} & \mathit{parity} : \mathsf{Nat} \to \mathsf{Bool} \\ & \mathit{parity}\ \mathsf{zero} \quad\ \ = \mathsf{false} \\ & \mathit{parity}\ (\mathsf{suc}\ n) = \mathit{not}\ (\mathit{parity}\ n)\ , \end{aligned}$$

and use the type $\Sigma\ \mathsf{Nat}\ (\lambda n \mapsto \mathit{parity}\ n \equiv b)$ (where $\_\equiv\_$ is the propositional equality type) for those natural numbers of parity b. The internalists would define a new datatype

$$\begin{aligned} & \mathbf{data}\ \mathsf{PNat} : \mathsf{Bool} \to \mathsf{Set}\ \mathbf{where} \\ & \quad \mathsf{pzero} : \mathsf{PNat}\ \mathsf{false} \\ & \quad \mathsf{psuc} \ \ : \{b : \mathsf{Bool}\} \to \mathsf{PNat}\ b \to \mathsf{PNat}\ (\mathit{not}\ b)\ , \end{aligned}$$

and use $\mathsf{PNat}\ b$ for the same set of natural numbers. Assume f' preserves parity, i.e., we can prove

$$f'\text{-}\mathit{parity} : \forall\ \{n\ b\} \to \mathit{parity}\ n \equiv b \to \mathit{parity}\ (f'\ n) \equiv b\ .$$

Following the same recipe, by exploiting the isomorphism

$$\mathsf{PNat}\ b\ \cong\ \Sigma\ \mathsf{Nat}\ (\lambda n \mapsto \mathit{parity}\ n \equiv b)$$

witnessed by

$$\begin{aligned} & \mathit{forget}_{\mathsf{P}} : \forall\ \{b\} \to \mathsf{PNat}\ b \to \mathsf{Nat} \\ & \mathit{forget}_{\mathsf{P}}\ \mathsf{pzero} \quad\ \ = \mathsf{zero} \\ & \mathit{forget}_{\mathsf{P}}\ (\mathsf{psuc}\ j) = \mathsf{suc}\ (\mathit{forget}_{\mathsf{P}}\ j) \end{aligned}$$

$$\begin{aligned} & \mathfrak{R}_{\mathsf{P}} : \forall\ \{b\} \to (j : \mathsf{PNat}\ b) \to \mathit{parity}\ (\mathit{forget}_{\mathsf{P}}\ j) \equiv b \\ & \mathfrak{R}_{\mathsf{P}}\ \mathsf{pzero} \qquad\qquad\qquad\quad\ = \mathsf{refl} \\ & \mathfrak{R}_{\mathsf{P}}\ (\mathsf{psuc}\ j)\ \mathbf{rewrite}\ \mathfrak{R}_{\mathsf{P}}\ j = \mathsf{refl} \end{aligned}$$

and

$$\begin{aligned} & \mathfrak{R}_{\mathsf{P}}^{-1} : \forall\ \{b\} \to (n : \mathsf{Nat}) \to \mathit{parity}\ n \equiv b \to \mathsf{PNat}\ b \\ & \mathfrak{R}_{\mathsf{P}}^{-1}\ \mathsf{zero} \quad\ \ \mathsf{refl} = \mathsf{pzero} \\ & \mathfrak{R}_{\mathsf{P}}^{-1}\ (\mathsf{suc}\ n)\ \mathsf{refl} = \mathsf{psuc}\ (\mathfrak{R}_{\mathsf{P}}^{-1}\ n\ \mathsf{refl})\ , \end{aligned}$$

we can again upgrade f' to work with PNat:

$$\begin{aligned} & f_{\mathsf{P}} : \forall\ \{b\} \to \mathsf{PNat}\ b \to \mathsf{PNat}\ b \\ & f_{\mathsf{P}}\ j = \mathfrak{R}_{\mathsf{P}}^{-1}\ (f'\ (\mathit{forget}_{\mathsf{P}}\ j))\ (f'\text{-}\mathit{parity}\ (\mathfrak{R}_{\mathsf{P}}\ j))\ . \end{aligned}$$

Finally, consider finite numbers with parity information. The externalists would simply put the two predicates together and get the type $\Sigma\ \mathsf{Nat}\ (\lambda n \mapsto (m > n) \times (\mathit{parity}\ n \equiv b))$ for the natural numbers bounded above by m and of parity b. The internalists would define yet another datatype

$$\begin{aligned} & \mathbf{data}\ \mathsf{PFin} : \mathsf{Nat} \to \mathsf{Bool} \to \mathsf{Set}\ \mathbf{where} \\ & \quad \mathsf{pfzero} : \forall\ \{m\} \to \mathsf{PFin}\ (\mathsf{suc}\ m)\ \mathsf{false} \\ & \quad \mathsf{pfsuc} \ \ : \forall\ \{m\ b\} \to \mathsf{PFin}\ m\ b \to \mathsf{PFin}\ (\mathsf{suc}\ m)\ (\mathit{not}\ b) \end{aligned}$$

and use $\mathsf{PFin}\ m\ b$ for the same set of natural numbers. We still have an isomorphism

$$\mathsf{PFin}\ m\ b\ \cong\ \Sigma\ \mathsf{Nat}\ (\lambda n \mapsto (m > n) \times (\mathit{parity}\ n \equiv b))$$

witnessed by

$$\begin{aligned} & \mathit{forget}_{\mathsf{PF}} : \forall\ \{m\ b\} \to \mathsf{PFin}\ m\ b \to \mathsf{Nat} \\ & \mathit{forget}_{\mathsf{PF}}\ \mathsf{pfzero} \quad\ \ = \mathsf{zero} \\ & \mathit{forget}_{\mathsf{PF}}\ (\mathsf{pfsuc}\ k) = \mathsf{suc}\ (\mathit{forget}_{\mathsf{PF}}\ k) \end{aligned}$$

$$\begin{aligned} & \mathfrak{R}_{\mathsf{PF}}\text{-}l : \forall\ \{m\ b\} \to (k : \mathsf{PFin}\ m\ b) \to m > \mathit{forget}_{\mathsf{PF}}\ k \\ & \mathfrak{R}_{\mathsf{PF}}\text{-}l\ \mathsf{pfzero} \quad\ \ = \mathsf{base} \\ & \mathfrak{R}_{\mathsf{PF}}\text{-}l\ (\mathsf{pfsuc}\ k) = \mathsf{step}\ (\mathfrak{R}_{\mathsf{PF}}\text{-}l\ k) \end{aligned}$$

$$\begin{aligned} & \mathfrak{R}_{\mathsf{PF}}\text{-}r : \forall\ \{m\ b\} \to (k : \mathsf{PFin}\ m\ b) \to \mathit{parity}\ (\mathit{forget}_{\mathsf{PF}}\ k) \equiv b \\ & \mathfrak{R}_{\mathsf{PF}}\text{-}r\ \mathsf{pfzero} \qquad\qquad\qquad\qquad\quad\ = \mathsf{refl} \\ & \mathfrak{R}_{\mathsf{PF}}\text{-}r\ (\mathsf{pfsuc}\ k)\ \mathbf{rewrite}\ \mathfrak{R}_{\mathsf{PF}}\text{-}r\ k = \mathsf{refl} \end{aligned}$$

and

$$\begin{aligned} & \mathfrak{R}_{\mathsf{PF}}^{-1} : \forall\ \{m\ b\} \to (n : \mathsf{Nat}) \to m > n \to \mathit{parity}\ n \equiv b \to \mathsf{PFin}\ m\ b \\ & \mathfrak{R}_{\mathsf{PF}}^{-1}\ .\mathsf{zero} \quad\ \ \mathsf{base} \quad\ \ \mathsf{refl} = \mathsf{pfzero} \\ & \mathfrak{R}_{\mathsf{PF}}^{-1}\ .(\mathsf{suc}\ \_)\ (\mathsf{step}\ \mathit{gt})\ \mathsf{refl} = \mathsf{pfsuc}\ (\mathfrak{R}_{\mathsf{PF}}^{-1}\ \_\ \mathit{gt}\ \mathsf{refl})\ , \end{aligned}$$

and the isomorphism can again be used to upgrade f' to work with PFin, but this time the proof part *reuses* the existing proofs $f'\text{-}\mathit{bound}$ and $f'\text{-}\mathit{parity}$:

$$\begin{aligned} & f_{\mathsf{PF}} : \forall\ \{m\ b\} \to \mathsf{PFin}\ m\ b \to \mathsf{PFin}\ m\ b \\ & f_{\mathsf{PF}}\ k = \mathfrak{R}_{\mathsf{PF}}^{-1}\ (f'\ (\mathit{forget}_{\mathsf{PF}}\ k)) \\ & \qquad\qquad\quad (f'\text{-}\mathit{bound}\ (\mathfrak{R}_{\mathsf{PF}}\text{-}l\ k))\ (f'\text{-}\mathit{parity}\ (\mathfrak{R}_{\mathsf{PF}}\text{-}r\ k))\ . \end{aligned}$$

Had we implemented f_{F} and f_{P} directly instead of exploiting the isomorphisms, it would have been much less straightforward to synthesise f_{PF} from them. It is thanks to the isomorphism maps $\mathfrak{R}$ and $\mathfrak{R}^{-1}$ that we can routinely synthesise f_{F} and f_{P} from corresponding externalist proofs, and — more interestingly — that we can develop f_{PF} modularly, reusing those externalist proofs. The reusability problem is thus reduced to writing the isomorphisms, and the good news is that the isomorphisms can be synthesised *datatype-generically*. Acquiring the power of datatype-generic programming, we can even synthesise PFin from the ingredients used to make Fin and PNat out of Nat, revealing the same compositional structure of the internalist types corresponding to that of their externalist brethren.

Outline of this paper. Our work is heavily based on McBride's datatype *ornaments* [11], which provide a datatype-generic language in which to talk about the relationship among structurally similar datatypes. McBride's work is summarised in Section 2. An ornament describes how to upgrade a basic datatype to a fancier one, often embedding some constraints into data construction. Then an interpretation based on realisability is given in Section 3: Given an ornament, objects of the basic datatype are considered as incomplete proofs of the fancier datatype, and the information needed to restore a complete proof from an incomplete one is stated by the *realisability predicate* induced by the ornament. With the interpretation, we are enabled to think about *composition of ornaments*, and thus indexed datatypes with multiple constraints, in terms of pointwise conjunction of realisability predicates. As an initial experiment, in Section 4 we consider the special case where one of the two ornaments being composed is *algebraic*. We prove that the pointwise conjunction of the realisability predicates induced by the component ornaments is isomorphic to the realisability predicate induced by the composite ornament, and demonstrate how this helps to write functions on indexed datatypes incorporating multiple constraints in a modular style. Section 5 discusses how the interpretation connects internalism and externalism, and how we might exploit this connection to structure our libraries. Section 6 compares ours with previous work, and finally Section 7 presents some possible future directions. We have implemented our ideas in Agda [14], source available at `http://www.cs.ox.ac.uk/people/hsiang-shang.ko/OAOAOO/`.

2. A recapitulation of datatype ornaments

To state the realisability interpretation generically, first we need a *datatype-generic* framework for talking about the relationship between structurally similar datatypes. Central to datatype-generic programming is the idea that the structure of datatypes can be coded as first-class entities and thus become ordinary parameters to programs. The same idea is also found in Martin-Löf's Type Theory [10], in which a set of codes for datatypes is called a *universe* (à la Tarski), and there is a decoding function translating codes to actual types. Type theory being the foundation of dependently typed languages, universe construction can be implemented directly in such languages, so datatype-generic programming becomes just ordinary programming in the dependently typed world [1].

McBride's seminal work on datatype ornaments [11] is ideally suited to our purposes. What he did was to construct a universe in Agda, i.e., a datatype whose inhabitants are codes to be translated into actual types, with generic fold and induction for decoded types, and define another datatype whose inhabitants — called *ornaments* — explain how to "patch" a code to a richer one but retaining the basic structure. For example, a list is a Peano-style natural number whose successor nodes are decorated with elements, and a vector is a list whose type is indexed with its length. Ornaments are designed to encode these two kinds of addition of information: *decoration* (element insertion) and *refinement* (index upgrade). Consequently, induced by every ornament is a forgetful map erasing the added information from an object of the ornamented datatype and recovering an object of the raw datatype. For example, the forgetful map induced by the ornamentation from natural numbers to lists is just *length*, which discards the elements associated with the cons nodes. The forgetful map is a fold, and the algebra of the fold is called the *ornamental algebra*, as it is induced by an ornament. Conversely, every algebra induces an *algebraic ornament*, which provides a systematic way to index the type of an object with the result of the fold of the algebra applied to that object. The vector type is a typical example — it arises from the algebraic ornamentation of lists which indexes the type of a list with its length.

Datatype descriptions. Concretely, McBride used the datatype

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textbf{data}\ \mathsf{Desc}\ (I : \mathsf{Set}) : \mathsf{Set}_1\ \textbf{where}} \\
\quad \mathsf{say} & : & I \rightarrow \mathsf{Desc}\, I \\
\quad \sigma & : & (S : \mathsf{Set}) \rightarrow (S \rightarrow \mathsf{Desc}\, I) \rightarrow \mathsf{Desc}\, I \\
\quad \mathsf{ask\_{*}\_} & : & I \rightarrow \mathsf{Desc}\, I \rightarrow \mathsf{Desc}\, I
\end{array}
$$

as the universe. A term of type $\mathsf{Desc}\, I$ describes an inductive family of type $I \rightarrow \mathsf{Set}$ by specifying how its data are constructed: The first constructor $\mathsf{say}\ i$ marks the end of a description and delivers data at index i; the second constructor $\sigma\ S\ D$ inserts an element of type S on which the remaining description D may depend; the third constructor $\mathsf{ask}\ i * D$ recursively requests data at index i and then continues with D. For example, the code for the type of natural numbers is

$$
\begin{array}{lll}
\mathit{NatD} & : & \mathsf{Desc}\ \top \\
\mathit{NatD} & = & \sigma\ \mathsf{Bool}\ (\mathit{false}{\mapsto}\ \mathsf{say}\ \mathsf{tt} \\
 & & \qquad\qquad \mathit{true}{\mapsto}\ \ \mathsf{ask}\ \mathsf{tt} * \mathsf{say}\ \mathsf{tt})\ ,
\end{array}
$$

where $\top$ is a one-element type whose only constructor is tt, and $\mathit{false}{\mapsto}\_\mathit{true}{\mapsto}\_$ is a function imitating dependent case expressions,

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathit{false}{\mapsto}\_\mathit{true}{\mapsto}\_ : \{P : \mathsf{Bool} \rightarrow \mathsf{Set}_1\} \rightarrow} \\
\multicolumn{3}{l}{\qquad\qquad\qquad P\ \mathsf{false} \rightarrow P\ \mathsf{true} \rightarrow (b : \mathsf{Bool}) \rightarrow P\ b} \\
(\mathit{false}{\mapsto}\ p\ \mathit{true}{\mapsto}\ q)\ \mathsf{false} & = & p \\
(\mathit{false}{\mapsto}\ p\ \mathit{true}{\mapsto}\ q)\ \mathsf{true} & = & q\ .
\end{array}
$$

The description *NatD* describes exactly how to construct a Peano-style natural number: We choose one constructor out of two by giving a boolean value; if it is false, the construction is complete and the result is delivered at the trivial index tt; otherwise it is true, in which case we recursively ask for a natural number before delivering the result.

To translate a description of type $\mathsf{Desc}\, I$ to an actual type, first we decode it to an endofunctor on $I \rightarrow \mathsf{Set}$.

$$
\begin{array}{llll}
\multicolumn{4}{l}{[\![\_]\!] : \{I : \mathsf{Set}\} \rightarrow \mathsf{Desc}\, I \rightarrow (I \rightarrow \mathsf{Set}) \rightarrow I \rightarrow \mathsf{Set}} \\
[\![\mathsf{say}\ i]\!] & X\ i' & = & i \equiv i' \\
[\![\sigma\ S\ D]\!] & X\ i' & = & \Sigma\ S\ \lambda s \mapsto [\![D\ s]\!]\ i' \\
[\![\mathsf{ask}\ i * D]\!] & X\ i' & = & X\ i \times [\![D]\!]\ i'
\end{array}
$$

Then we can take the least fixed point of the decoded functor by the following native inductive datatype:

$$
\begin{array}{l}
\textbf{data}\ \mu\ \{I : \mathsf{Set}\}\ (D : \mathsf{Desc}\, I) : I \rightarrow \mathsf{Set}\ \textbf{where} \\
\quad \langle\_\rangle\ :\ \forall\ \{i\} \rightarrow [\![D]\!]\ (\mu\ D)\ i \rightarrow \mu\ D\ i\ .
\end{array}
$$

If we introduce a notation for functions on $I \rightarrow \mathsf{Set}$,

$$
\begin{array}{l}
\_{\Rightarrow}\_ : \{I : \mathsf{Set}\} \rightarrow (I \rightarrow \mathsf{Set}) \rightarrow (I \rightarrow \mathsf{Set}) \rightarrow \mathsf{Set} \\
X \Rightarrow Y\ =\ \forall\ \{i\} \rightarrow X\ i \rightarrow Y\ i\ ,
\end{array}
$$

we see that $\langle\_\rangle : [\![D]\!]\ (\mu\ D) \Rightarrow \mu\ D$ has the familiar form of an algebra for the functor $[\![D]\!]$, which is in fact the initial algebra. So the type of natural numbers, *Nat*, is obtained by decoding *NatD*.[1]

$$
\begin{array}{lll}
\mathit{Nat} & : & \mathsf{Set} \\
\mathit{Nat} & = & \mu\ \mathit{NatD}\ \mathsf{tt}
\end{array}
$$

The decoded type *Nat* being a native inductive type, we can define functions on such natural numbers by pattern matching, albeit a bit cryptically, like

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathit{pred} : \mathit{Nat} \rightarrow \mathit{Nat}} \\
\mathit{pred}\ \langle \mathsf{false}, \mathsf{refl} \rangle & = & \mathit{zero} \\
\mathit{pred}\ \langle \mathsf{true}, n, \mathsf{refl} \rangle & = & n
\end{array}
$$

where $\mathit{zero} = \langle \mathsf{false}, \mathsf{refl} \rangle : \mathit{Nat}$. But later when we need to define operations and state properties for all the types encoded by the universe, it is necessary to have a generic fold operator parametrised by the codes:

$$
\mathit{fold} : \{I\ X : \mathsf{Set}\}\ \{D : \mathsf{Desc}\, I\} \rightarrow ([\![D]\!]\ X \Rightarrow X) \rightarrow \mu\ D \Rightarrow X\ .
$$

There is also a generic induction operator, which is more powerful and subsumes generic fold, but fold is much easier to use when the full power of induction is not required. The implementation details of the two operators are not essential to our exposition and hence are omitted from this paper.

Ornaments. Next we define the ornaments. An ornament is a "relative" description which is written with respect to another description and marks changes relative to the latter. One of the two kinds of information expressed in ornaments is *refinement*: how to promote the I-indices in an I-description to J-indices with respect to an index erasure function $e : J \rightarrow I$ — the new J-indices appearing in an ornament must be erasable by e to the original I-indices. The following inverse-image datatype helps to enforce this requirement:

$$
\begin{array}{l}
\textbf{data}\ \_^{-1}\_\ \{I\ J : \mathsf{Set}\}\ (e : J \rightarrow I) : I \rightarrow \mathsf{Set}\ \textbf{where} \\
\quad \mathsf{ok}\ :\ (j : J) \rightarrow e^{-1}\ (e\ j)\ .
\end{array}
$$

If we have a value of type $e^{-1}\ i$, then we are guaranteed to be able to extract from it a value j such that $e\ j$ is definitionally equal to i. The ornaments are then defined as a datatype indexed by descriptions of type $\mathsf{Desc}\, I$. Its first three constructors mirror those of $\mathsf{Desc}\, I$, refining I-indices to J-indices, while the fourth constructor Δ provides the second kind of ornamental information

---

[1] A typographical convention: Type and data constructors introduced by native data declarations are typeset in $\mathsf{sans\ serif}$, while other terms like functions, variables, etc. are typeset in *italics*. So the Nat we saw in Section 1 is a native datatype, whereas *Nat* here is a decoded datatype.

on *decoration*, signalling insertion of a new element on which the trailing ornament may depend.

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathbf{data}\ \mathsf{Orn}\ \{I : \mathsf{Set}\}\ (J : \mathsf{Set})\ (e : J \to I) : \mathsf{Desc}\ I \to \mathsf{Set}_1\ \mathbf{where}} \\
\quad \mathsf{say} & : & \{i : I\} \to e^{-1}\ i \to \mathsf{Orn}\ J\ e\ (\mathsf{say}\ i) \\
\quad \sigma & : & (S : \mathsf{Set})\ \{D : S \to \mathsf{Desc}\ I\} \to \\
 & & \quad (\forall\ s \to \mathsf{Orn}\ J\ e\ (D\ s)) \to \mathsf{Orn}\ J\ e\ (\sigma\ S\ D) \\
\quad \mathsf{ask\_{*}\_} & : & \{i : I\} \to e^{-1}\ i \to \\
 & & \quad \forall\ \{D\} \to \mathsf{Orn}\ J\ e\ D \to \mathsf{Orn}\ J\ e\ (\mathsf{ask}\ i * D) \\
\quad \Delta & : & (S : \mathsf{Set})\ \{D : \mathsf{Desc}\ I\} \to \\
 & & \quad (S \to \mathsf{Orn}\ J\ e\ D) \to \mathsf{Orn}\ J\ e\ D
\end{array}$$

For example, the ornament

$$\begin{array}{l}
\mathit{ListO} : \mathsf{Set} \to \mathsf{Orn}\ \top\ \mathit{id}\ \mathit{NatD} \\
\mathit{ListO}\ A = \\
\quad \sigma\ \mathsf{Bool}\ (\mathit{false} \mapsto \mathsf{say}\ (\mathsf{ok}\ \mathsf{tt}) \\
\qquad\qquad\quad \mathit{true} \mapsto\ \Delta\ A\ \lambda\_ \mapsto \mathsf{ask}\ (\mathsf{ok}\ \mathsf{tt}) * \mathsf{say}\ (\mathsf{ok}\ \mathsf{tt}))
\end{array}$$

describes the ornamentation from natural numbers to lists. It looks very much like a description except that the indices are wrapped in ok and the Δ should have been σ. We get these differences because *ListO A* is a description *relative to NatD*: The new indices have to prove that they respect *id* by wrapping themselves in ok and Δ is used in place of σ to indicate that the element is not originally in *NatD*. Generically, an ornament of type $\mathsf{Orn}\ J\ e\ D$ can of course be decoded into an "absolute" description of type $\mathsf{Desc}\ J$ by unwrapping the J-indices and translating Δ to σ:

$$\begin{array}{lll}
\multicolumn{3}{l}{\lfloor\_\rfloor : \forall\ \{I\ J\ e\}\ \{D : \mathsf{Desc}\ I\} \to \mathsf{Orn}\ J\ e\ D \to \mathsf{Desc}\ J} \\
\lfloor \mathsf{say}\ (\mathsf{ok}\ j) \rfloor & = & \mathsf{say}\ j \\
\lfloor \sigma\ S\ O \rfloor & = & \sigma\ S\ \lambda s \mapsto \lfloor O\ s \rfloor \\
\lfloor \mathsf{ask}\ (\mathsf{ok}\ j) * O \rfloor & = & \mathsf{ask}\ j * \lfloor O \rfloor \\
\lfloor \Delta\ S\ O \rfloor & = & \sigma\ S\ \lambda s \mapsto \lfloor O\ s \rfloor\ .
\end{array}$$

So the decoded description $\lfloor \mathit{ListO}\ A \rfloor$ expands to

$$\begin{array}{l}
\sigma\ \mathsf{Bool}\ (\mathit{false} \mapsto \mathsf{say}\ \mathsf{tt} \\
\qquad\qquad\quad \mathit{true} \mapsto\ \sigma\ A\ \lambda\_ \mapsto \mathsf{ask}\ \mathsf{tt} * \mathsf{say}\ \mathsf{tt})
\end{array}$$

as expected, which can then be decoded to the list type $\mathit{List}\ A = \mu\ \lfloor \mathit{ListO}\ A \rfloor\ \mathsf{tt}$.

An ornament $O : \mathsf{Orn}\ J\ e\ D$ gives rise to an *ornamental algebra* $\mathit{ornAlg}\ O : [\![\lfloor O \rfloor]\!]\ (\mu\ D \cdot e) \Rightarrow (\mu\ D \cdot e)$ which erases elements added by Δ and demotes the indices. (The $\_\cdot\_$ operator is function composition.) First we define a polymorphic restructuring map erasing information added by Δ,

$$\begin{array}{llcl}
\multicolumn{4}{l}{\mathit{erase} : \forall\ \{I\ J\ e\}\ \{D : \mathsf{Desc}\ I\}\ (O : \mathsf{Orn}\ J\ e\ D)\ \{X\} \to} \\
\multicolumn{4}{l}{\qquad [\![\lfloor O \rfloor]\!]\ (X \cdot e) \Rightarrow [\![D]\!]\ X \cdot e} \\
\mathit{erase}\ (\mathsf{say}\ (\mathsf{ok}\ j)) & \mathsf{refl} & = & \mathsf{refl} \\
\mathit{erase}\ (\sigma\ S\ O) & (s,\ \mathit{ds}) & = & s,\ \mathit{erase}\ (O\ s)\ \mathit{ds} \\
\mathit{erase}\ (\mathsf{ask}\ (\mathsf{ok}\ j) * O) & (d, \mathit{ds}) & = & d,\ \mathit{erase}\ O\ \mathit{ds} \\
\mathit{erase}\ (\Delta\ S\ O) & (s,\ \mathit{ds}) & = & \mathit{erase}\ (O\ s)\ \mathit{ds}\ ,
\end{array}$$

and then the ornamental algebra is defined by

$$\begin{array}{l}
\mathit{ornAlg} : \forall\ \{I\ J\ e\}\ \{D : \mathsf{Desc}\ I\}\ (O : \mathsf{Orn}\ J\ e\ D) \to \\
\qquad [\![\lfloor O \rfloor]\!]\ (\mu\ D \cdot e) \Rightarrow (\mu\ D \cdot e) \\
\mathit{ornAlg}\ \ O\ \ \mathit{ds}\ =\ \langle \mathit{erase}\ O\ \mathit{ds} \rangle\ .
\end{array}$$

Folding with the ornamental algebra gives us the *forgetful map*

$$\begin{array}{l}
\mathit{forget} : \forall\ \{I\ J\ e\}\ \{D : \mathsf{Desc}\ I\}\ (O : \mathsf{Orn}\ J\ e\ D) \to \\
\qquad \mu\ \lfloor O \rfloor \Rightarrow (\mu\ D \cdot e) \\
\mathit{forget}\ \ O\ =\ \mathit{fold}\ (\mathit{ornAlg}\ O)\ .
\end{array}$$

For example, the length of a list is computed by

$$\begin{array}{l}
\mathit{length} : \forall\ \{A\} \to \mathit{List}\ A \to \mathit{Nat} \\
\mathit{length}\ \ \{A\}\ =\ \mathit{forget}\ (\mathit{ListO}\ A)\ .
\end{array}$$

Algebraic ornaments. Being first-class data, ornaments can be generated systematically. McBride proposed a class of ornaments induced by algebras: Given $D : \mathsf{Desc}\ I$ and an algebra $\phi : [\![D]\!]\ J \Rightarrow J$, the *algebraic ornament* induced by ϕ is defined by

$$\begin{array}{l}
\mathit{algOrn} : \{I : \mathsf{Set}\}\ \{J : I \to \mathsf{Set}\} \to \\
\qquad (D : \mathsf{Desc}\ I)\ (\phi : [\![D]\!]\ J \Rightarrow J) \to \mathsf{Orn}\ (\Sigma\ I\ J)\ \mathsf{proj}_1\ D \\
\mathit{algOrn}\ (\mathsf{say}\ i)\ \phi = \mathsf{say}\ (\mathsf{ok}\ (i, \phi\ \mathsf{refl})) \\
\mathit{algOrn}\ (\sigma\ S\ D)\ \phi = \sigma\ S\ \lambda s \mapsto \mathit{algOrn}\ (D\ s)\ (\Lambda\ \phi\ s) \\
\mathit{algOrn}\ \{J = J\}\ (\mathsf{ask}\ i * D)\ \phi = \\
\qquad \Delta\ (J\ i)\ \lambda j \mapsto \mathsf{ask}\ (\mathsf{ok}\ (i, j)) * \mathit{algOrn}\ D\ (\Lambda\ \phi\ j)\ ,
\end{array}$$

where Λ is the currying operator. It is perhaps easier to understand algebraic ornaments in a specialised scenario. Suppose we are given $f : A \to B \to B$ and $e : B$, which constitute an algebra for folding a list of type $\mathsf{List}\ A$. The algebraic ornamentation of $\mathsf{List}\ A$ induced by that algebra would lead to the following datatype, where the new indices and elements are framed.

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathbf{data}\ \mathsf{AlgList} : \boxed{B} \to \mathsf{Set}\ \mathbf{where}} \\
\quad [\,] & : & \mathsf{AlgList}\ \boxed{e} \\
\quad \_{::}\_ & : & (x : A)\ \boxed{\{b : B\}}\ (\mathit{xs} : \mathsf{AlgList}\ \boxed{b}) \to \mathsf{AlgList}\ \boxed{(f\ x\ b)}
\end{array}$$

If we temporarily ignore the framed parts, we see that an AlgList is basically still a list. The difference is that the index of an AlgList is guaranteed to be the result of folding the underlying list using the given algebra: The new index for the type of $[\,]$ is e, which is the result of folding $[\,]$; for $\_{::}\_$, a new element $b : B$ is inserted before the recursive node xs for storing the index which has been inductively computed for xs and can be assumed to be the result of folding xs, so the final index $f\ x\ b$ is the result of folding $x :: \mathit{xs}$. In the generic implementation of *algOrn*, the tuple to be fed to the algebra ϕ is revealed one component at a time in each step of the case analysis, so ϕ acts as an accumulating parameter, accepting the component revealed in each step with the help of Λ, and emitting the final result when the say case is reached and the final component of the tuple, refl, is fed to it. Additionally, in the ask case where we encounter a recursive node, a new element is inserted by Δ for storing the index j that has been inductively computed for that node.

An example is vectors, which are lists indexed by the result of *length*, which is a fold whose algebra is *ornAlg* (*ListO A*), so the ornamentation from lists to vectors is algebraic:

$$\begin{array}{l}
\mathit{VecO} : (A : \mathsf{Set}) \to \mathsf{Orn}\ (\top \times \mathit{Nat})\ \mathsf{proj}_1\ \lfloor \mathit{ListO}\ A \rfloor \\
\mathit{VecO}\ \ A\ =\ \mathit{algOrn}\ \lfloor \mathit{ListO}\ A \rfloor\ (\mathit{ornAlg}\ (\mathit{ListO}\ A))\ .
\end{array}$$

It expands to

$$\begin{array}{l}
\sigma\ \mathsf{Bool}\ (\mathit{false} \mapsto \mathsf{say}\ (\mathsf{ok}\ (\mathsf{tt}, \mathit{zero})) \\
\qquad\qquad\quad \mathit{true} \mapsto\ \sigma\ A\ \lambda\_ \mapsto \Delta\ \mathit{Nat}\ \lambda n \mapsto \\
\qquad\qquad\qquad\qquad \mathsf{ask}\ (\mathsf{ok}\ (\mathsf{tt}, n)) * \mathsf{say}\ (\mathsf{ok}\ (\mathsf{tt}, \mathit{suc}\ n)))
\end{array}$$

where $\mathit{suc} = \lambda n \mapsto \langle \mathsf{true}, n, \mathsf{refl} \rangle : \mathit{Nat} \to \mathit{Nat}$. The decoded type $\mathit{Vec}\ A\ n = \mu\ \lfloor \mathit{VecO}\ A \rfloor\ (\mathsf{tt}, n)$ is essentially the same datatype delivered by the following native data declaration:[2]

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathbf{data}\ \mathsf{Vec}\ (A : \mathsf{Set}) : \mathsf{Nat} \to \mathsf{Set}\ \mathbf{where}} \\
\quad [\,] & : & \mathsf{Vec}\ A\ \mathsf{zero} \\
\quad \_{::}\_ & : & (x : A)\ \{n : \mathsf{Nat}\}\ (\mathit{xs} : \mathsf{Vec}\ A\ n) \to \mathsf{Vec}\ A\ (\mathsf{suc}\ n)\ .
\end{array}$$

An algebraically ornamented datatype does not carry more information than the raw datatype, but simply exposes some known knowledge in the index, namely the value obtained by folding the

---

[2] Frequently we translate decoded datatypes into native data declarations in this paper, but it is only for the purpose of exposition — the decoded datatypes have no formal connection with the natively declared datatypes in Agda (as suggested by the use of different fonts). It is hoped that in future dependently typed languages, native data declarations will become syntactic sugar for codes for datatypes, so the distinction between native datatypes and decoded datatypes will disappear [6].

underlying data. Hence there is not only a forgetful map from the ornamented datatype to the raw datatype, as induced by any ornament, but also a *remembering map* converting the raw datatype to the ornamented datatype, computing the index on the fly. The two maps are inverse to each other, meaning that the algebraically ornamented datatype and the raw datatype really are isomorphic. The remembering map can be defined generically,

$$\begin{aligned}&\mathit{remember} : \\ &\quad \forall \{I\ J\}\ (D : \mathsf{Desc}\ I)\ (\phi : [\![D]\!]\ J \Rightarrow J) \\ &\quad \{i\}\ (x' : \mu\ D\ i) \rightarrow \mu\ \lfloor \mathit{algOrn}\ D\ \phi \rfloor\ (i, \mathit{fold}\ \phi\ x')\ ,\end{aligned}$$

whose implementation is by generic induction and is omitted here.

The type of *remember* states what the index would be when raw data are converted to algebraically ornamented data, namely the result of folding the raw data. Conversely, when algebraically ornamented data are converted to raw data, the *recomputation lemma* states that the forgotten index can be recovered by folding the raw data.

$$\begin{aligned}&\mathit{recomputation} : \\ &\quad \forall \{I\ J\}\ (D : \mathsf{Desc}\ I)\ (\phi : [\![D]\!]\ J \Rightarrow J) \\ &\quad \{ij : \Sigma\ I\ J\}\ (x : \mu\ \lfloor \mathit{algOrn}\ D\ \phi \rfloor\ ij) \rightarrow \\ &\qquad \mathit{fold}\ \phi\ (\mathit{forget}\ (\mathit{algOrn}\ D\ \phi)\ x) \equiv \mathsf{proj}_2\ ij\end{aligned}$$

The implementation is again by generic induction and is omitted.

Algebraically ornamented datatypes provide an internalist way of constructing an object specified by requiring the result of folding that object to be a predetermined value. Suppose we are asked to construct

$$x' : \mu\ D\ i \qquad \text{such that} \qquad \mathit{fold}\ \phi\ x' \equiv j\ .$$

Instead of constructing x' directly and proving afterwards that the specification is satisfied, we can construct an ornamented object

$$x : \mu\ \lfloor \mathit{algOrn}\ D\ \phi \rfloor\ (i, j)$$

and set

$$x' = \mathit{forget}\ (\mathit{algOrn}\ D\ \phi)\ x\ :\ \mu\ D\ i\ .$$

Then the recomputation lemma says exactly that x' satisfies the specification. This construction method is central to the realisability interpretation we are proposing.

3. A realisability interpretation of ornamental-algebraic ornaments

From now on, we focus on what we might call *ornamental-algebraic ornaments*, i.e., algebraic ornaments induced by algebras that are themselves ornamental algebras; these can be given an intuitive interpretation, taking inspiration from the theory of *realisability*. In the Curry-Howard world, we are familiar with what it means for a proof term to *prove* a proposition, i.e., to inhabit a type — the term is related to the type by the typing meta-relation. Compare this with the realisability view, under which we say a term x' *realises* (instead of *proves*) a proposition φ when x' is related to φ by some relation defined in the language, traditionally written $x' \Vdash \varphi$. The predicate $\lambda x' \mapsto x' \Vdash \varphi$ is called the *realisability predicate* for φ, so saying that x' realises φ is equivalent to saying that x' satisfies the realisability predicate for φ. When $x' \Vdash \varphi$ holds, the term x' is called a *realiser* of the proposition φ. The relation $\_\Vdash\_$ being defined in the language, a proof of $x' \Vdash \varphi$ exists as a proof term, which we call a *realisability proof*. In his original realisability paper [9], Kleene hinted that a realiser is "an incomplete communication of a more specific statement," and a realisability proof provides "items as may be necessary to complete the communication." This is close to our interpretation: A realiser is an incomplete proof that can somehow be derived from a complete proof without losing the basic structure of the latter. A realisability predicate states what information needs to be supplied if we wish to augment an incomplete proof to a complete one, and a realisability proof provides the missing information.

For example, let us consider lists as realisers of the vector type. That is, lists are incomplete vectors, in the sense that a list is a vector whose length information is forgotten. To synthesise a vector of length n out of a list, we need to prove that the list has length n. One way to state this is to use the inductively defined relation

$$\begin{aligned}&\mathbf{data}\ \mathsf{Length}\ \{A : \mathsf{Set}\}\ : \mathsf{Nat} \rightarrow \boxed{\mathsf{List}\ A} \rightarrow \mathsf{Set}\ \mathbf{where} \\ &\quad \mathsf{nil}\quad : \mathsf{Length}\ \mathsf{zero}\ \boxed{[\,]} \\ &\quad \mathsf{cons}\ : \forall\ \{x\ n\ \boxed{xs}\} \rightarrow \\ &\qquad\qquad \mathsf{Length}\ n\ \boxed{xs} \rightarrow \mathsf{Length}\ (\mathsf{suc}\ n)\ \boxed{(x :: xs)}\ .\end{aligned}$$

The predicate $\mathsf{Length}\ n$ on lists is the realisability predicate which, when satisfied by a list, states that the list can be upgraded to a vector of length n. That is, we can establish an isomorphism

$$\mathsf{Vec}\ A\ n\ \cong\ \Sigma\ (\mathsf{List}\ A)\ (\mathsf{Length}\ n)$$

which allows a list that can be proved to have length n to be converted to a vector of length n or vice versa. If we temporarily ignore the framed parts of Length, we see that it is just the vector type, so an inhabitant of $\mathsf{Length}\ n\ xs$ is actually a vector of length n whose type is indexed by the underlying list xs. Since the underlying list is computed by the forgetful map, we see that Length is the algebraic ornamentation of $\mathsf{Vec}\ A$ using the ornamental algebra from vectors to lists. The use of the ornament language here is a hint of datatype-genericity.

So let us go generic: An ornament $O : \mathsf{Orn}\ J\ e\ D$ of a description $D : \mathsf{Desc}\ I$ states how to augment the datatype $\mu\ D : I \rightarrow \mathsf{Set}$ to a richer datatype $\mu\ \lfloor O \rfloor : J \rightarrow \mathsf{Set}$, and induces a forgetful map $\mathit{forget}\ O : \mu\ \lfloor O \rfloor \Rightarrow \mu\ D \cdot e$. If we regard $\mu\ \lfloor O \rfloor$ as the *complete type*, then $\mu\ D$ is incomplete with respect to $\mu\ \lfloor O \rfloor$ and serves as the type of *potential* realisers of $\mu\ \lfloor O \rfloor$.[3] A complete object of type $\mu\ \lfloor O \rfloor\ j$ can be compressed by $\mathit{forget}\ O$ to an incomplete one of type $\mu\ D\ (e\ j)$ but retaining the basic structure. Conversely, given an incomplete object $x' : \mu\ D\ (e\ j)$, can we reconstruct a complete object $x : \mu\ \lfloor O \rfloor\ j$ such that x has the same basic structure as x', i.e., $\mathit{forget}\ O\ x \equiv x'$ is provable? This is exactly the scenario where the construction method supported by algebraically ornamented datatypes can be applied, since $\mathit{forget}\ O = \mathit{fold}\ (\mathit{ornAlg}\ O)$. So the answer is: Yes, if we can construct

$$r : \mu\ \lfloor \mathit{algOrn}\ \lfloor O \rfloor\ (\mathit{ornAlg}\ O) \rfloor\ (j, x')\ ,$$

then by setting

$$x = \mathit{forget}\ (\mathit{algOrn}\ \lfloor O \rfloor\ (\mathit{ornAlg}\ O))\ r\ :\ \mu\ \lfloor O \rfloor\ j$$

we are assured by the recomputation lemma that

$$\begin{aligned}&\quad\ \mathit{forget}\ O\ x \\ &\equiv\ \mathit{fold}\ (\mathit{ornAlg}\ O)\ x \\ &\equiv\ \mathit{fold}\ (\mathit{ornAlg}\ O)\ (\mathit{forget}\ (\mathit{algOrn}\ \lfloor O \rfloor\ (\mathit{ornAlg}\ O))\ r) \\ &\equiv\ \quad \{\text{-}\ \mathit{recomputation}\ \lfloor O \rfloor\ (\mathit{ornAlg}\ O)\ r\ \text{-}\} \\ &\quad\ \mathsf{proj}_2\ (j, x') \\ &\equiv\ x'\ .\end{aligned}$$

---

[3] A $\mu\ D$ object is not necessarily a realiser of $\mu\ \lfloor O \rfloor$, as it may not satisfy the realisability predicate. We will nevertheless simply call $\mu\ D$ the *realiser type*, instead of the "potential realiser type."

The predicate $\lambda x' \mapsto \mu\ \lfloor \mathit{algOrn}\ \lfloor O \rfloor\ (\mathit{ornAlg}\ O) \rfloor\ (j, x')$ thus acts as the realisability predicate. Consequently we define

$$\begin{array}{l}
\mathit{rpOrn} : \forall\ \{I\ J\ e\}\ \{D : \mathsf{Desc}\ I\}\ (O : \mathsf{Orn}\ J\ e\ D) \to \\
\qquad\quad \mathsf{Orn}\ (\Sigma\ J\ (\mu\ D \cdot e))\ \mathsf{proj}_1\ \lfloor O \rfloor \\
\mathit{rpOrn}\ \ O\ =\ \mathit{algOrn}\ \lfloor O \rfloor\ (\mathit{ornAlg}\ O)
\end{array}$$

and

$$\begin{array}{l}
[\_]\_\Vdash\_ : \forall\ \{I\ J\ e\}\ \{D : \mathsf{Desc}\ I\} \to \\
\qquad\quad (j : J)\ (x' : \mu\ D\ (e\ j))\ (O : \mathsf{Orn}\ J\ e\ D) \to \mathsf{Set} \\
[\,j\,]\ x' \Vdash O\ =\ \mu\ \lfloor \mathit{rpOrn}\ O \rfloor\ (j, x')\ .
\end{array}$$

This is interpreted as the type of a proof that x' can be completed to yield an object of $\mu\ \lfloor O \rfloor$. We also define $x' \Vdash O = [\_]\ x' \Vdash O$ so the index j can be omitted when it can be inferred.

Since a realisability predicate is implemented as an algebraic ornamentation of the complete type, it is isomorphic to the complete type — more precisely, to be called a type a realisability predicate needs to be applied to a realiser, so the complete type is isomorphic to the dependent pair of the realiser type and the realisability predicate. The isomorphism can be nicely interpreted in terms of realisability: Given a complete object, a realiser can be obtained by applying *forget O* to the object, and the corresponding realisability proof is produced by

$$\begin{array}{l}
\mathfrak{R} : \forall\ \{I\ J\ e\}\ \{D : \mathsf{Desc}\ I\}\ (O : \mathsf{Orn}\ J\ e\ D) \\
\qquad \{j\}\ (x : \mu\ \lfloor O \rfloor\ j) \to \mathit{forget}\ O\ x \Vdash O \\
\mathfrak{R}\ \ O\ \ x\ =\ \mathit{remember}\ \lfloor O \rfloor\ (\mathit{ornAlg}\ O)\ x\ .
\end{array}$$

We call this direction of the isomorphism the *realisability transformation*, because it helps to switch from the “proving” view to the “realising” view. The inverse transformation metaphorically combines a realiser and its realisability proof, whose computation depends only on the latter:

$$\begin{array}{l}
\mathfrak{R}^{-1} : \forall\ \{I\ J\ e\}\ \{D : \mathsf{Desc}\ I\}\ (O : \mathsf{Orn}\ J\ e\ D) \\
\qquad\ \ \{j\}\ (x' : \mu\ D\ (e\ j))\ (r : x' \Vdash O) \to \mu\ \lfloor O \rfloor\ j \\
\mathfrak{R}^{-1}\ \ O\ \ x'\ \ r\ =\ \mathit{forget}\ (\mathit{rpOrn}\ O)\ r\ .
\end{array}$$

That $\mathfrak{R}$ and $\mathfrak{R}^{-1}$ are indeed inverse to each other can be proven by *recomputation* and the fact that *forget* and *remember* are inverses. For example, one inverse property we will need is

$$\begin{array}{l}
\mathit{realiser\text{-}recovery} : \\
\quad \forall\ \{I\ J\ e\}\ \{D : \mathsf{Desc}\ I\}\ (O : \mathsf{Orn}\ J\ e\ D) \to \\
\quad (x' : \mu\ D\ (e\ j))\ (r : x' \Vdash O) \to \mathit{forget}\ O\ (\mathfrak{R}^{-1}\ O\ x'\ r) \equiv x' \\
\mathit{realiser\text{-}recovery}\ O\ x'\ r = \mathit{recomputation}\ \lfloor O \rfloor\ (\mathit{ornAlg}\ O)\ r\ ,
\end{array}$$

which says that the realiser extracted from a complete object synthesised from a realiser x' is just x' again.

To recap: By defining an ornament, we specify a complete type relative to a realiser type, and the corresponding realisability predicate can be immediately derived from that ornament. From this follow the realisability transformation and its inverse transformation that allow us to break a complete object into a realiser and a corresponding realisability proof, or recover a complete object from a realisability proof, which depends on a realiser.

Examples. Let us turn back to the example in which lists are viewed as realisers of the vector type, which arises from the ornament *VecO A*. The derived realisability predicate is

$$\begin{array}{l}
\mathit{Length} : \forall\ \{A\} \to \mathit{Nat} \to \mathit{List}\ A \to \mathsf{Set} \\
\mathit{Length}\ \ \{A\}\ \ n\ \ \mathit{xs}\ =\ [\mathsf{tt}, n]\ \mathit{xs} \Vdash \mathit{VecO}\ A\ ,
\end{array}$$

which translates to the Length datatype given previously.

A slightly confusing but classic example is given by the ornament *ListO A*. The complete type specified by this ornament is *List A*, and the realiser type is *Nat* — a natural number is an incomplete list, with the elements missing. The derived realisability predicate $n \Vdash \mathit{ListO}\ A$ is just *Vec A n*, meaning that to augment a natural number n to a list of A’s we need to supply a vector of type *Vec A n*, i.e., n elements of type A. It may seem strange at first that to construct a list, we end up constructing a vector, which is “heavier” than a list. But in fact we are asking not just for any list, but a list whose length is n. By constructing the list as a vector indexed by n, the requirement that the list constructed should have length n, i.e., that it has the same basic structure as n, is met by construction. A metaphor for this is that n is scaffolding to guide the construction of a list, and a vector is the finished construction still with the scaffold. To get the list constructed, we remove the scaffold by $\mathfrak{R}^{-1}$, i.e., *forget*.

For an example other than vectors, assume that a less-than-or-equal-to relation $\_\leq\_$ on natural numbers is suitably defined, and consider the following datatype for sorted lists of natural numbers indexed by a lower bound:

$$\begin{array}{l}
\textbf{data}\ \mathsf{SList} : \mathsf{Nat} \to \mathsf{Set}\ \textbf{where} \\
\quad \mathsf{snil}\quad\ : \forall\ \{b\} \to \mathsf{SList}\ b \\
\quad \mathsf{scons}\ :\ (x : \mathsf{Nat}) \to \forall\ \{b\} \to b \leq x \to \mathsf{SList}\ x \to \mathsf{SList}\ b\ .
\end{array}$$

Coding sorted lists as an ornamentation of lists,

$$\begin{array}{lcl}
\mathit{SListO} & : & \mathsf{Orn}\ \mathit{Nat}\ !\ \lfloor \mathit{List}\ \mathit{Nat} \rfloor \\
\mathit{SListO} & = & \sigma\ \mathsf{Bool}\ (\mathit{false} \mapsto \Delta\ \mathit{Nat}\ (\lambda b \mapsto \mathsf{say}\ (\mathsf{ok}\ b)) \\
 & & \qquad\qquad\quad \mathit{true} \mapsto\ \ \sigma\ \mathit{Nat}\ (\lambda x \mapsto \\
 & & \qquad\qquad\qquad\qquad\quad \Delta\ \mathit{Nat}\ (\lambda b \mapsto \Delta\ (b \leq x)\ (\lambda\_ \mapsto \\
 & & \qquad\qquad\qquad\qquad\qquad \mathsf{ask}\ (\mathsf{ok}\ x) * \mathsf{say}\ (\mathsf{ok}\ b)))))\ ,
\end{array}$$

where $! = \lambda\_ \mapsto \mathsf{tt} : \forall\ \{A\} \to A \to \top$, we obtain $\mathit{SList} = \mu\ \lfloor \mathit{SListO} \rfloor$. The derived realisability predicate is

$$\begin{array}{l}
\mathit{Sorted} : \mathit{Nat} \to \mathit{List}\ \mathit{Nat} \to \mathsf{Set} \\
\mathit{Sorted}\ \ n\ \ \mathit{xs}\ =\ [n]\ \mathit{xs} \Vdash \mathit{SListO}\ \mathit{Nat}\ ,
\end{array}$$

which translates to

$$\begin{array}{l}
\textbf{data}\ \mathsf{Sorted} : \mathsf{Nat} \to \mathsf{List}\ \mathsf{Nat} \to \mathsf{Set}\ \textbf{where} \\
\quad \mathsf{nil}\quad : \forall\ \{b\} \to \mathsf{Sorted}\ b\ [\,] \\
\quad \mathsf{cons}\ : \forall\ \{x\ b\} \to b \leq x \to \\
\qquad\qquad\ \forall\ \{\mathit{xs}\} \to \mathsf{Sorted}\ x\ \mathit{xs} \to \mathsf{Sorted}\ b\ (x :: \mathit{xs})\ .
\end{array}$$

It is an inductively defined predicate stating that a list is sorted and bounded below by a number. If we can prove that a list satisfies this predicate, then the list can be cast as a sorted list bounded below.

For an example other than lists, recall the finite numbers presented in Section 1, which can be coded as an ornamentation of natural numbers:

$$\begin{array}{l}
\mathit{FinO} : \mathsf{Orn}\ \mathit{Nat}\ !\ \mathit{NatD} \\
\mathit{FinO} = \\
\quad \sigma\ \mathsf{Bool}\ (\mathit{false} \mapsto \Delta\ \mathit{Nat}\ (\lambda n \mapsto \mathsf{say}\ (\mathsf{ok}\ (\mathit{suc}\ n))) \\
\qquad\qquad\quad \mathit{true} \mapsto\ \ \Delta\ \mathit{Nat}\ (\lambda n \mapsto \mathsf{ask}\ (\mathsf{ok}\ n) * \mathsf{say}\ (\mathsf{ok}\ (\mathit{suc}\ n))))\ .
\end{array}$$

The decoded type of finite numbers is thus $\mathit{Fin} = \mu\ \lfloor \mathit{FinO} \rfloor$. The derived realisability predicate translates to the greater-than relation also presented in Section 1 — to say a natural number n is a finite number bounded by m, what we need to prove is exactly $m > n$.

Realisability predicates for algebraic ornaments. The example regarding lists as realisers of the vector type may have made the reader feel uneasy — the derived realisability predicate *Length* looks rather heavyweight. Given that an algebraic ornament does not add extra information to a datatype, shouldn’t the realisability predicate be more lightweight and sometimes even trivially satisfiable? Indeed, the realisability predicate for an algebraic ornament should simply amount to an equality, e.g., $\mathit{length}\ \mathit{xs} \equiv n$ for the ornament *VecO A* instead of *Length n xs*, and this can be proved generically.

For one direction, we wish to prove

$$\begin{array}{l}
\mathit{AOE}\ :\ \forall\ \{I\ J\}\ (D : \mathsf{Desc}\ I)\ (\phi : [\![D]\!]\ J \Rightarrow J) \\
\qquad\quad \{ij : \Sigma\ I\ J\}\ \{x' : \mu\ D\ (\mathsf{proj}_1\ ij)\} \to \\
\qquad\quad (r : [\mathit{ij}]\ x' \Vdash \mathit{algOrn}\ D\ \phi) \to \mathit{fold}\ \phi\ x' \equiv \mathsf{proj}_2\ \mathit{ij}\ .
\end{array}$$

Note that the realiser x' is ornamentally two levels away from the realisability proof r, but since the two ornaments involved are both algebraic, x' and r really are isomorphic. The goal type looks quite similar to the conclusion of the recomputation lemma at the first level, so we try to apply *recomputation* to a complete object, the natural choice being $\mathfrak{R}^{-1}\ (\mathit{algOrn}\ D\ \phi)\ x'\ r$. We supply the proof term

$$\mathit{recomputation}\ D\ \phi\ (\mathfrak{R}^{-1}\ (\mathit{algOrn}\ D\ \phi)\ x'\ r) \tag{1}$$

as the result, whose type

$$\mathit{fold}\ \phi\ (\mathit{forget}\ (\mathit{algOrn}\ D\ \phi)\ (\mathfrak{R}^{-1}\ (\mathit{algOrn}\ D\ \phi)\ x'\ r)) \equiv \mathsf{proj}_2\ \mathit{ij}$$

requires a bit of tweaking, though: The argument to $\mathit{fold}\ \phi$ should be just x' to match the goal type, which is achieved by rewriting with

$$\mathit{realiser\text{-}recovery}\ (\mathit{algOrn}\ D\ \phi)\ x'\ r\ . \tag{2}$$

In Agda, we first use **with** to put the term (1) into the context and then **rewrite** the context with (2) before delivering the term left in the context as the result, whose type has been appropriately rewritten. This programming pattern will be used a lot.

The other direction requires us to prove

$$\begin{array}{ll} \mathit{AOE}^{-1} : & \forall\ \{I\ J\}\ (D : \mathsf{Desc}\ I)\ (\phi : \llbracket D \rrbracket\ J \Rightarrow J) \\ & \{\mathit{ij} : \Sigma\ I\ J\}\ \{x' : \mu\ D\ (\mathsf{proj}_1\ \mathit{ij})\} \to \\ & \mathit{fold}\ \phi\ x' \equiv \mathsf{proj}_2\ \mathit{ij} \to [\mathit{ij}]\ x' \Vdash \mathit{algOrn}\ D\ \phi\ . \end{array}$$

Promoting x' two levels up should do the job, so we give the term

$$\mathfrak{R}\ (\mathit{algOrn}\ D\ \phi)\ (\mathit{remember}\ D\ \phi\ x')$$

as the result after tweaking its type, which is originally

$$\begin{array}{l} [\mathsf{proj}_1\ \mathit{ij}, \mathit{fold}\ \phi\ x'] \\ \qquad \mathit{forget}\ (\mathit{algOrn}\ D\ \phi)\ (\mathit{remember}\ D\ \phi\ x') \Vdash \mathit{algOrn}\ D\ \phi\ . \end{array}$$

Since *forget* is a left inverse to *remember* and we have an assumption $\mathit{fold}\ \phi\ x' \equiv \mathsf{proj}_2\ \mathit{ij}$, the type can be rewritten as our goal.

Incidentally, implementation of *remember* and *recomputation* can be made symmetric under the realisability view, i.e., if the combinators $\mathfrak{R}$, $\mathfrak{R}^{-1}$, *AOE*, and AOE^{-1} are taken as primitives. First consider *remember*: To promote $x' : \mu\ D\ i$ to an object of type

$$\mu\ \lfloor \mathit{algOrn}\ D\ \phi \rfloor\ (i, \mathit{fold}\ \phi\ x')\ ,$$

we apply the inverse realisability transformation $\mathfrak{R}^{-1}$ to x' and a corresponding realisability proof, which can simply be a proof of $\mathit{fold}\ \phi\ x' \equiv \mathit{fold}\ \phi\ x'$ because of AOE^{-1}.

$$\begin{array}{l} \mathit{remember} : \\ \quad \forall\ \{I\ J\}\ (D : \mathsf{Desc}\ I)\ (\phi : \llbracket D \rrbracket\ J \Rightarrow J) \\ \quad \{i\}\ (x' : \mu\ D\ i) \to \mu\ \lfloor \mathit{algOrn}\ D\ \phi \rfloor\ (i, \mathit{fold}\ \phi\ x') \\ \mathit{remember}\ D\ \phi\ x' = \mathfrak{R}^{-1}\ (\mathit{algOrn}\ D\ \phi)\ x'\ (\mathit{AOE}^{-1}\ D\ \phi\ \mathsf{refl}) \end{array}$$

As for *recomputation*, we are given a complete object x and asked to produce the equality form of a realisability proof, which we can easily obtain by applying the realisability transformation $\mathfrak{R}$ to x and then resorting to *AOE*.

$$\begin{array}{l} \mathit{recomputation} : \\ \quad \forall\ \{I\ J\}\ (D : \mathsf{Desc}\ I)\ (\phi : \llbracket D \rrbracket\ J \Rightarrow J) \\ \quad \{\mathit{ij} : \Sigma\ I\ J\}\ (x : \mu\ \lfloor \mathit{algOrn}\ D\ \phi \rfloor\ \mathit{ij}) \to \\ \qquad \mathit{fold}\ \phi\ (\mathit{forget}\ (\mathit{algOrn}\ D\ \phi)\ x) \equiv \mathsf{proj}_2\ \mathit{ij} \\ \mathit{recomputation}\ D\ \phi\ x = \mathit{AOE}\ D\ \phi\ (\mathfrak{R}\ (\mathit{algOrn}\ D\ \phi)\ x) \end{array}$$

We see that *remember* is $\mathfrak{R}^{-1}$ whose argument is produced by AOE^{-1}, while *recomputation* is $\mathfrak{R}$ whose result is modified by *AOE*. Hence *recomputation* and *remember* are actually the realisability transformation and the inverse transformation, specialised for algebraic ornaments.

Function upgrade. After we define an ornament to get a fancier type, naturally we want to port functions working on the original type to the fancier type. For example, we should be able to upgrade list append to vector append. Our strategy is based on realisers and realisability predicates for *function types*. In type theory, a proof of an implication $\varphi \to \psi$ takes a proof of φ to a proof of ψ, while in the realisability world the role of proofs are taken by realisers, so a realiser of $\varphi \to \psi$ takes a realiser of φ to a realiser of ψ, i.e., it is a function $f' : \varphi' \to \psi'$ where φ' and ψ' are the type of potential realisers of φ and ψ. But in order to justify that f' really takes realisers to realisers, we need to prove that when an input $x' : \varphi'$ is a realiser, i.e., we are given a proof of $x' \Vdash \varphi$, the output $f'\ x' : \psi'$ is also a realiser, i.e., we can produce a proof of $f'\ x' \Vdash \psi$. This justification is thus a proof of type

$$(x' : \varphi') \to x' \Vdash \varphi \to f'\ x' \Vdash \psi\ ,$$

which is defined to be the realisability predicate for $\varphi \to \psi$. Back in the context of ornaments, this suggests that to upgrade a function, we can consider it as a realiser of a function type between ornamented types, and obtain a complete function by supplying a suitable realisability proof.

For the example of upgrading list append to vector append, our goal is to write the append function for vectors

$$\mathit{vappend} : \forall\ \{A\ m\ n\} \to \mathit{Vec}\ A\ m \to \mathit{Vec}\ A\ n \to \mathit{Vec}\ A\ (m+n)$$

in terms of list append

$$\_{+\!\!+}\_ : \forall\ \{A\} \to \mathit{List}\ A \to \mathit{List}\ A \to \mathit{List}\ A\ .$$

Given $\mathit{xs} : \mathit{Vec}\ A\ m$ and $\mathit{ys} : \mathit{Vec}\ A\ n$, to produce a vector of type $\mathit{Vec}\ A\ (m+n)$, we invoke $\mathfrak{R}^{-1}$ and thereby split the goal into two parts — the realiser and the realisability proof. The realiser, which is a list, is obtained by extracting the two underlying lists $\mathit{xs}' = \mathit{forget}\ (\mathit{VecO}\ A)\ \mathit{xs}$ and $\mathit{ys}' = \mathit{forget}\ (\mathit{VecO}\ A)\ \mathit{ys}$ and appending them. For the realisability proof, because of AOE^{-1}, the proof obligation is reduced to the equality

$$\mathit{length}\ (\mathit{xs}' \mathbin{+\!\!+} \mathit{ys}') \equiv m+n\ .$$

We know *length* is a list homomorphism, i.e.,

$$\mathit{length}\ (\mathit{xs}' \mathbin{+\!\!+} \mathit{ys}') \equiv \mathit{length}\ \mathit{xs}' + \mathit{length}\ \mathit{ys}' \quad \text{for all } \mathit{xs}' \text{ and } \mathit{ys}'\ .$$

The type of the realisability proof for list append is merely a restatement of the fact above:

$$\begin{array}{l} \mathit{append\text{-}length} : \\ \quad \forall\ \{A\}\ (\mathit{xs}'\ \mathit{ys}' : \mathit{List}\ A)\ \{m\ n\} \to \\ \quad \mathit{length}\ \mathit{xs}' \equiv m \to \mathit{length}\ \mathit{ys}' \equiv n \to \mathit{length}\ (\mathit{xs}' \mathbin{+\!\!+} \mathit{ys}') \equiv m+n\ . \end{array}$$

And the two equality premises of *append-length* are discharged by applying $\mathfrak{R}$ and *AOE* to *xs* and *ys*. The whole Agda translation is shown below.

$$\begin{array}{l} \mathit{vappend} : \forall\ \{A\ m\ n\} \to \mathit{Vec}\ A\ m \to \mathit{Vec}\ A\ n \to \mathit{Vec}\ A\ (m+n) \\ \mathit{vappend}\ \{A\}\ \mathit{xs}\ \mathit{ys} = \mathfrak{R}^{-1}\ (\mathit{VecO}\ A)\ (\mathit{xs}' \mathbin{+\!\!+} \mathit{ys}') \\ \quad (\mathit{AOE}^{-1}\ \lfloor \mathit{ListO}\ A \rfloor\ \phi\ (\mathit{append\text{-}length}\ \mathit{xs}'\ \mathit{ys}'\ \mathit{eq}_1\ \mathit{eq}_2)) \\ \quad \textbf{where}\ \begin{array}[t]{lcl} \mathit{xs}' & = & \mathit{forget}\ (\mathit{VecO}\ A)\ \mathit{xs} \\ \mathit{ys}' & = & \mathit{forget}\ (\mathit{VecO}\ A)\ \mathit{ys} \\ \phi & = & \mathit{ornAlg}\ (\mathit{ListO}\ A) \\ \mathit{eq}_1 & = & \mathit{AOE}\ \lfloor \mathit{ListO}\ A \rfloor\ \phi\ (\mathfrak{R}\ (\mathit{VecO}\ A)\ \mathit{xs}) \\ \mathit{eq}_2 & = & \mathit{AOE}\ \lfloor \mathit{ListO}\ A \rfloor\ \phi\ (\mathfrak{R}\ (\mathit{VecO}\ A)\ \mathit{ys}) \end{array} \end{array}$$

4. A first step towards ornament composition

The inverse realisability transformation combines a realisability proof with a realiser to get a complete object. For example, we combine length information with a list to get a vector, or we combine a proof of the *Sorted* predicate with a list to get a sorted list. It is then natural to ask: Can we combine *both* the length information and the sortedness proof with a list, to get a sorted vector?

To clarify, the datatypes involved are shown in the following diagram, ornaments drawn as double-headed arrows and the realisability predicates framed. It can be read as "the datatype *List Nat* is revised to the datatype *SList b* by the ornament *SListO*" and so on.

$$\begin{tikzcd}
& List\ Nat \arrow[dl, two heads, "SListO"'] \arrow[d, two heads, "SVecO"] \arrow[dr, two heads, "VecO\ Nat"] & \\
SList\ b \arrow[d, two heads] & SVec\ b\ n \arrow[d, two heads] & Vec\ Nat\ n \arrow[d, two heads] \\
\boxed{Sorted\ b} & \boxed{SLen\ b\ n} & \boxed{Length\ n}
\end{tikzcd}$$

We know that to promote a list *xs* to a sorted vector, we need to provide a realisability proof of type *SLen b n xs*, but what we are given are proofs of *Sorted b xs* and *Length n xs*. Nevertheless, intuitively we see that $Sorted\ b\ xs \times Length\ n\ xs$ is isomorphic to *SLen b n xs*, i.e., the realisability predicate for the ornament *SVecO* is the composition (pointwise conjunction) of the realisability predicates for *SListO* and *VecO Nat*. Since realisability predicates are derived from ornaments, we are led to seeking a way of regarding *SVecO* as the composition of *SListO* and *VecO Nat*. Our hypothesis, then, is that the realisability predicate for a composite ornament amounts to the composition (pointwise conjunction) of the realisability predicates for the component ornaments.

As an initial experiment, in this paper we consider only composition of two ornaments one of which is algebraic, which has a simpler implementation. Let $D : \mathsf{Desc}\ I$, $O : \mathsf{Orn}\ J\ e\ D$, and $\phi : [\![D]\!]\ K \Rightarrow K$. The composition of O and $algOrn\ D\ \phi$ will be called $algOrn'\ O\ \phi$. The datatypes involved are shown in the following diagram. We omit the names of ornaments on the arrows that represent them, because the names are shown in the datatypes at the end of the arrows. A dashed arrow indicates an algebraic ornament.

$$\begin{tikzcd}
& \mu\ D \arrow[dl, two heads] \arrow[dd, two heads] \arrow[dr, two heads, dashed] & \\
\mu\ \lfloor O \rfloor & & \mu\ \lfloor algOrn\ D\ \phi \rfloor \\
& \mu\ \lfloor algOrn'\ O\ \phi \rfloor &
\end{tikzcd}$$

The function *algOrn′* does the same thing as *algOrn* except that it works *on an ornament* — $algOrn'\ O\ \phi$ patches O algebraically so the resulting ornament on D has new indices which are the result of folding with ϕ. We call it an *algebraic ornament-ornament*. (Fortunately this rather ugly name will appear only once more.)

```
algOrn' : {I J : Set} {K : I → Set} {e : J → I} {D : Desc I}
          (O : Orn J e D) (φ : ⟦D⟧ K ⇒ K) →
          Orn (Σ J (K · e)) (e · proj₁) D
algOrn' (say (ok j)) φ = say (ok (j, φ refl))
algOrn' (σ S O)      φ = σ S λs ↦ algOrn' (O s) (Λ φ s)
algOrn' {K = K} {e} (ask (ok j) ∗ O) φ =
    Δ (K (e j)) λk ↦ ask (ok (j,k)) ∗ algOrn' O (Λ φ k)
algOrn' (Δ S O)      φ = Δ S λs ↦ algOrn' (O s) φ .
```

Each of the three ornaments appearing in the diagram induces its own realisability predicate, and we are going to show that a realisability proof for $algOrn'\ O\ \phi$ can be projected to a realisability proof for O or for $algOrn\ D\ \phi$, or synthesised by integrating realisability proofs for O and $algOrn\ D\ \phi$.

Projections. First we deal with the left and right projection,

```
project-l :
  ∀ {I J K e} {D : Desc I} (O : Orn J e D) (φ : ⟦D⟧ K ⇒ K)
  {jk : Σ J (K · e)} {x' : μ D (e (proj₁ jk))} →
  (r : [jk] x' ⊩ algOrn' O φ) → [proj₁ jk] x' ⊩ O
```

```
project-r :
  ∀ {I J K e} {D : Desc I} (O : Orn J e D) (φ : ⟦D⟧ K ⇒ K)
  {jk : Σ J (K · e)} {x' : μ D (e (proj₁ jk))} →
  (r : [jk] x' ⊩ algOrn' O φ) → [(e × id) jk] x' ⊩ algOrn D φ ,
```

where _×_ is overloaded to mean $(f \times g)\ (x, y) = (f\ x, g\ y)$. It is difficult to prove the projections directly by generic induction on r. Rather, we will first complete the ornament diagram by defining two *difference ornaments* from which the decoded datatypes are isomorphic to $\mu\ \lfloor algOrn'\ O\ \phi \rfloor$ (the isomorphisms are shown as two-way arrows below),

$$\begin{tikzcd}
& \mu\ D \arrow[dl, two heads] \arrow[dd, two heads] \arrow[dr, two heads, dashed] & \\
\mu\ \lfloor O \rfloor \arrow[d, two heads, dashed] & & \mu\ \lfloor algOrn\ D\ \phi \rfloor \arrow[d, two heads] \\
\mu\ \lfloor diffOrn\text{-}l\ O\ \phi \rfloor \arrow[r, leftrightarrow] & \mu\ \lfloor algOrn'\ O\ \phi \rfloor \arrow[r, leftrightarrow] & \mu\ \lfloor diffOrn\text{-}r\ O\ \phi \rfloor
\end{tikzcd}$$

and then route a realisability proof through the appropriate forgetful maps, isomorphisms, and remembering maps to get the proof we want. Let us look at the left part of the completed diagram, to which the induced realisability predicates have been added.

$$\begin{tikzcd}
& & \mu\ D \arrow[dl, two heads] \arrow[dd, two heads] \\
\boxed{\_ \Vdash O} & \mu\ \lfloor O \rfloor \arrow[l, two heads, dashed] \arrow[d, two heads, dashed] & \\
& \mu\ \lfloor diffOrn\text{-}l\ O\ \phi \rfloor \arrow[r, leftrightarrow] & \mu\ \lfloor algOrn'\ O\ \phi \rfloor \arrow[d, two heads, dashed] \\
& & \boxed{\_ \Vdash algOrn'\ O\ \phi}
\end{tikzcd}$$

The left difference ornament is an algebraic ornament defined by

```
diffOrn-l : ∀ {I J K e} {D : Desc I}
            (O : Orn J e D) (φ : ⟦D⟧ K ⇒ K) →
            Orn (Σ J (K · e)) proj₁ ⌊O⌋
diffOrn-l O φ = algOrn ⌊O⌋ (φ · erase O) .
```

One direction of the isomorphism between $\mu\ \lfloor diffOrn\text{-}l\ O\ \phi \rfloor$ and $\mu\ \lfloor algOrn'\ O\ \phi \rfloor$ is iso_1,

```
iso₁ : ∀ {I J K e} {D : Desc I}
       (O : Orn J e D) (φ : ⟦D⟧ K ⇒ K) →
       μ ⌊algOrn' O φ⌋ ⇒ μ ⌊diffOrn-l O φ⌋
iso₁ O φ = fold (⟨_⟩ · iso₁-cast O φ) ,
```

where iso_1-*cast* is a polymorphic restructuring map like *erase*, which is actually just an identity map.

```
iso₁-cast : ∀ {I J K e} {D : Desc I}
            (O : Orn J e D) (φ : ⟦D⟧ K ⇒ K) {X} →
            ⟦⌊algOrn' O φ⌋⟧ X ⇒ ⟦⌊diffOrn-l O φ⌋⟧ X
iso₁-cast (say (ok j)) φ refl = refl
iso₁-cast (σ S O) φ (s, xs) = s, iso₁-cast (O s) (Λ φ s) xs
iso₁-cast (ask (ok j) ∗ O) φ (k, x, xs) = k, x,
                                      iso₁-cast O (Λ φ k) xs
iso₁-cast (Δ S O) φ (s, xs) = s, iso₁-cast (O s) φ xs
```

The other direction of the isomorphism, iso_2, has the same implementation. Each ornament induces a forgetful map, and additionally a remembering map if it is algebraic, as shown in Figure 1.

$$\begin{tikzcd}
 & & \mu\, D \\
r_1 : \boxed{\_ \Vdash O} \arrow[r, bend left, "\Re^{-1}"] & \mu\, \lfloor O \rfloor \arrow[l, bend left, "\Re"] \arrow[ur, "\mathit{forget}"] \arrow[d, bend left, "\mathit{remember}"] & \\
 & \mu\, \lfloor \mathit{diffOrn\text{-}l}\; O\; \phi \rfloor \arrow[u, bend left, "\mathit{forget}"] \arrow[r, bend left, "\mathit{iso}_2"] & \mu\, \lfloor \mathit{algOrn}'\; O\; \phi \rfloor \arrow[l, bend left, "\mathit{iso}_1"] \arrow[uu, "\mathit{forget}"'] \arrow[d, bend left, "\Re"] \\
 & & r : \boxed{\_ \Vdash \mathit{algOrn}'\; O\; \phi} \arrow[u, bend left, "\Re^{-1}"]
\end{tikzcd}$$

Figure 1. (Commutative) diagram of ornament-induced maps and the isomorphism maps iso_1 and iso_2.

In the diagram there is a path of maps along which we can take a realisability proof for $\mathit{algOrn}'\; O\; \phi$ to one for O: Starting from a composite realisability proof

$$r : [\,jk\,]\; x' \Vdash \mathit{algOrn}'\; O\; \phi\ ,$$

the term

$$r_1 = \Re\; O\; (\mathit{forget}\; (\mathit{diffOrn\text{-}l}\; O\; \phi)\; (\mathit{iso}_1\; O\; \phi \\ (\Re^{-1}\; (\mathit{algOrn}'\; O\; \phi)\; x'\; r)))$$

is the desired left-component realisability proof, but its type again needs tweaking. Its original type is

$$[\,\mathsf{proj}_1\, jk\,] \\ \mathit{forget}\; O\; (\mathit{forget}\; (\mathit{diffOrn\text{-}l}\; O\; \phi)\; (\mathit{iso}_1\; O\; \phi \\ (\Re^{-1}\; (\mathit{algOrn}'\; O\; \phi)\; x'\; r))) \Vdash O\ ,$$

while our goal type is

$$[\,\mathsf{proj}_1\, jk\,]\; x' \Vdash O\ .$$

The composition of the two *forget*s and iso_1, however, can be reduced to just one big $\mathit{forget}\; (\mathit{algOrn}'\; O\; \phi)$. This can be proved by two applications of fold fusion [5], and ultimately reduces to naturality [16] of the underlying restructuring maps — *erase* and $\mathit{iso}_1\text{-}\mathit{cast}$ — and the fact that

$$\mathit{erase}\; O\; (\mathit{erase}\; (\mathit{diffOrn\text{-}l}\; O\; \phi)\; (\mathit{iso}_1\text{-}\mathit{cast}\; O\; \phi\; \mathit{xs})) \\ \equiv \mathit{erase}\; (\mathit{algOrn}'\; O\; \phi)\; \mathit{xs}$$

holds for all *xs*, which can be easily proved by induction on O. Thus the type we are left with is

$$[\,\mathsf{proj}_1\, jk\,]\; \mathit{forget}\; (\mathit{algOrn}'\; O\; \phi)\; (\Re^{-1}\; (\mathit{algOrn}'\; O\; \phi)\; x'\; r) \Vdash O\ ,$$

and *realiser-recovery* says that the realiser is just x'. Thus we reach our goal type and finish the implementation of the left projection.

As for the right projection, after defining the right difference ornament,

$$\begin{array}{l}
\mathit{diffOrn\text{-}r} : \forall\ \{I\; J\; K\; e\}\ \{D : \mathsf{Desc}\; I\} \\
\qquad (O : \mathsf{Orn}\; J\; e\; D)\ (\phi : [\![D]\!]\; K \Rightarrow K) \to \\
\qquad \mathsf{Orn}\; (\Sigma\, J\; (K \cdot e))\; (e \times \mathit{id})\; \lfloor \mathit{algOrn}\; D\; \phi \rfloor \\
\mathit{diffOrn\text{-}r}\ (\mathsf{say}\; (\mathsf{ok}\; j))\ \phi\ =\ \mathsf{say}\; (\mathsf{ok}\; (j, \mathsf{refl})) \\
\mathit{diffOrn\text{-}r}\ (\sigma\; S\; O) \qquad \phi\ =\ \sigma\; S\; \lambda s \mapsto \mathit{diffOrn\text{-}r}\; (O\; s)\; (\Lambda\; \phi\; s) \\
\mathit{diffOrn\text{-}r}\ \{K = K\}\ \{e\}\ (\mathsf{ask}\; (\mathsf{ok}\; j) * O)\ \phi = \\
\qquad \sigma\; (K\; (e\; j))\; \lambda k \mapsto \mathsf{ask}\; (\mathsf{ok}\; (j, k)) * \mathit{diffOrn\text{-}r}\; O\; (\Lambda\; \phi\; k) \\
\mathit{diffOrn\text{-}r}\ (\Delta\; S\; O) \qquad \phi\ =\ \Delta\; S\; \lambda s \mapsto \mathit{diffOrn\text{-}r}\; (O\; s)\; \phi\ ,
\end{array}$$

the implementation is completely symmetric and is omitted here.

Integration. Now we look at integration:

$$\begin{array}{l}
\mathit{integrate} : \\
\quad \forall\ \{I\; J\; K\; e\}\ \{D : \mathsf{Desc}\; I\}\ (O : \mathsf{Orn}\; J\; e\; D)\ (\phi : [\![D]\!]\; K \Rightarrow K) \\
\quad \{jk\}\ \{x' : \mu\; D\; (e\; (\mathsf{proj}_1\; jk))\} \to \\
\quad (r_1 : [\,\mathsf{proj}_1\, jk\,]\; x' \Vdash O)\ (r_2 : [\,(e \times \mathit{id})\; jk\,]\; x' \Vdash \mathit{algOrn}\; D\; \phi) \to \\
\qquad [\,jk\,]\; x' \Vdash \mathit{algOrn}'\; O\; \phi\ .
\end{array}$$

Had we considered general ornament composition, integration would have been much harder to implement, because ingredients from both component realisability proofs are essential and really need to be integrated by hard work. But since we are considering algebraic ornament-ornaments, the left difference ornament is algebraic and thus induces a remembering map, completing a path of maps along which we can smuggle a realisability proof for O as one for $\mathit{algOrn}'\; O\; \phi$ — again see Figure 1. (A realisability proof for $\mathit{algOrn}\; D\; \phi$ is nevertheless still needed, which provides information about the index, as we will see later.) Starting from the left-component realisability proof

$$r_1 : [\,\mathsf{proj}_1\, jk\,]\; x' \Vdash O\ ,$$

the composite realisability proof we deliver is

$$r = \Re\; (\mathit{algOrn}'\; O\; \phi)\; (\mathit{iso}_2\; O\; \phi \\ (\mathit{remember}\; \lfloor O \rfloor\; (\phi \cdot \mathit{erase}\; O)\; (\Re^{-1}\; O\; x'\; r_1)))\ ,$$

which has type

$$[\,\mathsf{proj}_1\, jk, \mathit{fold}\; (\phi \cdot \mathit{erase}\; O)\; (\Re^{-1}\; O\; x'\; r_1)\,] \\ \mathit{forget}\; (\mathit{algOrn}'\; O\; \phi)\; (\mathit{iso}_2\; O\; \phi \\ (\mathit{remember}\; \lfloor O \rfloor\; (\phi \cdot \mathit{erase}\; O)\; (\Re^{-1}\; O\; x'\; r_1))) \Vdash \mathit{algOrn}'\; O\; \phi\ ,$$

while our goal type is

$$[\,jk\,]\; x' \Vdash \mathit{algOrn}'\; O\; \phi\ .$$

Comparing the two types, we see that we need to establish two equalities,

$$\mathit{fold}\; (\phi \cdot \mathit{erase}\; O)\; (\Re^{-1}\; O\; x'\; r_1) \equiv \mathsf{proj}_2\; jk \tag{3}$$

and

$$\mathit{forget}\; (\mathit{algOrn}'\; O\; \phi)\; (\mathit{iso}_2\; O\; \phi \\ (\mathit{remember}\; \lfloor O \rfloor\; (\phi \cdot \mathit{erase}\; O)\; (\Re^{-1}\; O\; x'\; r_1))) \equiv x'\ . \tag{4}$$

The left-hand side of the first equality (3) looks like the left-hand side of *realiser-recovery*, but instead of $\mathit{fold}\; (\phi \cdot \mathit{erase}\; O)$ what we wish to see is $\mathit{forget}\; O$. Nevertheless, we see that $\mathit{fold}\; (\phi \cdot \mathit{erase}\; O)$ is just $\mathit{fold}\; \phi$ lifted to work with $\mu\; \lfloor O \rfloor$, so we can perform fission [8] — the conceptual opposite of fusion — by proving that

$$\mathit{fold}\; (\phi \cdot \mathit{erase}\; O)\; x \equiv \mathit{fold}\; \phi\; (\mathit{forget}\; O\; x) \qquad \text{for all } x\ .$$

Hence the $\mathit{forget}\; O$ coming out of the fission cancels out the $\Re^{-1}$ by *realiser-recovery*, reducing (3) to

$$\mathit{fold}\; \phi\; x' \equiv \mathsf{proj}_2\; jk\ . \tag{5}$$

This is where we need a realisability proof for $\mathit{algOrn}\; D\; \phi$. In the beginning we were also given the right-component realisability proof

$$r_2 : [\,(e \times \mathit{id})\; jk\,]\; x' \Vdash \mathit{algOrn}\; D\; \phi\ .$$

Notice that r_2 is a realisability proof for an *algebraic* ornament, so it can be transformed by *AOE* to a proof of an equality, which is exactly (5). So the first equality is successfully discharged. As for the second equality (4), we perform fission again to exchange the big $\mathit{forget}\; (\mathit{algOrn}'\; O\; \phi)$ composed with iso_2 for two smaller *forget*s, one of which cancels out the *remember*. The equality is thus reduced to

$$\mathit{forget}\ O\ (\mathfrak{R}^{-1}\ O\ x'\ r_1) \equiv x'\ ,$$

which is just an instance of *realiser-recovery*.

Example. Consider the function

$$\begin{array}{ll}
\mathit{insert} : \mathit{Nat} \to \mathit{List}\ \mathit{Nat} \to \mathit{List}\ \mathit{Nat} & \\
\mathit{insert}\ y\ \langle \mathsf{false}, \mathsf{refl} \rangle = y :: [\,] & \\
\mathit{insert}\ y\ \langle \mathsf{true}, x, \mathit{xs}, \mathsf{refl}\rangle & \textbf{with}\ y \leq_? x \\
\ldots & \mid \mathsf{yes}\ \_ = y :: x :: \mathit{xs} \\
\ldots & \mid \mathsf{no}\ \ \_ = x :: \mathit{insert}\ y\ \mathit{xs}\ ,
\end{array}$$

which is used, for example, in insertion sort. (The function $\_{\leq_?}\_$ compares two natural numbers, returning as a result either yes *eq* or no *neq* where *eq* and *neq* are proof terms justifying the result. Neither of the two proof terms is used in this basic version of *insert*, however.) We know that *insert y xs* has one more element than *xs*, i.e., we can prove

$$\begin{array}{l}
\mathit{insert\text{-}length} : \\
\quad \forall\ y\ \mathit{xs}\ \{n\} \to \mathit{length}\ \mathit{xs} \equiv n \to \mathit{length}\ (\mathit{insert}\ y\ \mathit{xs}) \equiv \mathit{suc}\ n\ .
\end{array}$$

This is the realisability proof for upgrading *insert* to work with vectors, i.e., to the function

$$\mathit{vinsert} : \mathit{Nat} \to \forall\ \{n\} \to \mathit{Vec}\ \mathit{Nat}\ n \to \mathit{Vec}\ \mathit{Nat}\ (\mathit{suc}\ n)\ .$$

Also we know that *insert* produces a sorted list if the input list is sorted, i.e., we can prove

$$\begin{array}{l}
\mathit{insert\text{-}sorted} : \\
\quad \forall\ y\ \mathit{xs}\ \{b\} \to \mathit{Sorted}\ b\ \mathit{xs} \to \mathit{Sorted}\ (b \sqcap y)\ (\mathit{insert}\ y\ \mathit{xs})
\end{array}$$

where $b \sqcap y$ is the minimum of b and y. Again this serves as a realisability proof for upgrading *insert* to work with sorted lists, i.e., to the function

$$\mathit{sinsert} : (y : \mathit{Nat}) \to \forall\ \{b\} \to \mathit{SList}\ b \to \mathit{SList}\ (b \sqcap y)\ .$$

Now suppose we wish to upgrade it to work with sorted vectors,

$$\begin{array}{l}
\textbf{data}\ \mathsf{SVec} : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Set}\ \textbf{where} \\
\quad \mathsf{nil}\quad\ : \forall\ \{b\} \to \mathsf{SVec}\ b\ \mathsf{zero} \\
\quad \mathsf{cons}\ : (x : \mathsf{Nat}) \to \forall\ \{b\} \to b \leq x \to \\
\qquad\qquad \forall\ \{n\} \to \mathsf{SVec}\ x\ n \to \mathsf{SVec}\ b\ (\mathsf{suc}\ n)\ ,
\end{array}$$

which is described by the ornament

$$\begin{array}{lcl}
\mathit{SVecO} & : & \mathsf{Orn}\ (\mathit{Nat} \times \mathit{Nat})\ !\ \lfloor \mathit{ListO}\ \mathit{Nat} \rfloor \\
\mathit{SVecO} & = & \mathit{algOrn}'\ \mathit{SListO}\ (\mathit{ornAlg}\ (\mathit{ListO}\ \mathit{Nat}))\ .
\end{array}$$

This time, however, we do not need to prove repetitively and monolithically that *insert y xs* is sorted and has length *suc n* if *xs* is sorted and has length *n*; instead, we can reuse *insert-length* and *insert-sorted* with the help of *project-l*, *project-r*, and *integrate*. The function we wish to write is

$$\begin{array}{l}
\mathit{svinsert} : \\
\quad (y : \mathit{Nat}) \to \forall\ \{b\ n\} \to \mathit{SVec}\ b\ n \to \mathit{SVec}\ (b \sqcap y)\ (\mathit{suc}\ n)\ .
\end{array}$$

Assume that $y : \mathit{Nat}$ and $\mathit{xs} : \mathit{SVec}\ b\ n$ are given. We invoke the inverse realisability transformation and supply $\mathit{insert}\ y\ \mathit{xs}'$, where $\mathit{xs}' = \mathit{forget}\ \mathit{SVecO}\ \mathit{xs}$, as the realiser, and we need to produce a corresponding realisability proof of type $\mathit{insert}\ y\ \mathit{xs}' \Vdash \mathit{SVecO}$ from a realisability proof of type $\mathit{xs}' \Vdash \mathit{SVecO}$. Since *SVecO* is a composite ornament, we can break the given composite realisability proof into two component proofs with *project-l* and *project-r*, use them to build two required component proofs independently, and *integrate* the two independently built proofs to get the required composite proof. The program is shown below.

$$\begin{array}{l}
\mathit{svinsert} : (y : \mathit{Nat}) \to \forall\ \{b\ n\} \to \mathit{SVec}\ b\ n \to \mathit{SVec}\ (b \sqcap y)\ (\mathit{suc}\ n) \\
\mathit{svinsert}\ y\ \mathit{xs} = \mathfrak{R}^{-1}\ \mathit{SVecO}\ (\mathit{insert}\ y\ \mathit{xs}') \\
\quad (\mathit{integrate}\ \mathit{SListO}\ \phi \\
\qquad (\mathit{insert\text{-}sorted}\ y\ \mathit{xs}'\ r_1) \\
\qquad (\mathit{AOE}^{-1}\ \lfloor \mathit{ListO}\ \mathit{Nat} \rfloor\ \phi \\
\qquad\quad (\mathit{insert\text{-}length}\ y\ \mathit{xs}'\ (\mathit{AOE}\ \lfloor \mathit{ListO}\ \mathit{Nat} \rfloor\ \phi\ r_2)))) \\
\quad \textbf{where}\ \begin{array}[t]{lcl}
\mathit{xs}' & = & \mathit{forget}\ \mathit{SVecO}\ \mathit{xs} \\
\phi & = & \mathit{ornAlg}\ (\mathit{ListO}\ \mathit{Nat}) \\
r & = & \mathfrak{R}\ \mathit{SVecO}\ \mathit{xs} \\
r_1 & = & \mathit{project\text{-}l}\ \mathit{SListO}\ \phi\ r \\
r_2 & = & \mathit{project\text{-}r}\ \mathit{SListO}\ \phi\ r
\end{array}
\end{array}$$

5. Discussion

The realisability interpretation in fact works for *general* algebraic ornaments, ornamental-algebraic ornaments being a special case: Given a description $D : \mathsf{Desc}\ I$ and an algebra $\phi : [\![D]\!]\ J \Rightarrow J$, the type $\mu\ D$ is interpreted as the complete type, J as the realiser type, and $\mu\ \lfloor \mathit{algOrn}\ D\ \phi \rfloor$ as the realisability predicate. Assuming that $x : \mu\ D\ i$ is a complete object, the type of *remember* says that $\mathit{fold}\ \phi\ x : J\ i$ satisfies the realisability predicate, so *remember* is the realisability transformation, while the inverse transformation is *forget*. $\mathfrak{R}$ and $\mathfrak{R}^{-1}$ are just *remember* and *forget* specialised for ornamental algebras. The reason we introduced the realisability transformation based on ornaments instead of algebras is that ultimately we use the transformation to talk about ornament composition. It is convenient to have the intuition that every ornament expresses the relationship between a realiser type and a complete type and induces a corresponding realisability predicate. Subsequently, composing ornaments gives rise to a new and richer complete type, and the induced realisability predicate can be decomposed into realisability predicates for the component ornaments. Algebra-based interpretation does not offer this intuition, because algebras do not compose: For example, we can fold both a list and a tree to a natural number, say computing the number of elements, but it is not obvious what composite datatype would arise in this situation.

More importantly, introducing the realisability interpretation in terms of ornamental-algebraic ornaments brings out the correspondence between internalism and externalism regarding constraint composition. Under the realisability view, data and constraints are separated into realisers and realisability predicates. This is exactly externalism — realisers do not carry with them proofs that they are indeed realisers. Multiple constraints simply correspond to multiple realisability predicates applied to the same piece of data. For internalism, constraints are encoded in ornaments, and to express multiple constraints we use ornament composition. The realisability transformation points out the correspondence between the two different ways of expressing constraints — ornaments for internalism and realisability predicates for externalism: An ornament *induces* a realisability predicate, which is the manifestation of the ornament in the world of decoded datatypes, and moreover, composition of realisability predicates mirrors composition of ornaments. A bridge is thus formed between externalism and internalism, and subsequently, externalist modularity is brought into internalist datatypes.

It is worth noting that upgrading a function using the realisability transformation does not really exempt us from reimplementing the logic. For example, when we upgrade *insert* to work with sorted lists, the realisability proof we need to supply is *insert-sorted*, which takes one *Sorted*ness proof and produces another. *Sorted* being isomorphic to *SList*, implementing *insert-sorted* is not so different from reimplementing *insert* for sorted lists. So what is the difference? Let us temporarily change our perspective and consider how we might synthesise *svinsert* from *sinsert* and *vinsert*, without the help of the realisability transformation. We would get a sorted list and a vector from the input sorted vector, feed them to *sinsert* and *vinsert* separately, and combine the outputs to get a sorted vector as the final result. The main obstacle is that we cannot freely integrate a sorted list with a vector to get a sorted vector, because the underlying list of the sorted list may not be the same as that of the vector. If we are able to guarantee that the sorted list and the vector have the same underlying list, however, then the integration goes through, but it is awkward to express the guarantee. It is by employing realisability predicates that this awkwardness can be overcome. A realisability predicate exposes the underlying data in the index, so by taking proofs of realisability predicates *applied to the same index*, our *integrate* function gets precisely the guarantee that it needs. The ability to express the guarantee in this elegant manner is a demonstration of the strength of internalism. Thus the use of realisability predicates, which is central to externalist compositionality, can in fact be regarded as an application of an internalist technique to solve the compositionality problem of internalist datatypes.

Practically, how do we structure our libraries with the realisability transformation for better reusability? As McBride suggested, the datatypes should be delivered as codes and ornaments. The datatypes on which an operation is defined should be as general as possible, and other versions of the operation on more specialised types should be implemented in the form of realisability proofs. For example, *insert* should be defined for plain lists, and implemented for sorted lists and vectors as a function on sortedness proofs and length equalities respectively. Delivered in this way, then, *insert* for sorted lists, vectors, and sorted vectors can all be derived routinely by the realisability transformation as we have seen. This is the reusability and modularity offered by externalism. On the other hand, some operations are best defined on more specialised types, so preconditions can be cleanly expressed and manipulated. An example is the safe lookup function

```
lookup : {A : Set} → ∀ {n} → Fin n → Vec A n → A
lookup fzero    (x :: xs) = x
lookup (fsuc i) (x :: xs) = lookup i xs .
```

It is natural to define this function on vectors (instead of lists) and use Fin (instead of Nat) as the index type, as the length constraint is embedded in the indices of the types of the data and requires no extra management, which is the advantage offered by internalism. So here is the development pattern we have in mind: Once a rich collection of ornaments are provided, programmers will have the freedom to choose which constraints they wish to impose on a basic type, compose the relevant ornaments and decode the composite ornament to a suitable inductive family T. Existing operations are upgraded to work with T routinely by the realisability transformation. And then, operations specific to T can be programmed directly on T, benefiting from the precision and convenience of programming with inductive families.

6. Related work

Section 2 is a faithful albeit condensed summary of McBride's original implementation of ornaments [11], except for a few notational changes. Our work is heavily based on algebraic ornaments and the associated construction method. Ornamental-algebraic ornaments have already appeared in McBride's original paper, and in particular, the *Length* predicate was derived from the ornament *VecO A*, which was one of our motivating examples. Also, a variant of less-than-or-equal-to relation on natural numbers was derived using an algebraic ornament by McBride, which led us to notice the similarity between Fin and _>_.

The idea of viewing vectors as realisability predicates was proposed by Bernardy [3, p 82], which refers to the realisability transformation defined for pure type systems by Bernardy and Lasson [4]. He started with the list type in which the element-type parameter is marked as "first-level," whereas the list type itself is "second-level." Applying the "projecting transformation," which removes first-level terms and demotes second-level terms to first-level, the second-level type of lists is transformed to the first-level type of natural numbers. And then, applying their realisability transformation, the list type is transformed to a second-level vector type indexed by first-level natural numbers. Our realisability interpretation can be seen as a translation of his idea into the language of ornaments without introducing levels: Our notion of complete objects and types would be second-level in Bernardy's system, while realisers and their types would be first-level. When applied to programs, their projecting transformation corresponds to our ornamental forgetful map. Due to the syntax-generic character of his transformations, Bernardy was able to derive vector append effortlessly from list append, and in particular deduce that, in the type of vector append, the index of the resulting vector is the sum of the indices of the two input vectors, because natural number addition is the (functional) realiser extracted from list append. Extraction of functional realisers from complete functions is not, and should not be, possible in our framework, however: The behaviour of a function taking a complete object may depend essentially on the added information, which is not available to a function taking only a realiser. For example, a function of type *List Nat* → *List Nat* may be defined to compute the sum s of the input list and emit a list of length s whose elements are all zero. We cannot hope to write a function of type *Nat* → *Nat* that reproduces the corresponding behaviour on natural numbers. On the other hand, it is reasonable to project list append to natural number addition, because list append is polymorphic and cannot inspect the elements. Indeed, in Bernardy and Lasson's system, it is impossible to produce second-level terms by induction on first-level terms, as the first-level terms are designed to be "computationally irrelevant" to second-level terms. This could be overcome by, for example, employing singleton types [12] to link different levels, but it can be inconvenient to do so explicitly. Our framework does not embody computational irrelevance, and trades the ability to derive polymorphic programs for simplicity and convenience.

The classic application of realisability in computing is program extraction, e.g., in Coq [15]. Terms are marked either as "informative" or "non-informative," and the non-informative terms, i.e., the proof terms irrelevant to computation, are removed during extraction, leaving the informative terms as the extracted program. It should be noted that our inverse transformation is not in general possible for other realisability systems, e.g., the one for the Calculus of Constructions in [15]. That is, it is not the case in general that having a realiser of a proposition implies that the proposition has a proof. Realisability in such systems can be used to show consistency of axioms — a proposition may not be provable, but can be postulated as an axiom consistently if it can be shown to be realisable. Our use of realisability terminology reflects that our development started from applying the notion to interpret ornamental-algebraic ornaments, but our development does not intend to follow faithfully those of the existing realisability theories and clearly deviates from those systems.

7. Future work

General ornament composition is a natural goal to pursue. A quick example that requires general ornament composition is *finite lists*, which are lists guaranteed to be shorter than a certain length:

```
data FList (A : Set) : Nat → Set where
  fnil  : ∀ {m} → FList (suc m)
  fcons : A → ∀ {m} → FList m → FList (suc m) .
```

The datatype comes out of composing the ornaments *ListO A* and *FinO*, neither of which is algebraic. One particular difficulty we encounter when trying to define general ornament composition is that the new index set is a pullback, which is awkward to deal with. Also the implementation of *integrate* for general ornament composition is conceivably more complex. These should just be technical difficulties, though, and do not seem to detract from the feasibility of general ornament composition.

Before we commit ourselves to the implementation of general ornament composition, we may first consider increasing the expressive power of datatype descriptions and ornaments. For example, to define sorted lists without also indexing the type with a lower bound requires induction-induction [13]:

```
mutual
  data SList′ : Set where
    snil′  : SList′
    scons′ : (x : Nat) (xs : SList′) → x ≼ xs → SList′
  data _≼_ (y : Nat) : SList′ → Set where
    nil  : y ≼ snil′
    cons : ∀ {x xs b} → y ≤ x → y ≼ scons′ x xs b .
```

To talk about this and other similar datatypes, first we need to expand the universe to include codes for datatypes defined by induction-induction (or induction-recursion [7]). Another example is lists indexed with one of their prefixes:

```
data PList (A : Set) : List A → Set where
  pnil      : PList []
  pcons-[]  : (x : A) → ∀ {xs} → PList xs → PList []
  pcons-::  : (x : A) → ∀ {xs} → PList xs → PList (x :: xs) .
```

It is possible to use the ornament

```
PListO : (A : Set) → Orn (List A) ! ⌊ListO A⌋
PListO A =
  σ Bool (false ↦ say (ok [])
          true ↦  σ A λx ↦ Δ (List A) λxs ↦
                    ask (ok xs) ∗
                      Δ Bool (false ↦ say (ok [])
                              true ↦  say (ok (x :: xs))))
```

which, in the cons case, inserts a boolean just before saying the index, which can be either [] or $x :: xs$, depending on the boolean. However, it is desirable to make the ornament reflect the fact that the native declaration has three constructors rather than two. To do so, we need to be able to refine the type Bool for the outermost σ to some three-element type. This requires expansion of the ornament language.

As with McBride's implementation of ornaments, we implement the realisability transformation in Agda just for experimenting with the idea, and do not intend to actually structure Agda programs with the combinators. To make the realisability transformation practically usable, it may have to be built into the language (along with ornaments) and supported by the development environment, allowing, e.g., automatic insertion of the transformation and inference of the datatype-generic parameters, or at least providing specific interactive commands to invoke the transformation, so the programmer need not bother with the details.

Theoretically, we may wish to get rid of the implementation details of datatype descriptions and ornaments, and examine all the concepts in terms of a cleaner mathematical semantics, like the one presented by Atkey, Johann, and Ghani [2]. Ornaments themselves now have an interesting compositional structure, so it is possible to develop an algebra of ornaments. Moreover, the correspondence between ornaments and realisability predicates looks like a subject ideally deserving a categorical treatment. We hope that our work will someday find a counterpart in the mathematical theory of datatypes, so it can be better characterised and understood.

Acknowledgements

We would like to thank Pierre-Évariste Dagand for referring us to Bernardy's idea, Shin-Cheng Mu for having the first discussion on this work with the first author, Liang-Ting Chen for suggesting the example of prefix-indexed lists, Jean-Philippe Bernardy and Fredrik Nordvall Forsberg for providing invaluable comments, and especially Conor McBride for sharing with us in the first place his unpublished work on ornaments. Meetings of the *Reusability and Dependent Types* project and *Algebra of Programming* research group greatly helped the development of our ideas. The first author is supported by the University of Oxford Clarendon Fund Scholarship, and both authors by the UK Engineering and Physical Sciences Research Council project *Reusability and Dependent Types*.

References

[1] T. Altenkirch and C. McBride. Generic programming within dependently typed programming. In *IFIP TC2/WG2.1 Working Conference on Generic Programming*, pages 1–20. Kluwer, B.V., 2003.

[2] R. Atkey, P. Johann, and N. Ghani. When is a type refinement an inductive type? In M. Hofmann, editor, *Foundations of Software Science and Computational Structures*, volume 6604 of *Lecture Notes in Computer Science*, pages 72–87. Springer-Verlag, 2011.

[3] J.-P. Bernardy. *A Theory of Parametric Polymorphism and an Application*. PhD thesis, Chalmers University of Technology, 2011.

[4] J.-P. Bernardy and M. Lasson. Realizability and parametricity in pure type systems. In M. Hofmann, editor, *Foundations of Software Science and Computation Structures*, volume 6604 of *Lecture Notes in Computer Science*, pages 108–122. Springer-Verlag, 2011.

[5] R. Bird and O. de Moor. *Algebra of Programming*. Prentice-Hall, 1997.

[6] J. Chapman, P.-É. Dagand, C. McBride, and P. Morris. The gentle art of levitation. In *International Conference on Functional Programming*, ICFP '10, pages 3–14. ACM, 2010.

[7] P. Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *Journal of Symbolic Logic*, 65(2):525–549, June 1998.

[8] J. Gibbons. Fission for program comprehension. In T. Uustalu, editor, *Mathematics of Program Construction*, volume 4014 of *Lecture Notes in Computer Science*, pages 162–179. Springer-Verlag, July 2006.

[9] S. C. Kleene. On the interpretation of intuitionistic number theory. *Journal of Symbolic Logic*, 10(4):109–124, December 1945.

[10] P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, Napoli, 1984.

[11] C. McBride. Ornamental algebras, algebraic ornaments. To appear in *Journal of Functional Programming*.

[12] S. Monnier and D. Haguenauer. Singleton types here, singleton types there, singleton types everywhere. In *Programming Languages meets Program Verification*, PLPV '10, pages 1–8. ACM, 2010.

[13] F. Nordvall Forsberg and A. Setzer. Inductive-inductive definitions. In A. Dawar and H. Veith, editors, *Computer Science Logic*, volume 6247 of *Lecture Notes in Computer Science*, pages 454–468. Springer-Verlag, 2010.

[14] U. Norell. Dependently typed programming in Agda. In P. Koopman, R. Plasmeijer, and D. Swierstra, editors, *Advanced Functional Programming (AFP 2008)*, volume 5832 of *Lecture Notes in Computer Science*, pages 230–266. Springer-Verlag, 2009.

[15] C. Paulin-Mohring. Extracting F_ω's programs from proofs in the Calculus of Constructions. In *Principles of Programming Languages*, pages 89–104. ACM, Jan. 1989.

[16] P. Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture*, pages 347–359. ACM, 1989.