Under consideration for publication in J. Functional Programming

Ornamental Algebras, Algebraic Ornaments

CONOR McBRIDE
Department of Computer and Information Sciences
University of Strathclyde
Glasgow, Scotland
(*e-mail:* conor@cis.strath.ac.uk)

Abstract

This paper re-examines the presentation of datatypes in dependently typed languages, addressing in particular the issue of what it means for one datatype to be in various ways more informative than another. Informal human observations like 'lists are natural numbers with extra labels' and 'vectors are lists indexed by length' are expressed in a first class language of *ornaments*—presentations of fancy new types based on plain old ones.

Each ornament adds information, so it comes with a forgetful function from fancy data back to plain, expressible as the fold of its *ornamental algebra*: lists built from numbers acquire the 'length' algebra. Conversely, each algebra for a datatype induces a way to index it—an *algebraic ornament*. The length algebra for lists induces the construction of the paradigmatic dependent vector types.

Dependent types thus provide not only a new 'axis of diversity'—indexing—for data structures, but also new abstractions to manage and exploit that diversity. In the new programming (2), coincidence is replaced by consequence.

1 Introduction

If it's not a strange question, where do datatypes come from? Most programming languages allow us to *declare* datatypes — that is to say, datatypes come from *thin air*. Programs involving the data thus circumscribed subsequently become admissible, and if we are fortunate, we may find that some of these programs are amongst those that we happen to want. What an outrageous coincidence!

In dependently typed programming languages, the possible variations of datatypes are still more dense and subtle, and the coincidences all the more outrageous. For example, if I have list types,

$$\begin{array}{l}\textbf{data}\ \mathsf{List}\,(X : \mathsf{Set}) : \mathsf{Set}\ \textbf{where}\\ \quad [] : \mathsf{List}\,X\\ \quad \_::\_ : X \to \mathsf{List}\,X \to \mathsf{List}\,X\end{array}$$

I might become frustrated by error cases in my attempts to define zip.

$$\begin{array}{l}\mathsf{zip} : \forall\{X\,Y\} \to \mathsf{List}\,X \to \mathsf{List}\,Y \to \mathsf{List}\,(X \times Y)\\ \mathsf{zip}\quad []\quad\quad []\quad\quad = []\\ \mathsf{zip}\quad []\quad (y :: ys) = ?\\ \mathsf{zip}\,(x :: xs)\quad []\quad\quad = ?\\ \mathsf{zip}\,(x :: xs)\,(y :: ys) = (x, y) :: \mathsf{zip}\,xs\,ys\end{array}$$

Perhaps it will occur to me to declare *vectors* instead.

```
data Vec (X : Set) : Nat → Set where
  [] : Vec X zero
  _::_ : ∀{n} → X → Vec X n → Vec X (suc n)
```

As it happens, simple unification constraints on indices rule out the error cases and allow us stronger guarantees about the valid cases:

```
zip : ∀{n X Y} → Vec X n → Vec Y n → Vec (X × Y) n
zip    []        []     = []
zip (x :: xs) (y :: ys) = (x , y) :: zip xs ys
```

These vectors may look a little strange, but perhaps they are in some way related to lists. Do you think it might be so? Could we perhaps write functions to convert between the two

```
vecList : ∀{n X} → Vec X n → List X
listVec : ∀{X} → (xs : List X) → Vec X (f xs)
```

for a suitable `f : ∀{X} → List X → Nat`? What might `f` be? I know a function in the library with the right type: perhaps `length` will do.

But I am being deliberately obtuse. Let us rather be acute. These vectors were conceived as a fancy version of lists, so we should not be surprised that there is a forgetful function which discards the additional indexing. Further, the purpose of indexing vectors is to expose length in types, so it is not a surprise that this index can be computed by the `length` function. Indeed, it took an act self-censorship not to introduce vectors to you in prose, 'Vectors are lists indexed by their length.', but rather just to declare them to you, as I might to a computer.

In this paper, I show how one might express this prose introduction to a computer, *constructing* the vectors from the definition of `length` in such a way as to guarantee their relationship with lists. The key is to make the definitions of datatypes first-class citizens of the programming language by establishing a datatype of datatype descriptions. This gives us the means to frame the question of what it is for one datatype to be a 'fancy version' of another. I shall introduce the notion of an *ornament* on a datatype, combining refinement of indexing and annotation with additional data. Ornaments, too, are first-class citizens — we can and will compute them systematically.

This technology allows us not only to express vectors as an ornament on lists, but lists themselves as an ornament on numbers. Moreover, the former can be seen as a consequence of the latter.

2 Describing Datatypes

In order to manipulate inductive (tree-like) datatypes, we shall need to represent their *descriptions* as data, then interpret those descriptions as types. That is, we must construct what Martin-Löf calls a *universe*. The techniques involved here are certainly not new: we can follow the recipe from Peter Dybjer and Anton Setzer's coding of induction-recursion (1), suitably adapted to the present purpose.

Let us start with plain unindexed first-order data structures, just to get the idea. You can interpret a `Plain` description as a format, or a program for reading a record corresponding

to one node of the tree.

```
data PlainDesc : Set₁ where
  arg : (A : Set) → (A → PlainDesc) → PlainDesc   — read field in A; continue, given its value
  rec : PlainDesc → PlainDesc                     — read a recursive subnode; continue regardless
  ret : PlainDesc                                 — stop reading
```

Note that arg behaves like a binding operator, so I shall sometimes write it as one,

$$\mathsf{arg}\, x : X \,.\, D \qquad \text{sugars} \qquad \mathsf{arg}\, X\, (\lambda x \,.\, D)$$

Let us have an example description: if we have a two element enumeration to act as a set of tags, we may describe the natural numbers. See how the dependency built into the arg construct allows us to treat 'constructor choice' as just another argument.

```
NatPlain : PlainDesc
NatPlain ↦ arg c : {zz, ss} . case c of
                                zz ↦ ret
                                ss ↦ rec ret
```

If we know the type R of recursive subnodes, we can interpret a description D as the record type which describes a D-node. We may then take a fixpoint, PlainData D, instantiating R with PlainData D itself.

```
⟦·⟧ : PlainDesc → Set → Set
⟦arg A D⟧ R ↦ (a : A) × ⟦D a⟧ R
⟦rec D⟧ R   ↦ R × ⟦D⟧ R
⟦ret⟧ R     ↦ 1

data PlainData (D : PlainDesc) : Set where
  ⟨·⟩ : ⟦D⟧ (PlainData D) → PlainData D
```

If we work through our natural number example, we find

```
Nat : Set
Nat ↦ PlainData NatPlain

zero : Nat          |  suc : Nat → Nat
zero ↦ ⟨zz, ⋆⟩      |  suc n ↦ ⟨ss, n, ⋆⟩
```

```
⟦NatPlain⟧ Nat ≡ (c : {zz, ss}) × case c of
                                    zz ↦ 1
                                    ss ↦ Nat × 1
```

Now, let us add indexing, giving a type which describes inductive definitions in $I \to$ Set. All that changes is that we must specify an index anytime we ask for a subnode or deliver a node.

```
data Desc (I : Set) : Set₁ where
  arg : (A : Set) → (A → Desc I) → Desc I   — read field in A; continue, given its value
  rec : I → Desc I → Desc I                 — read an indexed subnode; continue regardless
  ret : I → Desc I                          — stop reading and return the node's index
```

We may readily port our plain example, indexing with 1 and inserting trivial indices where required:

$$\begin{array}{l}\mathsf{NatDesc} : \mathsf{Desc}\,1\\ \mathsf{NatDesc} \mapsto \mathsf{arg}\,c : \{\mathsf{zz},\mathsf{ss}\}.\ \mathbf{case}\,c\,\mathbf{of}\\ \qquad\qquad\qquad\qquad\qquad\qquad \mathsf{zz} \mapsto \mathsf{ret}\star\\ \qquad\qquad\qquad\qquad\qquad\qquad \mathsf{ss} \mapsto \mathsf{rec}\star(\mathsf{ret}\star)\end{array}$$

When we interpret descriptions, we're given the indexed family of recursive subnodes $R : I \to \mathsf{Set}$, and we must deliver an $I \to \mathsf{Set}$. In effect we receive an index which the node must return. Hence, let us interpret the ret construct with an equality constraint.

$$\begin{array}{l}[\![\cdot]\!] : \mathsf{Desc}\,I \to (I \to \mathsf{Set}) \to I \to \mathsf{Set}\\ [\![\mathsf{arg}\,A\,D]\!]\,R\,i \mapsto (a : A) \times [\![D\,a]\!]\,R\,i\\ [\![\mathsf{rec}\,h\,D]\!]\,R\,i \mapsto R\,h \times [\![D]\!]\,R\,i\\ [\![\mathsf{ret}\,o]\!]\,R\,i \mapsto o = i\end{array}$$

$$\begin{array}{l}\mathbf{data}\ \mathsf{Data}\,(D : \mathsf{Desc}\,I) : I \to \mathsf{Set}\ \mathbf{where}\\ \langle\_\rangle : [\![D]\!]\,(\mathsf{Data}\,D\,i) \to \mathsf{Data}\,D\,i\end{array}$$

Our example becomes

$$\begin{array}{l|l|l}\mathsf{Nat} : \mathsf{Set} & \mathsf{zero} : \mathsf{Nat} & \mathsf{suc} : \mathsf{Nat} \to \mathsf{Nat}\\ \mathsf{Nat} \mapsto \mathsf{Data}\,\mathsf{NatDesc}\star & \mathsf{zero} \mapsto \langle \mathsf{zz}, \mathsf{refl}\rangle & \mathsf{suc}\,n \mapsto \langle \mathsf{ss}, n, \mathsf{refl}\rangle\end{array}$$

However, we can also define nontrivial indexed structures, like the vectors:

$$\begin{array}{l}\mathsf{VecDesc} : \mathsf{Set} \to \mathsf{Desc}\,\mathsf{Nat}\\ \mathsf{VecDesc}\,X \mapsto \mathsf{arg}\,c : \{\mathsf{zz},\mathsf{ss}\}.\ \mathbf{case}\,c\,\mathbf{of}\\ \qquad\qquad\qquad\qquad\qquad\qquad \mathsf{zz} \mapsto \mathsf{ret}\,\mathsf{zero}\\ \qquad\qquad\qquad\qquad\qquad\qquad \mathsf{ss} \mapsto \mathsf{arg}\,\_ : X.\ \mathsf{arg}\,n : \mathsf{Nat}.\ \mathsf{rec}\,n\,(\mathsf{ret}\,(\mathsf{suc}\,n))\end{array}$$

$$\begin{array}{l}\mathsf{Vec} : \mathsf{Set} \to \mathsf{Nat} \to \mathsf{Set}\\ \mathsf{Vec}\,X\,n \mapsto \mathsf{Data}\,(\mathsf{VecDesc}\,X)\,n\end{array}$$

$$\begin{array}{l|l}\mathsf{nil} : \mathsf{Vec}\,X\,\mathsf{zero} & \mathsf{cons}_n : X \to \mathsf{Vec}\,X\,n \to \mathsf{Vec}\,X\,(\mathsf{suc}\,n)\\ \mathsf{nil} \mapsto \langle \mathsf{zz}, \mathsf{refl}\rangle & \mathsf{cons}_n\,x\,xs \mapsto \langle \mathsf{ss}, x, n, xs, \mathsf{refl}\rangle\end{array}$$

3 Map and Fold with Indexed Algebras

In this section, I shall lift the standard treatment of map and fold to indexed data structures, and then show how to implement them for the indexed datatypes described by $\mathsf{Desc}\,I$.

When presenting an inductive datatype as the least fixpoint

$$\mathsf{in} : F\,(\mu F) \to \mu F$$

of a suitable functor $F : \mathsf{Set} \to \mathsf{Set}$, we provide the action of F on functions, lifting operations on elements uniformly to operations on structures

$$\mathsf{map}_F : (X \to Y) \to F\,X \to F\,Y$$

and are rewarded with an iteration operator

$$\begin{aligned}&\mathsf{fold}_F : (F\,X \to X) \to \mu F \to X\\&\mathsf{fold}_F\,\phi\,(\mathsf{in}\,ds) \mapsto \phi\,(\mathsf{map}_f\,(\mathsf{fold}_F\,\phi)\,ds)\end{aligned}$$

everywhere replacing in by ϕ. We can think of F as a *signature*, describing how to build expressions from subexpressions, and μF as the syntactic datatype of expressions so generated. A function $\phi : F\,X \to X$ explains how to implement each expression-form in the signature for values drawn from X — we say that ϕ is an F-*algebra* with carrier X. If we know how to implement the signature, then we can evaluate expressions: that is exactly what fold_F does, first using map_F to evaluate the subexpressions, then applying ϕ to deliver the value of the whole.

We can play the same game with our functors on $I \to \mathsf{Set}$. We must first say what the arrows are — functions which *respect indexing*:

$$\begin{aligned}&\cdot \subseteq \cdot : (I \to \mathsf{Set}) \to (I \to \mathsf{Set}) \to \mathsf{Set}\\&X \subseteq Y \mapsto (i{:}I) \to X\,i \to Y\,i\end{aligned}$$

Let us now equip all our descriptions with functorial actions:

$$\begin{aligned}&\mathsf{map} : \mathsf{Desc}\,I \to (X \subseteq Y) \to [\![D]\!]\,X \subseteq [\![D]\!]\,Y\\&\mathsf{map}\,(\mathsf{arg}\,A\,D)\,f\,i\,(a,d) \mapsto a, \mathsf{map}\,(D\,a)\,f\,i\,d\\&\mathsf{map}\,(\mathsf{rec}\,h\,D)\;f\,i\,(x,d) \mapsto f\,h\,x, \mathsf{map}\,D\,f\,i\,d\\&\mathsf{map}\,(\mathsf{ret}\,o)\quad f\,i\,q\quad\;\; \mapsto q\end{aligned}$$

Let us abbreviate the type of $[\![D]\!]$-algebras still further, to eliminate the repetition of the carrier.

$$\mathsf{Alg} : \mathsf{Desc}\,I \to (I \to \mathsf{Set}) \to \mathsf{Set}\quad \mathsf{Alg}\,D\,X \mapsto [\![D]\!]\,X \subseteq X$$

Now, the iterator will take any index-respecting algebra and perform index-respecting evaluation of indexed expressions:

$$\begin{aligned}&\mathsf{fold} : \mathsf{Desc}\,I \to (\mathsf{Alg}\,D\,X) \to \mathsf{Data}\,D \subseteq X\\&\mathsf{fold}\,D\,\phi\,i\,\langle ds\rangle \mapsto \phi\,i\,(\mathsf{map}\,D\,(\mathsf{fold}\,D\,\phi)\,i\,ds)\end{aligned}$$

Lots of popular operations can be expressed as folds. For example, addition...

$$\begin{aligned}&\cdot + \cdot : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}\\&x + y \mapsto \mathsf{fold}\,\mathsf{NatDesc}\,(\mathsf{adda}\,y)\,\_\,x\end{aligned}$$

$$\begin{aligned}&\mathsf{adda} : \mathsf{Nat} \to \mathsf{Alg}\,\mathsf{NatDesc}\,(\lambda\_.\,\mathsf{Nat})\\&\mathsf{adda}\,y\,\_\,(\mathsf{zz},\mathsf{refl}) \mapsto y\\&\mathsf{adda}\,y\,\_\,(\mathsf{ss},sum,\mathsf{refl}) \mapsto \mathsf{suc}\,sum\end{aligned}$$

...and vector concatenation — note the careful abstraction of m to yield the carrier of the algebra

$$\begin{aligned}&\cdot +\!\!+^n_m \cdot : \mathsf{Vec}\,X\,m \to \mathsf{Vec}\,X\,n \to \mathsf{Vec}\,X\,(m+n)\\&xs +\!\!+^n_m ys \mapsto \mathsf{fold}\,\mathsf{VecDesc}\,(\mathsf{conca}_n\,ys)\,m\,xs\end{aligned}$$

$$\begin{aligned}&\mathsf{conca}_n : \mathsf{Vec}\,X\,n \to \mathsf{Alg}\,(\mathsf{VecDesc}\,X)\,(\lambda m.\,\mathsf{Vec}\,X\,(m+n)))\\&\mathsf{conca}_n\,ys\,\mathsf{zero}\quad(\mathsf{zz},\mathsf{refl}) \mapsto ys\\&\mathsf{conca}_n\,ys\,(\mathsf{suc}\,m)\,(\mathsf{ss},x,m,conc,\mathsf{refl}) \mapsto \mathsf{cons}_{m+n}\,x\,conc\end{aligned}$$

4 Ornaments and their Algebras

The point of this paper is to explore systematic transformations of datatypes and the computational structure thus induced. In this section, I shall introduce the idea of *ornamenting* an indexed data structure, combining the business of *decorating* a datatype with extra stored information with that of *refining* a datatype with a more subtle index structure.

Suppose we have some description $D : \mathsf{Desc}\, I$ of an I-indexed family of datatypes. Now suppose we come up with a more informative index set J, together with some function $e : J \to I$ which erases this richer information. Let us consider how we might develop a description $D' : \mathsf{Desc}\, J$ which is richer than D in an e-respecting way, so that we can always erase bits of a $\mathsf{Data}\, D'\, j$ to get an unadorned $\mathsf{Data}\, D\, (e\, j)$. For example, if we start with a plain $\mathsf{1}$-indexed family, then e can only be $!$, the terminal arrow which always returns $\star$.

How are we to build such a D' from D? Certainly, wherever D mentions indices $i : I$, D' will need an index j such that $e\, j = i$. It will help to define the *inverse image* of e, as follows:

$$\begin{array}{l}\mathbf{data}\ \mathsf{Inv}_J\ (e : J \to I) : I \to \mathsf{Set}\ \mathbf{where}\\ \quad \mathsf{inv} : (j : J) \to \mathsf{Inv}\, e\, (e\, j)\end{array}$$

That is to say, $\mathsf{inv}\, j : \mathsf{Inv}\, e\, i$ if and only if $e\, j$ and i are definitionally equal. Notice that $\mathsf{Inv}_J\, !\, \star$ is just a copy of J, because $!\, j$ is always $\star$ — if there is no structure to respect, then we may choose whatever we like.

Now, let us see if we can give a language for ornamenting a given description. The first three constructors just follow the structure of descriptions making sure that every I-index is assigned a corresponding J-index, but the fourth does something more curious — it permits us to insert new non-recursive fields into the datatype upon which subsequent ornamentation may depend. This will prove important, because we may need more information in order to decide which J-indices to choose than was present in the original I-indexed structure.

$$\begin{array}{l}\mathbf{data}\ \mathsf{Orn}_J\ (e : J \to I) : \mathsf{Desc}\, I \to \mathsf{Set}\ \mathbf{where}\\ \quad \mathsf{arg}_A : ((a : A) \to \mathsf{Orn}_J\, e\, (D\, a)) \to \mathsf{Orn}_J\, e\, (\mathsf{arg}\, A\, D)\\ \quad \mathsf{rec}\ \ : \mathsf{Inv}_J\, e\, h \to \mathsf{Orn}_J\, e\, D \to \mathsf{Orn}_J\, e\, (\mathsf{rec}\, h\, D)\\ \quad \mathsf{ret}\ \ : \mathsf{Inv}_J\, e\, o \to \mathsf{Orn}_J\, e\, (\mathsf{ret}\, o)\\ \quad \mathsf{new} : (A : \mathsf{Set}) \to ((a : A) \to \mathsf{Orn}_J\, e\, D) \to \mathsf{Orn}_J\, e\, D\end{array}$$

Here, I treat arg as an untyped binding operator — the type comes from the original description — and new as a binding operator. I overload constructor names to connect values with the indices to which they correspond.

For a simple but crucial example, let us ornament the natural numbers to get the type of *lists*. This ornament is a simple decoration without refinement: a list is a natural number with decorated successors!

$$\begin{array}{l}\mathsf{ListOrn} : \mathsf{Set} \to \mathsf{Orn}_{\mathsf{1}}\, !\, \mathsf{NatDesc}\\ \mathsf{ListOrn}\, X \mapsto \mathsf{arg}\, a.\, \mathbf{case}\, a\, \mathbf{of}\\ \qquad\qquad\qquad \mathsf{zz} \mapsto \mathsf{ret}\, (\mathsf{inv}\, \star)\\ \qquad\qquad\qquad \mathsf{ss} \mapsto \mathsf{new}\, \_ : X.\, \mathsf{rec}\, (\mathsf{inv}\, \star)\, (\mathsf{ret}\, (\mathsf{inv}\, \star))\end{array}$$

Of course, from every ornament we can extract the new description:

$$\begin{array}{l}
\mathsf{orn} : \mathsf{Orn}_J\, e\, D \to \mathsf{Desc}\, J \\
\mathsf{orn}\,(\mathsf{arg}_A\, O) \mapsto \mathsf{arg}\, a{:}A.\, \mathsf{orn}\,(O\, a) \\
\mathsf{orn}\,(\mathsf{rec}\,(\mathsf{inv}\, j)\, O) \mapsto \mathsf{rec}\, j\,(\mathsf{orn}\, O) \\
\mathsf{orn}\,(\mathsf{ret}\,(\mathsf{inv}\, j)) \mapsto \mathsf{ret}\, j \\
\mathsf{orn}\,(\mathsf{new}\, A\, O) \mapsto \mathsf{arg}\, a{:}A.\, \mathsf{orn}\,(O\, a)
\end{array}$$

Let us just check our example:

$$\begin{array}{l}
\mathsf{orn}\,(\mathsf{ListOrn}\, X) \equiv \mathsf{arg}\, c{:}\{\mathsf{zz},\mathsf{ss}\}.\, \mathbf{case}\, c\, \mathbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{zz} \mapsto \mathsf{ret}\, \star \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{ss} \mapsto \mathsf{arg}\, _{:}X.\, \mathsf{rec}\, \star\, (\mathsf{ret}\, \star)
\end{array}$$

As we might hope, we now have a 'nil' and a 'cons'.

Ornamental Algebras

What use is it to construct lists from numbers in this way? By presenting lists as an ornament on the numbers, we have ensured that lists carry at least as much information as numbers. Correspondingly, there must be an operation which erases this extra information and reveals for each list its inner number. In effect, we have made *length* an intrinsic property of lists.

More generally, for every ornament $O : \mathsf{Orn}_J\, e\, D$, we get a forgetful map

$$|\cdot|_O : (j{:}J) \to \mathsf{Data}\,(\mathsf{orn}\, O)\, j \to \mathsf{Data}\, D\,(e\, j)$$

which rubs out the new information and restores the less informative index. As you might expect, $|\cdot|_{\mathsf{ListOrn}\, X}$ is just the *length* function.

How shall we implement the forgetful map? As the fold of an algebra, of course! Let us take

$$|\cdot|_O \mapsto \mathsf{fold}\,(\mathsf{orn}\, O)\, \chi_O$$

where O's *ornamental algebra* χ_O is defined as follows

$$\begin{array}{l}
\chi_O : \mathsf{Alg}\,(\mathsf{orn}\, \mathsf{O})\,(\mathsf{Data}\, D \circ e) \\
\chi_O\, j\, os \mapsto \langle \mathsf{erase}\, O\, j\, os \rangle\ \mathbf{where} \\
\quad \mathsf{erase} : \mathsf{Orn}_J\, e\, D \to [\![\mathsf{orn}\, O]\!]\,(R \circ e) \subseteq ([\![D]\!]\, R) \circ e \\
\quad \mathsf{erase}\,(\mathsf{arg}\, O) \quad\quad j\,(a, os) \mapsto (a, \mathsf{erase}\,(O\, a)\, j\, os) \\
\quad \mathsf{erase}\,(\mathsf{rec}\,(\mathsf{inv}\, h)\, O)\, j\,(r, os) \mapsto (r, \mathsf{erase}\, O\, j\, os) \\
\quad \mathsf{erase}\,(\mathsf{ret}\,(\mathsf{inv}\, j)) \quad j\, \mathsf{refl} \quad\quad \mapsto \mathsf{refl} \\
\quad \mathsf{erase}\,(\mathsf{new}\, A\, O) \quad\quad j\,(a, os) \mapsto \mathsf{erase}\,(O\, a)\, j\, os
\end{array}$$

The work is actually done by the erase function, which is happy to work with any $R : I \to \mathsf{Set}$ describing the recursive objects: its main job is to remove those elements from the record given by $\mathsf{orn}\, O$ which correspond to the uses of new in O. However, we are also satisfying the index constraints — where we have an $(R \circ e)\, h$ on the left, we deliver an $R\,(e\, h)$ on the right, and in the ret case, unifying the js on the left ensures that the equation we must prove on the right is simply $e\, j = e\, j$.

Somehow, erasure of the new is all we may reasonably expect 'for free' from an arbitrary ornament — it is what we paid for. However, what we now have is a *language* of ornaments, allowing us not only to interpret them, but also to *generate* them in systematic ways. By constructing ornaments to a purpose, we have still more to gain.

5 Algebraic Ornaments

An algebra ϕ describes a structural method to interpret data, giving rise to a $\mathsf{fold}\,\phi$ operation, applying the method recursively. Unsurprisingly, the resulting tree of calls to ϕ has the same structure as the original data — that is the point, after all. But what if that were, *before* all, the point? Suppose we wanted to fix the result of $\mathsf{fold}\,\phi$ in advance, representing only those data which would deliver the answer we wanted. We should need the data to fit with a tree of ϕ calls which delivers that answer. Can we restrict our data to exactly that? Of course we can, if we *index* by the answer.

Starting from a description $D : \mathsf{Desc}\,I$, every $[\![D]\!]$-algebra $\phi : \mathsf{Alg}\,D\,J$ yields an *algebraic ornament*. This is indexed over pairs in $(i : I) \times J\,i$ whose first component must coincide with the original I-index — so the erasure map is just fst — but whose second component is computed by ϕ. We can compute this ornament by inspecting D, requesting a new J-value for each recursive object and steadily building a record of arguments for ϕ so that we can compute and return a J-index for the whole node.

$$
\begin{array}{l}
\mathsf{algo} : (D : \mathsf{Desc}\,I) \to \mathsf{Alg}\,D\,J \to \mathsf{Orn}_{(iI)\times J\,i}\,\mathsf{fst}\,D \\
\mathsf{algo}\,(\mathsf{arg}\,A\,D)\,\phi \mapsto \mathsf{arg}\,a.\,\mathsf{algo}\,(D\,a)\,(\lambda\,i\,ds.\,\phi\,i\,(a,ds)) \\
\mathsf{algo}\,(\mathsf{rec}\,h\,D)\,\phi \mapsto \mathsf{new}\,j : J\,h.\,\mathsf{rec}\,(\mathsf{inv}\,(h,j))\,(\mathsf{algo}\,D\,(\lambda\,i\,ds.\,\phi\,i\,(j,ds))) \\
\mathsf{algo}\,(\mathsf{ret}\,o)\quad\ \phi \mapsto \mathsf{ret}\,(\mathsf{inv}\,(o,\phi\,o\,\mathsf{refl}))
\end{array}
$$

References

Dybjer, P., Setzer, A.: A Finite Axiomatization of Inductive-Recursive Definitions, *TLCA* (J.-Y. Girard, Ed.), 1581, Springer, 1999, ISBN 3-540-65763-0.

McBride, C., McKinna, J.: The View From The Left, *JFP*, 2004.