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 decoration' 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 — encompassing both decoration and, in the sense of Tim Freeman and Frank Pfenning (1991), refinement.

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 spirit of 'the new programming' (McBride & McKinna, 2004), the engineering of coincidence is replaced by the propagation of consequence.

1 Introduction

If it is not a peculiar 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 invented 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 a remarkable coincidence!

In dependently typed programming languages, the possible variations of datatypes are still more dense and subtle, and the coincidences all the more astonishing. For example, working in Agda (Norell, 2008), if I have list types, declared thus,

```
data List (X : Set) : Set where
  []   :                List X
  _::_ : X → List X → List X
```

I might seek to define zip, the function which rearranges a pair of lists into a list of pairs. Inevitably, I will face the question of what to do in the 'off-diagonal' cases, where one list is shorter than the other.

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

One possibility is to return a *dummy* value, perhaps [], as in the Haskell standard Prelude (Peyton Jones, 2003). However, this practice can often mask the presence of a bug: indeed, I found such a bug in an early version of Agda. Such cautionary tales might prompt me to declare *vectors* instead.

```
data Vec (X : Set) : Nat → Set where
  []   :                              Vec X zero
  _::_ : X → ∀ {n} → 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 : ∀ {X Y n} → 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 : ∀ {X n} → Vec X n          → List X          -- forgets index
listVec : ∀ {X n} → (xs : List X) → Vec X (f xs)    -- recomputes index with f
```

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 — a *universe*, in the sense of Per Martin-Löf (1984). This gives us an effective 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.

The work I present here is an initial experiment with ornaments, conducted in Agda, but with the intention of delivering new functionality for Epigram (McBride & McKinna, 2004), where the native notion of datatype is now delivered via a universe (Chapman *et al.*, 2010). It is ironic that an Agda formalisation of ornaments is no use for programming with Agda's native datatypes, good only for those datatypes within the formal universe. It is this distinction between the native data and their formal forgeries which must be abolished if this technology is to achieve its potential, and in the rudimentary Epigram prototype, it has been. Here, however, I am happy to exploit the more polished syntax of Agda to assist in the exposition of the basic idea. The literate Agda source for this paper is available at `http://personal.cis.strath.ac.uk/~conor/pub/OAAO/LitOrn.lagda`.

Related Work

There is a rich literature on *refinement* types, initially conceived by Tim Freeman and Frank Pfenning (1991) as a kind of 'logical superstructure' for ML datatypes, and extensively pursued by Frank Pfenning and his students. Refinement types are at the heart of Hongwei Xi's design for Dependent ML (Xi & Pfenning, 1999): programs are checked at advanced types guaranteeing safety properties, then erased to Standard ML for execution. Rowan Davies (2005) and Joshua Dunfield (2007) further study types and type checking in this setting where the computational substrate is without dependent types.

More recently, William Lovas and Frank Pfenning (2010) have adapted the notion to the Logical Framework, allowing the construction of subset types with proof irrelevance. Matthieu Sozeau (2008) gives a treatment of programming with subset types for a proof-irrelevant incarnation of Coq which has not entirely materialised.

I profit from the insights of the refinement type literature, but in a more general setting — full dependent types with proof-irrelevant propositions (Altenkirch *et al.*, 2007) — and with greater fluidity. Refinements are one kind of ornament, but here we shall see that decoration with additional non-logical data fits in the same scheme. It becomes interesting to condsider the erasure functions which ornaments induce as helpful first-class functions, programmed for free, not just as the erasure preceding execution. We shall surely want to work with both lists and vectors, for example, but without the need to spend effort establishing the connection between them.

Literally and metaphorically closer to home, my Strathclyde colleagues Robert Atkey, Neil Ghani and Patricia Johann (2011) have recently made a study of type refinements induced by algebras over a datatype's structure — the *algebraic* ornaments of this paper's title — citing an early draft of this paper amongst their motivation. Their paper gives a crisp and enlightening categorical account of the broad semantic structure of algebraic ornaments via fibrations, abstracting away from the details of a particular universe encoding. I compliment and complement their work, giving a concrete implementation, and showing how algebras arise *from* ornaments in my more general decoration-and-refinement sense.

Acknowledgements

This work was funded in part by EPSRC grant EP/G034699/1 *Reusability and Dependent Types*, whose prospectus derives in part from my first talk on ornaments, at a Dependently

Typed Programming workshop in February 2008. I am grateful to be a coinvestigator on this project, and I should like to thank the project team — Thorsten Altenkirch, Pierre-Évariste Dagand, Neil Ghani, Jeremy Gibbons, Josh Ko and Peter Morris — for highly productive discussions. I must also thank the members of IFIP Working Group 2.1 for helpful feedback on a presentation of this work. I am delighted also that the Mathematically Structured Programming group at Strathclyde — including Neil and Pierre-Évariste, and also Robert Atkey and Patricia Johann — has provided such a fertile environment in which to develop this work. James McKinna, insightful as ever, provoked the example at the end of the paper: my thanks go also to him.

Technological gratitude is deserved in generous measure by Andres Löh, for his peerless `lhs2TeX` literate programming system (now adapted to Agda), and, of course, to Ulf Norell for the Agda system itself. In a time when research output metrics can so easily lose sight of the very software artefacts which keep us in business, we must stand up and give the credit due.

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 Per Martin-Löf calls a *universe* (Martin-Löf, 1984). The techniques involved here are certainly not new: Marcin Benke, Peter Dybjer and Patrik Jansson (2003) provide a helpful survey. Here, I follow the recipe from Peter Dybjer and Anton Setzer's coding of induction-recursion (Dybjer & Setzer, 1999), suitably adapted to the present purpose.

2.1 Unindexed inductive datatypes

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 a tree-like data structure.

```
data Desc : Set1 where
  end     :                              Desc  -- end of node
  σ       : (S : Set) → (S → Desc) → Desc  -- dependent pair
  node×_  : Desc →                       Desc  -- subnode, then more
```

The description runs left-to-right, with end signalling the end of the node, σ *S D* indicating an element *s* : *S* followed by the rest described by *D s*, and node× *D* marking a place for a subnode followed by more data described by *D*.

An imaginary Robert Harper reminds me to remark that the use of functions to account for type dependency in the σ constructor does not constitute 'higher-order abstract syntax' in the sense of the Logical Framework (Harper *et al.*, 1993). In terms of *polarity* (Licata *et al.*, 2008), the 'positive' LF function space is restricted to correspond exactly to variable binding, but here I use the full 'negative' function space which allows actual computation, admitting the so-called 'exotic terms' which destroy the crucial adequacy property of HOAS representations. This choice is quite deliberate on my part: I make essential use of that extra computational power, as I shall shortly demonstrate.

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. I name the enumeration [ze, su], a suggestive identifier, and I exploit Agda's mixfix notation to give a convenient case analysis operator, especially suited to partial application.

```
data [ze, su] : Set where ze su : [ze, su]
ze↦_su↦_ : ∀ {a} → {P : [ze, su] → Set a} → P ze → P su → (c : [ze, su]) → P c
(ze↦ z  su↦ s) ze = z
(ze↦ z  su↦ s) su = s
```

Ulf Norell's (2007) rationalisation of Randy Pollack's (1992) *argument synthesis* incorporates Dale Miller's (1992) *pattern unification*, which will infer a suitably dependent P just when case analysis is *not* fully applied to a scrutinee.

Now we can use the fully computational dependency built into the σ construct to treat 'constructor choice' as just another component of a variant record.

```
NatP : Desc
NatP = σ [ze, su]   (ze↦ end   su↦ node× end)
```

What does this say? A natural number node begins with a choice of tag; if the tag is ze, we have reached the end; if the tag is su, we require one 'predecessor' subnode, then end.

We shall need to interpret descriptions as actual types in a way which corresponds to the explanation above. To do so, we may follow the standard construction of an inductive datatype as the initial algebra of an endofunctor on Set which drives the *algebra of programming* view of data and functions (Bird & de Moor, 1997). If we have a set X representing subnodes, we can give the set representing whole nodes as follows.

```
⟦_⟧ : Desc → Set → Set
⟦ end       ⟧ X = 1
⟦ σ S D     ⟧ X = Σ S λ s → ⟦ D s ⟧ X
⟦ node× D   ⟧ X = X × ⟦ D ⟧ X
```

Here, 1 is the unit type, with sole inhabitant ⟨⟩ and Σ S T is the type of dependent pairs s, t (unparenthesized) such that s : S and t : T s. Agda extends the scope of a λ rightwards as far as possible, reducing the need for parentheses. Note that $S \times T$ is defined to be the non-dependent special case Σ S λ _ → T.

Correspondingly, if Nat were the type of natural numbers, we should have

```
⟦ NatP ⟧ Nat = Σ [ze, su]   (ze↦ 1   su↦ Nat × 1)
```

But how can we define such a Nat, given NatP? This ⟦_⟧ interprets a description as a strictly positive operator on sets, so we are indeed entitled to an inductive datatype Data D, taking a fixpoint, instantiating X with Data D itself.

```
data Data (D : Desc) : Set where
  ⟨_⟩ : ⟦ D ⟧ (Data D) → Data D
```

We may thus define the set of natural numbers, and its constructors.

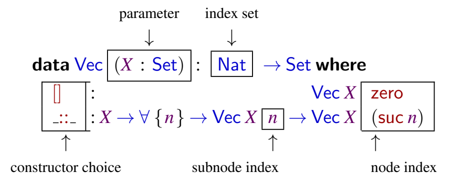
```
Nat : Set             zero : Nat              suc : Nat → Nat
Nat = Data NatP       zero = ⟨ ze, ⟨⟩ ⟩       suc n = ⟨ su, n, ⟨⟩ ⟩
```

Sadly, Agda will not let us use these definitions in pattern matching, so we shall be forced to face artefacts of encoding as we work. Encoded datatypes make William Aitken and John Reppy's (1992) proposed notion of definition fit for left and right still more overdue.

We could go on to develop the initial algebra semantics for this class of functors, but let us first generalise to the indexed case.

2.2 Indexed inductive types

We can give a type which describes inductively defined families of datatypes in I → Set (Dybjer, 1994). Vectors, parametrized by their element type but indexed by their length, are a typical example. Let us revisit them and examine theor salient features.



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

All that changes is that we must ask for a specific index anytime we need a subnode, and we must say which index we deliver when we reach the end of a node. Let us adapt our coding system to account for these extra details.

```
data Desc (I : Set) : Set1 where
  say   : I →                           Desc I
  σ     : (S : Set) (D : S → Desc I) → Desc I
  ask_*_ : I → Desc I →                 Desc I
```

Accordingly, the description of vectors is as follows:

```
VecD : Set → Desc Nat
VecD X = σ [ze, su] (
  ze↦                                  say zero
  su↦ σ X λ x → σ Nat λ n → ask n * say (suc n))
```

We interpret descriptions again as strictly positive endofunctors, but this time on I → Set. We are given the indexed family of recursive subnodes X : I → Set, and we must deliver an indexed family of nodes in I → Set. In effect we are told the index which the node must say. Hence, we should interpret the say construct with an equality constraint: $i' == i$, here, is the usual intensional equality type, allowing constructor refl whenever i' is definitionally equal to i. When the description asks, we know at which index to invoke X. The treatment of σ is just as before.

```
⟦_⟧ : ∀ {I} → Desc I → (I → Set) → (I → Set)
⟦ say i′     ⟧ X i = i′ ≡ i
⟦ σ S D      ⟧ X i = Σ S λ s → ⟦ D s ⟧ X i
⟦ ask i′ * D ⟧ X i = X i′ × ⟦ D ⟧ X i
```

Of course, we acquire inductive datatypes by taking the least fixpoint.

```
data Data {I : Set} (D : Desc I) (i : I) : Set where
  ⟨_⟩ : ⟦ D ⟧ (Data D) i → Data D i
```

Unindexed datatypes like Nat still fit in this picture, just indexing with 1 and inserting trivial indices where required:

```
NatD : Desc 1
NatD = σ [ze, su]   (ze ↦ say ⟨⟩   su ↦ ask ⟨⟩ * say ⟨⟩)
```

The corresponding type and value constructors acquire trivial indices and proofs.

```
Nat : Set              zero : Nat             suc : Nat → Nat
Nat = Data NatD ⟨⟩     zero = ⟨ ze, refl ⟩    suc n = ⟨ su, n, refl ⟩
```

However, we are now in a position to implement the type and value constructors for nontrivially indexed structures, like the vectors:

```
Vec : Set → Nat → Set
Vec X n = Data (VecD X) n

[] : ∀ {X} → Vec X zero
[] = ⟨ ze, refl ⟩

_::_ : ∀ {X} → X → ∀ {n} → Vec X n → Vec X (suc n)
_::_ x {n} xs = ⟨ su, x, n, xs, refl ⟩
```

There are many ways to go about the encoding of inductive families. This choice is rather limited in that it does not allow infinitely branching recursion, and rather rigid in that it prevents us from using the node index *a priori* to compute the node structure. There is no barrier in principle to adapting the techniques of this paper for more sophisticated encodings (Chapman *et al.*, 2010), but there is, I hope, a pedagogical benefit in choosing an encoding which corresponds straightforwardly and faithfully to the more familiar mode of defining inductive families in type theory or the 'Generalized Algebraic Data Types' now popular in Haskell (Cheney & Hinze, 2003; Xi *et al.*, 2003; Sheard, 2004; Peyton Jones *et al.*, 2006).

3 Map and Fold with Indexed Algebras

It is not enough to construct data — we must compute with them. In this section, I describe standard 'algebra of programming' apparatus, but for indexed data structures, and then show how to implement it for the indexed datatypes described by Desc *I*.

Informally, when presenting an inductive datatype as the initial algebra, in : $F\ (\mu\ F) \to \mu\ F$, for a suitable functor F : Set → Set, we provide the action of F on functions, lifting operations on elements uniformly to operations on structures

```
map_F : ∀ {X Y} → (X → Y) → F X → F Y
```

and are rewarded with an *iterator*, everywhere replacing in by ϕ.

```
fold_F : ∀ {X} → (F X → X) → μ F → X
fold_F ϕ (in ds) = ϕ (map_F (fold_F ϕ) ds)
```

We can think of F as a *signature*, describing how to build expressions from subexpressions, and $\mu\ F$ as the syntactic datatype of expressions so generated. An *algebra* $\phi : F\ X \to X$ explains how to implement each of the signature's expression-forms 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.

Let us play the same game with our operators on I → Set. We must first characterise the morphisms or 'arrows' of I → Set — functions which *respect indexing*.

```
_⇉_ : ∀ {I} → (I → Set) → (I → Set) → Set
X ⇉ Y = ∀ {i} → X i → Y i
```

The usual polymorphic identity and composition specialise readily to these ⇉ arrows, verifying the standard categorical laws. We can revert to using unindexed types by instantiating X and Y with constant functions, conveniently given by the usual κ combinator, where κ $i\ S = S$.

We are now in a position to equip our descriptions with their functorial action on arrows.

```
map : ∀ {I X Y} → (D : Desc I) → (X ⇉ Y) → ⟦ D ⟧ X ⇉ ⟦ D ⟧ Y
map (say i)     f q      = q
map (σ S D)     f (s,ds) = s,  map (D s) f ds
map (ask i * D) f (d,ds) = f d, map D f ds
```

We might correspondingly hope to acquire an iterator, taking any index-respecting algebra and performing the index-respecting evaluation of indexed expressions.

```
fold : ∀ {I X} {D : Desc I} → (⟦ D ⟧ X ⇉ X) → Data D ⇉ X
fold {D = D} ϕ ⟨ ds ⟩ = ϕ (map D (fold ϕ) ds)
```

Frustratingly, Agda rejects this definition as less than obviously terminating. Agda's termination oracle cannot see that map will apply fold ϕ recursively only to subterms of *ds*. One might seek to improve the oracle's perspicacity, or better, to express the requirement which map satisfies by type — Abel's *sized type* discipline (Abel, 2006) is certainly adequate to this task. Locally, however, we can expose a satisfactory subterm structure just by long-windedly specialising map to its instance for fold.

```
mutual
  fold : ∀ {I X} {D : Desc I} → (⟦ D ⟧ X ⇉ X) → Data D ⇉ X
  fold {D = D} ϕ ⟨ ds ⟩ = ϕ (mapFold D D ϕ ds)

  mapFold : ∀ {I X} (D E : Desc I) → (⟦ D ⟧ X ⇉ X) →
    ⟦ E ⟧ (Data D) ⇉ ⟦ E ⟧ X
  mapFold D (say i)    ϕ q     = q
mapFold D (σ S E)     φ (s,xs)  =  s,         mapFold D (E s) φ xs
mapFold D (ask i ∗ E) φ (x,xs)  =  fold φ x, mapFold D E φ xs
```

We see here the first instance of a recurring pattern in this paper. We process nodes of recursive structures a little at a time, retaining a fixed description D for the main structure, whilst a helper function walks through E, the description of the current node's tail, invoked with $E = D$. It reminds me of coding division by primitive recursion, when I was a boy.

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

```
addA : Nat → ⟦ NatD ⟧ (κ Nat) ⇸ κ Nat
addA y (ze, _)     = y
addA y (su, z, _)  = suc z

_+_ : Nat → Nat → Nat
x + y  =  fold (addA y) x
```

. . . and vector concatenation — note the careful abstraction of m in the carrier of the algebra:

```
concatA : ∀ {X n} → Vec X n →
   ⟦ VecD X ⟧ (λ m → Vec X (m + n)) ⇸ (λ m → Vec X (m + n))
concatA ys (ze, refl)           = ys
concatA ys (su, x, _, zs, refl)  = x :: zs

_++_ : ∀ {m n X} → Vec X m → Vec X n → Vec X (m + n)
xs ++ ys  =  fold (concatA ys) xs
```

I hesitate to claim that fold delivers the most perspicuous definitions of these operations, especially given the visual overhead of the datatype encoding, but the point is that we can exploit the fact that these operations are folds in reasoning about them, and in performing further constructions. Epigram, of course, was originally conceived as a language combining a readable pattern matching syntax with definition by structural recursion operators (McBride & McKinna, 2004), so we may reasonably hope to have the best of both worlds. Whether or not you view it as desirable, it is not *necessary* to adopt a pointfree style of programming to work with recursion schemes.

4 Ornaments and their Algebras

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 and that of *refining* a datatype with a more subtle index structure. By constructing a fancy datatype in this way, we establish, for free, an algorithmic relationship with its plain counterpart.

Suppose we have some description D : Desc I of an I-indexed family of datatypes (e.g., lists indexed by 1). Now suppose we come up with a more informative index set J (e.g., Nat), together with some function $e : J \to I$ which erases this richer information (e.g., !, the terminal arrow which always returns ⟨⟩). Let us consider how we might develop a description D' : Desc J (e.g., for vectors) with the same recursive structure as D, but richer indexing in an e-respecting way. We should be able to always erase bits of a Data D' j to get an unadorned Data D $(e\, j)$ (e.g., converting vectors to plain old lists).

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:

```
data _⁻¹_ {I J : Set} (e : J → I) : I → Set where
  ok : (j : J) → e ⁻¹ (e j)
```

That is to say, ok $j : e^{-1}\, i$ if and only if $e\, j$ and i are definitionally equal: only the js in i's inverse image are ok. Notice that when we work with $! : J \to 1$, $!^{-1}$ ⟨⟩ is a copy of J, because $!\, j$ is always ⟨⟩ — if there is no structure to respect, it is easily respected.

Now, let us think of a language Orn, for ornamenting a given description. We should be able to mimic the existing structure of descriptions making sure that every I-index is assigned a corresponding J-index. The first three constructors, below, overload the description constructors and deliver just that capability, but the fourth, Δ for 'difference', 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, and we may want more information, anyway.

```
data Orn {I} (J : Set) (e : J → I) : Desc I → Set1 where
  say    : {i : I} → e ⁻¹ i →                                Orn J e (say i)
  σ      : (S : Set) → ∀ {D} → ((s : S) → Orn J e (D s)) →   Orn J e (σ S D)
  ask_∗_ : {i : I} → e ⁻¹ i → ∀ {D} → Orn J e D →            Orn J e (ask i ∗ D)
  Δ      : (S : Set) → ∀ {D} → (S → Orn J e D) →             Orn J e D
```

Before we go the length of vectors, let us have a simple but crucial example, ornamenting 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!

```
ListO : Set → Orn 1 ! NatD
ListO X  =  σ [ze, su] (
  ze ↦                           say (ok ⟨⟩)
  su ↦ Δ X λ _ → ask (ok ⟨⟩) ∗ say (ok ⟨⟩))
```

The difference is given by the Δ, attaching an element of X in the su case.

Now, an ornament is no use unless we can extract the new description to which it leads. To do this, we need only turn Δ to σ and drop the fancy indices into place.

```
⌊_⌋ : ∀ {I J f} {D : Desc I} → Orn J f D → Desc J
⌊ say (ok j) ⌋      = say j
⌊ σ S O ⌋           = σ S λ s → ⌊ O s ⌋
⌊ ask (ok j) ∗ O ⌋  = ask j ∗ ⌊ O ⌋
⌊ Δ S O ⌋           = σ S λ s → ⌊ O s ⌋
```

Checking the example, we define

```
ListD : Set → Desc 1
ListD X  =  ⌊ ListO X ⌋
```

and may then take

```
List : Set → Set
List X = Data (ListD X) ⟨⟩
[] : ∀ {X} → List X
[] = ⟨ ze, refl ⟩
_::_ : ∀ {X} → X → List X → List X
x :: xs = ⟨ su, x, xs, refl ⟩
```

acquiring lists with a ‘nil’ and ‘cons’.

Ornamental Algebras

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

More generally, for every ornament O : Orn J e D, we get a forgetful map

```
forget : ∀ {I J e} {D : Desc I} (O : Orn J e D) →
    {j : J} → Data ⌊ O ⌋ j → Data D (e j)
```

which rubs out the Δ information and restores the less informative index. As you might expect, we shall have

```
length : ∀ {X} → List X → Nat
length {X} = forget (ListO X)
```

With judicious reindexing via composition, we can see forget as an index-respecting function between J-indexed sets, and define it as a fold, given a suitable algebra.

```
forget : ∀ {I J e} {D : Desc I} (O : Orn J e D) →
       (Data ⌊ O ⌋) ∸→ Data D · e
forget O = fold (ornAlg O)
```

where O’s *ornamental algebra* is given as follows

```
ornAlg :  ∀ {I J e} {D : Desc I} (O : Orn J e D) →
          ⟦ ⌊ O ⌋ ⟧ (Data D · e) ∸→ Data D · e
ornAlg O ds = ⟨ erase O ds ⟩
erase : ∀ {I J e} {D : Desc I} {X : I → Set} → (O : Orn J e D) →
          ⟦ ⌊ O ⌋ ⟧ (X · e) ∸→ ⟦ D ⟧ X · e
erase (say (ok j))       refl     = refl
erase (σ S O)            (s, rs)  = s, erase (O s) rs
erase (ask (ok j) ∗ O)   (r, rs)  = r, erase O rs
erase (Δ S O)            (s, rs)  =    erase (O s) rs    -- s is erased
```

The hard work is done by the natural transformation erase, good for plain I-indexed stuff in general (so Data D in particular), erasing, as you can see in the last line, just those components corresponding to a Δ. We are also satisfying the index constraints — where we have an $(X \cdot e)\ j$ on the left, we deliver an $X\ (e\ j)$ on the right, and in the say case, the

refl pattern unfies the js on the left, ensuring that proof obligation on the right is simply $e\ j == e\ j$. By defining ornaments relative to descriptions, we ensure a perfect fit.

5 Algebraic Ornaments

A ⟦ D ⟧-algebra, ϕ, describes a structural method to interpret the data described by D, giving rise to a fold ϕ 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 fold ϕ 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.

For every description D : Desc I, every algebra ϕ : ⟦ D ⟧ J ∸→ J yields an *algebraic ornament*, giving us a type indexed over pairs in Σ I J 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. My colleagues give a semantic account of algebraic ornaments in terms of the *families fibration* (Atkey *et al.*, 2011). In order to give a concrete implementation, I compute the algebraic ornament by recursion on the description D, inserting a J-value to index each recursive object asked for and steadily building a record of arguments for ϕ so that we can compute and report a J-index for the whole node. Here we see another of this paper’s routine techniques, the use of the currying operator, $^\wedge\phi\ s\ t\ =\ \phi\ (s,t)$, to feed an algebra one mouthful at a time.

```
algOrn : ∀ {I J} (D : Desc I) → (⟦ D ⟧ J ∸→ J) → Orn (Σ I J) fst D
algOrn           (say i)     ϕ = say (ok (i, ϕ refl))
algOrn           (σ S D)     ϕ = σ S λ s → algOrn (D s) (^ϕ s)
algOrn {J = J} (ask i ∗ D) ϕ = Δ (J i) λ j → ask (ok (i,j)) ∗ algOrn D (^ϕ j)
```

Working from left to right along the description, we satisfy ϕ’s hunger stepwise, until we are ready to drop the refl in at the right end.

Our first example algebra gives us an example algebraic ornament: we can use the addA m to characterize those numbers of form $m + d$, acquiring a definition of ‘less-or-equal’, which ornaments the type of the difference d, as follows:

```
Le : Nat → Nat → Set
Le m n = Data ⌊ (algOrn NatD (addA m)) ⌋ (⟨⟩, n)
leBase : ∀ {m} →                Le m m
leBase   = ⟨ ze, refl ⟩
leStep : ∀ {m n} → Le m n → Le m (suc n)
leStep x = ⟨ su, _, x, refl ⟩
```

As a consequence of this construction, we acquire the ‘safe subtraction’ function.

```
safeSub : (n m : Nat) → Le m n → Nat
safeSub n m = forget (algOrn NatD (addA m))
```

The safety proof does, in fact, encode the difference, so we need merely decode it.

6 Buy Lists, Get Vectors Free

In section 4, we saw how to make type of lists from the type of natural numbers by an ornament, acquiring an *ornamental algebra* to measure list length in the process. In section 5, we learned to extract an *algebraic ornament* from an algebra, adding an index to a set. Let us put the two together, taking the algebraic ornament of the ornamental algebra to make new ornaments from old.

```
aOoA : ∀ {I J} {e : J → I} {D : Desc I} →
         (O : Orn J e D) → Orn (Σ J (Data D · e)) fst ⌊ O ⌋
aOoA O = algOrn ⌊ O ⌋ (ornAlg O)
```

Applying this recipe to the list ornament, we acquire the ‘lists of a given length’, also known as *vectors*.

```
VecO : (X : Set) → Orn (1 × Nat) fst (ListD X)
VecO X = aOoA (ListO X)

VecD : (X : Set) → Desc (1 × Nat)
VecD X = ⌊ VecO X ⌋

Vec : Set → Nat → Set
Vec X n = Data (VecD X) (⟨⟩, n)
```

We can define the vector constructors in terms of our primitive components.

```
[]   : ∀ {X} →                Vec X zero
[]   = ⟨ ze, refl ⟩

_::_ : ∀ {X n} → X → Vec X n → Vec X (suc n)
x :: xs = ⟨ su, x, _, xs, refl ⟩
```

Like any other, our new ornament has an ornamental algebra inducing a forgetful map:

```
toList : ∀ {X n} → Vec X n → List X
toList {X} = forget (VecO X)
```

We explained how to define vectors when we explained how to see lists as a *decoration* of their lengths! This additional structure becomes manifest once we bring decoration and refinement together under one roof.

The Same Trick Twice

We had an ornament, which give us an algebra, which gave us an ornament, which gave us an algebra. Surely that gives us an ornament which gives us an algebra! The first time, we got lists indexed by length; now we get vectors indexed by their lists, otherwise known as the inductive definition of list length.

```
LengthO : (X : Set) → Orn ((1 × Nat) × List X) fst (VecD X)
LengthO X = aOoA (VecO X)    -- = aOoA (aOoA (ListO X))

Length : {X : Set} → List X → Nat → Set
Length {X} xs n = Data ⌊ LengthO X ⌋ ((⟨⟩, n), xs)
```

```
nilL : ∀ {X} → Length {X} [] zero
nilL = ⟨ ze, refl ⟩

consL : ∀ {X n} {x : X} {xs : List X} → Length xs n → Length (x :: xs) (suc n)
consL l = ⟨ su, _, _, _, l, refl ⟩
```

The forgetful function takes us from the proof of a list’s length to its representation as a vector.

```
lengthVec : ∀ {X n} {xs : List X} → Length xs n → Vec X n
lengthVec {X} = forget (LengthO X)
```

We have seen how to build the ornamental hierarchy, and to descend it with forgetful folds, throwing away index information. To *climb* the hierarchy, we need fold’s dependent cousin, *proof by induction*.

7 Induction

We may prove a generic induction principle for all our datatypes at once. The construction amounts to an effective treatment of the fibrational analysis due to Bart Jacobs and Claudio Hermida (1998), recently generalised by my colleagues, Neil Ghani, Patricia Johann and Clément Fumex (2010). Let me make the statement.

```
induction : {I : Set} (D : Desc I)
            (P : {i : I} → Data D i → Set) →
            ({i : I} (ds : ⟦ D ⟧ (Data D) i) → All D P ds → P ⟨ ds ⟩) →
            {i : I} (x : Data D i) → P x
```

Unpacking this statement, what do we have? For the *I*-indexed datatype with description *D*, let *P* be a ‘predicate’—I have slipped into the language of propositions and proof, but the same construction works for programming, too. To show that *P* holds for all *x*, we must show that *P* holds for each parent ⟨ *d* ⟩, given that it holds for all the *D*-children in *d*. We can compute what it means for *P* to hold for all those children, building a tuple of *P*s.

```
All : {I : Set} (E : Desc I) {D : I → Set}
      (P : {i : I} → D i → Set)
      {i : I} → ⟦ E ⟧ D i → Set
All (say i)     P x      = 1
All (σ S E)     P (s, e) = All (E s) P e
All (ask i ∗ E) P (d, e) = P d × All E P e
```

My colleagues rightly point out that All is, up to bureaucratic isomorphism, the functor on indexed sets given by the *initial* algebraic ornament, describable as ⌊ *algorn D* ⟨_⟩ ⌋. Ornamenting a datatype by its initial algebra expresses exactly the *singleton* property of being *constructed uniquely from constructors*, which holds for all elements, of course. The essence of the fibrational analysis is that induction on a datatype amounts to iteration on its family of singletons. I chose not to use that construction in this paper only because my simple but rigid Desc formulation renders it awkwardly, where the All above neatly computes the tuple structure from the record which indexes it.

The implementation of induction is thus suspiciously like that of fold. As a first attempt, we may try to define induction via a map-like operator.

```
induction D P p ⟨ ds ⟩ = p ds (everywhere D P (induction D P p) ds)
```

```
everywhere : {I : Set} (E : Desc I) {D : I → Set}
             (P : {i : I} → D i → Set) →
             ({i : I} (x : D i) → P x) →
             {i : I} (d : ⟦ E ⟧ D i) → All E P d
everywhere (say i)     P p _     = ⟨⟩
everywhere (σ S E)     P p (s,e) = everywhere (E s) P p e
everywhere (ask i ∗ E) P p (d,e) = p d, everywhere E P p e
```

As with fold, Agda cannot see why the unapplied recursive induction is justified: it does not trust that everywhere will apply the given method only to subobjects of d. Again, we can make the structural recursion clear by specializing everywhere to induction.

```
mutual
  induction : {I : Set} (D : Desc I) (P : {i : I} → Data D i → Set) →
     ({i : I} (ds : ⟦ D ⟧ (Data D) i) → All D P ds → P ⟨ ds ⟩) →
     {i : I} (x : Data D i) → P x
  induction D P p ⟨ ds ⟩ = p ds (everyInd D D P p ds)
  everyInd :
     {I : Set} (E D : Desc I)
     (P : {i : I} → Data D i → Set) →
     ({i : I} (d : ⟦ D ⟧ (Data D) i) → All D P d → P ⟨ d ⟩) →
     {i : I} (d : ⟦ E ⟧ (Data D) i) → All E P d
  everyInd (say i)     D P p _     = ⟨⟩
  everyInd (σ S E)     D P p (s,e) = everyInd (E s) D P p e
  everyInd (ask i ∗ E) D P p (d,e) = induction D P p d, everyInd E D P p e
```

We can use induction to prove properties of specific functions at a specific type. Here, for example, is what Bundy calls ‘the E. Coli of inductive proof’, associativity of addition:

```
assoc : (x y z : Nat) → ((x + y) + z) == (x + (y + z))
assoc x y z = induction NatD (λ x → ((x + y) + z) == (x + (y + z)))
  (ᵛ(ze ↦ (λ _ _ → refl)
     su ↦ ᵛ(λ x _ → ᵛλ H _ → suc ⦀ H)
  )) x
```

Agda does not support a pattern matching presentation of programming or tactical proof with hand-crafted induction schemes, so I am obliged to write an inscrutable proof expression, making use of the uncurrying operator $^{\vee}f\ (s,t) = f\ s\ t$ to split tuples. Moreover, while programmers rejoice when intermediate type information is suppressable, those very types are the intermediate hypotheses and subgoals which show the pattern of reasoning in *proofs*. The Curry-Howard correspondence does not extend to a good discipline of documentation! What you can perhaps make out from my proof is that assoc is not itself

a recursive definition: rather, the inductive step unpacks the inductive hypothesis, H, and delivers suc ⦀ H, the proof that applying suc to both sides preserves the equation.

Remembering, wholesale

We can also use induction in generic programming. With apologies to Philip K. Dick, I shall show how to implement the inverse of forget, for every *algebraic* ornament.

```
remember : ∀ {I J} {D : Desc I} (ϕ : ⟦ D ⟧ J ⇸ J) → let Dᵠ = ⌊ algOrn D ϕ ⌋ in
           {i : I} → (d : Data D i) → Data Dᵠ (i, fold ϕ d)
```

The type of remember says that we can turn a plain old d into its fancier J-indexed counterpart, delivering the very index computed by fold ϕ d. For example, a list becomes a vector of its own length.

The implementation of remember is framed by an induction, but the main work consists of melding the non-recursive data from the plain record with the fancy recursive data from the inductive hypotheses, inserting indices computed by fold ϕ where required.

```
remember {I} {J} {D} ϕ =
  induction D (λ {i} d → Data Dᵠ (i, fold ϕ d)) (λ ds hs → ⟨ meld D ϕ ds hs ⟩) where
   Dᵠ = ⌊ algOrn D ϕ ⌋
   meld : (E : Desc I) (ψ : ⟦ E ⟧ J ⇸ J) {i : I} (e : ⟦ E ⟧ (Data D) i) →
          All E (λ {i} d → Data Dᵠ (i, fold ϕ d)) e →
          ⟦ ⌊ algOrn E ψ ⌋ ⟧ (Data Dᵠ) (i, ψ (mapFold D E ϕ e))
   meld (say i)     ψ refl  hs     = refl
   meld (σ S E)     ψ (s,e) hs     = s, meld (E s) (ᴧψ s) e hs
   meld (ask i ∗ E) ψ (d,e) (h,hs) = j, h, meld E (ᴧψ j) e hs where j = fold ϕ d
```

Again, meld requires a recursion over the datatype structure, with E carrying the remainder of the description (initially D) and ψ its remaining algebra (initially ϕ). As in algOrn itself, ψ is repeatedly curried and fed one piece at a time.

Let us check our example.

```
toVec : {X : Set} (xs : List X) → Vec X (length xs)
toVec {X} = remember (ornAlg (ListO X))
```

To show that remember ϕ and forget (algOrn ϕ) are mutually inverse again requires induction. This is left as an exercise for the reader.

8 The Recomputation Lemma

Suppose we have a vector xs whose length index is n: what is length (toList xs)? We should be astonished if it were anything other than n, but how can we be sure? In this section, I shall prove that an index given by an algebraic ornament can be recomputed by the corresponding fold.

Let us state this property formally. Given a description D and a $[\![D]\!]$-algebra ϕ over some J, we may construct the algebraic ornament O^{ϕ}, and thence the fancy type D^{ϕ}. D^{ϕ} is indexed by pairs, with the second component being a J. We may state that every fancy x's J-index can be recovered from its plain counterpart by fold ϕ.

```
Recomputation : ∀ {I J} (D : Desc I) (φ : ⟦ D ⟧ J ⇉ J) →
                let O^φ = algOrn D φ; D^φ = ⌊ O^φ ⌋ in
                {ij : Σ I J} (x : Data D^φ ij) → fold φ (forget O^φ x) ≡ snd ij
```

The proof goes by induction on the fancy type, with an inner lemma fusing fold ϕ with forget O^ϕ.

```
Recomputation {I} {J} D φ =
  induction D^φ (λ {ij} x → fold φ (forget O^φ x) ≡ snd ij) (fuse D φ) where
    O^φ = algOrn D φ; D^φ = ⌊ O^φ ⌋
```

Of course, the forget is really a fold (ornAlg O^ϕ), so the heart of the proof shows how the two mapFolds combine to feed the algebra what we expect. The type of fuse follows our trusty pattern, abstracting the unprocessed description E and its hungry algebra ψ.

```
fuse : (E : Desc I) (ψ : ⟦ E ⟧ J ⇉ J) → let E^ψ = ⌊ algOrn E ψ ⌋ in
       {ij : Σ I J} (e : ⟦ E^ψ ⟧ (Data D^φ) ij) →
       All E^ψ (λ {ij} x → fold φ (forget O^φ x) ≡ snd ij) e →
       ψ (mapFold D E φ (erase (algOrn E ψ) (mapFold D^φ E^ψ (ornAlg O^φ) e)))
         ≡ snd ij
fuse (say i)     ψ refl      hs                  = refl
fuse (σ S E)     ψ (s, e)    hs                  = fuse (E s) (⌃ψ s) e hs
fuse (ask i ∗ E) ψ (j, x, e) (h, hs) rewrite h   = fuse E (⌃ψ j) e hs
```

In the ask $i * E$ case, j is the index of x, and the goal asks us to show that

$$\psi\ (\mathsf{fold}\ \phi\ (\mathsf{forget}\ O^\phi\ x), \ldots) \equiv \mathsf{snd}\ ij$$

Agda allows us to **rewrite** by inductive hypothesis,

$$h : \mathsf{fold}\ \phi\ (\mathsf{forget}\ O^\phi\ x) \equiv j$$

so that we can feed j to ψ and continue. When we reach the end of the record, we learn the value of its index ij—exactly the value we need.

The recomputation lemma is, in some sense, a statement of the obvious. It tells us that if we can perform a construction at a higher level of precision, we automatically know something about its less precise counterpart.

A Weapon of Math Construction

Standing the recomputation lemma on its head, we acquire a construction method. Suppose we want to write some function

$$\mathsf{f} : A \to \mathsf{Data}\ D\ i$$

whose specification is of form fold $\phi \cdot \mathsf{f} \equiv \mathsf{g}$. That is, the output must have a particular interpretation, according to some algebra ϕ. Algebraically ornamenting D by ϕ allows us to bake the specification into the type of the program. We can try to implement a fancier

function:

$$\mathsf{f}^\phi : (a : A) \to \mathsf{Data}\ \llbracket\ \lfloor \mathsf{algOrn}\ D\ \phi \rfloor\ \rrbracket\ (i, \mathsf{g}\ a)$$

If we succeed, we may define

$$\mathsf{f} = \mathsf{forget}\ (\mathsf{algOrn}\ D\ \phi) \cdot \mathsf{f}^\phi$$

and deduce that the specification is satisfied, gratis. Recomputation gives us that

$$\mathsf{fold}\ \phi\ (\mathsf{f}\ a) \equiv \mathsf{fold}\ \phi\ (\mathsf{forget}\ (\mathsf{algOrn}\ D\ \phi)\ (\mathsf{f}^\phi\ a)) \equiv \mathsf{g}\ a$$

The outside world need not know that we used a rather fancy type to construct f correctly. The ornamentation technology allows us to localize the use of the high precision type. Let us put this method to work.

9 Compiling Hutton's Razor Correctly

James McKinna suggested that I investigate the following example, inspired by Graham Hutton's *modus operandi*. Consider a minimal language of expressions—natural numbers with addition, sometimes referred to as "Hutton's Razor". By way of a reference semantics, we may readily define an interpreter for this language, summing all the numerical leaves in a tree of additions. However, we might prefer to compile such expressions, perhaps to a simple stack machine. A correct compiler will guarantee to produce code which respects the reference semantics. We shall extract this result *for free* from the recomputation lemma.

Our expression language certainly has a description in Desc 1, but let us define it directly: we shall not need to tinker with the structure of expressions, so we may as well see what we are doing.

```
data Exp : Set where
  val  : Nat → Exp
  plus : Exp → Exp → Exp
```

The reference semantics is given by a straightforward recursion, which could certainly be given as a fold.

```
eval : Exp → Nat
eval (val n)      = n
eval (plus a b)   = eval a + eval b
```

Let me direct our primary attention, however, to the stack machine code. For convenience, I define it as a tree structure with sequential composition as a constructor: one may readily flatten the tree at a later stage. For saftey, I index code by the initial and final *stack height* at which it operates. Given directly, we might declare code thus:

```
data Code : (Nat × Nat) → Set where
  PUSH : ∀ {i} →     Nat →                     Code (i, suc i)
  ADD  : ∀ {i} →                               Code (suc (suc i), suc i)
  SEQ  : ∀ {i j k} → Code (i, j) → Code (j, k) → Code (i, k)
```

Note how the indexing ensures that the stack cannot underflow: ADD may only be coded when we are sure that the stack holds at least two values and it decrements the stack height;

PUSH increments stack height; SEQ ensures that code fragments fit together, domino-style. One could, of course, consider Code to be a stack-safe ornament on unsafe code, itself an ornament on plain binary trees, but the point of this example is to push in the other direction, and add yet more indexing. Accordingly, let us translate this declaration systematically to a description. I declare an enumeration for the constructors, then define the description by case analysis.

```
data Op : Set where PUSH ADD SEQ : Op

CodeD : Desc (Nat × Nat)
CodeD  =  σ Op opCodeD where
  opCodeD : Op → Desc (Nat × Nat)
  opCodeD PUSH  =  σ Nat λ i → σ Nat λ _ → say (i, suc i)
  opCodeD ADD   =  σ Nat λ i →             say (suc (suc i), suc i)
  opCodeD SEQ   =  σ Nat λ i → σ Nat λ j → σ Nat λ k →
                    ask (i, j) ∗ ask (j, k) ∗   say (i, k)

Code : (Nat × Nat) → Set
Code  =  Data CodeD
```

Now, the semantic object associated with a given code fragment is the function it induces, mapping an initial stack to a final stack, where stacks are just numeric vectors of the required height.

```
Sem : (Nat × Nat) → Set
Sem (i, j)  =  Vec Nat i → Vec Nat j
```

We can then give the code its semantics by defining the 'execution algebra' for its syntax.

```
ξ : ⟦ CodeD ⟧ Sem ⇸ Sem
```

If we expand the definitions in that type, Agda allows us the following implementation.

```
ξ : ∀ {ij} → ⟦ CodeD ⟧ Sem ij → Vec Nat (fst ij) → Vec Nat (snd ij)
ξ (PUSH, _, n, refl)       ns             =  n :: ns
ξ (ADD, _, refl)           (n :: m :: ns) =  (m + n) :: ns
ξ (SEQ, _, _, _, f, g, refl) ns           =  g (f ns)

exec : Code ⇸ Sem
exec  =  fold ξ
```

We can now state our objective formally, namely to define

```
compile : ∀ {i} → Exp → Code (i, suc i)
```

such that

```
exec (compile e) ns == eval e :: ns
```

but as exec is fold ξ, let us deploy the above method, and try to define a fancier, manifestly correct version:

```
compileξ : ∀ {i} (e : Exp) → Data ⌊ algOrn CodeD ξ ⌋ ((i, suc i), _::_ (eval e))
```

The ornamented code type is indexed by the code's behaviour, as well as by stack height information. We shall only succeed in our efforts if we can deliver code which pushes the given expression's value. Here goes nothing!

```
compileξ (val _)      =  ⟨ PUSH, _, _, refl ⟩
compileξ (plus a b)   =  ⟨ SEQ, _, _, _, _, compileξ a, _, ⟨ SEQ, _, _, _, _, compileξ b, _,
                              ⟨ ADD, _, refl ⟩
                           , refl ⟩, refl ⟩
```

As you can see, Agda has been able to infer all of the indexing details. The basic plan—push values, compile summands then add—was all we had to make explicit. Indeed, we were able to suppress even the number pushed in the val case, as no other value satisfies the specification! We may now define compile and prove it correct at a stroke!

```
compile : ∀ {i} → Exp → Code (i, suc i)
compile  =  forget (algOrn CodeD ξ) · compileξ

compileCorrect : ∀ (e : Exp) {i} → exec (compile {i} e) == _::_ (eval e)
compileCorrect e  =  Recomputation CodeD ξ (compileξ e)
```

To be sure, this construction took some care and benefited from the simplicity of the language involved. For one thing, it was crucial to compile the summands in left-to-right order: had we reversed the order, we should have needed the commutativity of $+$ to persuade the type checker to accept compile$^{\xi}$. More seriously, we escaped some difficulty because our language has only expressions, not control: had we included, say, conditional expressions, we should have been obliged to prove that the actual behaviour, choosing which branch to execute, is equivalent to the reference behaviour, executing both branches and choosing between their values. These proofs are easy, but they are not *nothing*.

What, then, is the essence of this method? Rather than programming by iteration then proving correctness by the corresponding induction, we do both in tandem. Those steps of the proof which amount to rewriting by the inductive hypotheses vanish. For our example, that happens to be the whole of the proof. In general, we should expect some obligations to survive. Effectively, the construction internalizes aspects of the basic proof strategy delivered in Coq by the 'Program' tactics of Catherine Parent and Matthieu Sozeau (Parent-Vigouroux, 1997; Sozeau, 2008): these start from a recursive program and initiate a proof of its correctness by the corresponding induction, leaving the remaining proof obligations for the user.

10 Discussion

In this paper, I have exploited a *universe* construction to give a coding system not only for individual datatypes, but for annotation and refinement relationships *between* datatypes. By making these connections explicit and first class, I was able to give generic implementations of some standard components typically churned out by hand. From the ornament expressing that lists are 'natural numbers with annotations on successor', I acquired the length function as the fold of the ornamental algebra, then the notion of vector as an algebraic ornament, together with operations to shunt data up the ornamental hierarchy. I showed how to acquire cheap proofs for low level programs by using ornaments locally to work at a higher level of precision in the first place. My experiments take the form of Agda programs, typed, total, and available online.

I hasten to add that I do not claim ornaments provide a viable, scalable methodology for Agda programming, or that the rudimentary universe I use in this paper is optimal in practice. Rather, I am grateful that Agda provides a convenient platform for experimenting with the basic idea and delivering proof of concept. Before we can really get to work with this technology, however, we must ensure that our programming languages are geared to support it. Agda delivers a pleasant programming experience for its native **data**-declared types: it is far from pleasant to construct and manipulate their encoded counterparts in our universe, but not for any deep reason. Good language support for encoded data has not, thus far, been a priority, but it is surely possible to repurpose the existing **data** declaration as mere sugar for the *definition* of the corresponding code, and to ensure that those codes carry enough information (e.g., about constructor names, implicit argument conventions, and so forth) to sustain the readability to which we are accustomed.

Libraries should deliver encoded datatypes, to be taken as themes for local variation, ensuring that programmers can work with the precision appropriate to the task at hand without compromising the broader exchange of data. We should not lose the general functionality of a list library just because it is sometimes convenient to work with the precision of vectors, or ordered lists, or ordered vectors, or any of the other multifarious possibilities which we are now free to farm. At the same time, we might develop libraries of ornaments. Nobody should have to think about how to index a particular data structure by its number of nodes, or a bound on its depth, or its flattening to a sequence: such recipes belong in the repertoire. Declaring—rather than defining—datatypes should now be seen as an impediment to the properly compositional data design process, punished by payment for what is rightfully free. As the Epigram team has shown, it is not even necessary to declare the datatype which is used for the descriptions of datatypes: with care, and a handful of predefined components, a sufficiently expressive universe can be persuaded to appear as if it is self-encoded (Chapman *et al.*, 2010).

We are ready to begin an ornamental reimagining of programming. Once aware of the annotation relationship between numbers and lists, for example, we can ask just what extra is needed to develop concatenation from addition. Intuitively, we should just require a function to compute the decoration for the output successor from the decoration on the input successor: if we work polymorphically, parametricity tells us there is *no choice* for the output but to copy the input. By starting not from thin air and inspiration, but rather from a tangible template, we should not only save perspiration, but also achieve greater perspicuity. The proof that the length of a concatenation is the sum of the lengths of its pieces currently requires us to observe a coincidence, when it should rather be the direct consequence of an ornamental construction.

There is much to do, both in theory and in practice. On the theory side, we should seek a more abstract characterization of ornaments, independent of the particular encoding of datatypes at work. The theory of *containers* and *indexed containers* seem apt to the task and an exploration is in progress (Abbott *et al.*, 2005; Altenkirch & Morris, 2009). We should also consider alternative formulations of ornamentation, perhaps as a relation between two descriptions yielding a forgetful map from one to the other. Where in this paper, I consider only *insertion* of new information, we might also want to *delete* old information, provided

we give a way to recover it in the forgetful map. Combining insertion with deletion, we would acquire the general facility to rearrange data structures provided the overall effect is to upgrade with more information. The idea of extending a datatype, stood on its head, is an ornament of this character—restricting constructor choice generates a more specific type which obviously embeds in the original. A formal understanding of the corresponding modifications to programs might have a profound impact on engineering practice, reducing the cost of change.

In practical terms, a key question is just how this technology should be delivered to the programmer. How do I give a datatype, not as a standalone declaration, but as an ornament? How do I collect my free programs and theorems? At the moment, our usual means to refer to particular components of index information tends to be rather *positional* and is thus unlikely to scale: can we adapt *record* types to fit this purpose? Can we minimize the cost of changing representations? It should be possible to store an ornamented data structure as the dependent pair of the plain data with its ornamental extras, manipulating the two in sync, and ensuring that the forgetful function is a constant-time projection in practice. For algebraic ornaments, the additional information is effectively propositional—the evidence for the recomputation lemma—and should thus require no run-time space.

Meanwhile, there are plenty of ornaments which are not algebraic: what other ornament patterns can we identify and systematize? Where algebraic ornaments bubble descriptive indices up from the leaves, there must surely be ornaments which push presciptive indices down from the root or thread before-and-after indices in a static traversal of the data. We can write first-class programs with attribute grammars (Viera *et al.*, 2009) — let us also apply a similarly compositional analysis to first-class *types*.

Dependent types are attractively principled, but they will catch on in practice only when they enable the programmers who use them to outperform the programmers who do not. In certain specialised fields where precision is at a particular premium it can be profitable to bash away with today's technology, armed to the teeth with the latest tactics. But where productivity is the key, it is the programs we need not write ourselves which determine the effectiveness of a technology. The key to constructing programs is the organisation of data, and that is at last in our hands.

References

Abbott, Michael, Altenkirch, Thorsten, & Ghani, Neil. (2005). Containers - constructing strictly positive types. *TCS*.

Abel, Andreas. (2006). *A polymorphic lambda-calculus with sized higher-order types*. Ph.D. thesis, Ludwig-Maximilians-Universität München.

Aitken, William, & Reppy, John. (1992). *Abstract value constructors*. Tech. rept. TR 92-1290. Cornell University.

Altenkirch, Thorsten, & Morris, Peter. (2009). Indexed containers. *Pages 277–285 of: Lics*. IEEE Computer Society.

Altenkirch, Thorsten, McBride, Conor, & Swierstra, Wouter. (2007). Observational equality, now! *Pages 57–68 of:* Stump, Aaron, & Xi, Hongwei (eds), *Plpv*. ACM.

Atkey, Robert, Ghani, Neil, & Johann, Patricia. (2011). When is a Type Refinement an Inductive Type? *FOSSACS*. To appear.

Benke, Marcin, Dybjer, Peter, & Jansson, Patrik. (2003). Universes for generic programs and proofs in dependent type theory. *Nord. j. comput.*, **10**(4), 265–289.

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

Chapman, James, Dagand, Pierre-Évariste, McBride, Conor, & Morris, Peter. (2010). The gentle art of levitation. *Pages 3–14 of:* Hudak, Paul, & Weirich, Stephanie (eds), *Icfp*. ACM.

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

Davies, Rowan. 2005 (May). *Practical refinement-type checking*. Ph.D. thesis, Carnegie Mellon University. CMU-CS-05-110.

Dunfield, Joshua. 2007 (Aug.). *A unified system of type refinements*. Ph.D. thesis, Carnegie Mellon University. CMU-CS-07-129.

Dybjer, Peter. (1994). Inductive families. *Formal asp. comput.*, **6**(4), 440–465.

Dybjer, Peter, & Setzer, Anton. (1999). A finite axiomatization of inductive-recursive definitions. *Pages 129–146 of:* Girard, Jean-Yves (ed), *Tlca*. Lecture Notes in Computer Science, vol. 1581. Springer.

Freeman, Tim, & Pfenning, Frank. (1991). Refinement Types for ML. *Pages 268–277 of: Pldi*.

Ghani, Neil, Johann, Patricia, & Fumex, Clément. (2010). Fibrational induction rules for initial algebras. *Pages 336–350 of:* Dawar, Anuj, & Veith, Helmut (eds), *Csl*. Lecture Notes in Computer Science, vol. 6247. Springer.

Harper, Robert, Honsell, Furio, & Plotkin, Gordon D. (1993). A framework for defining logics. *J. acm*, **40**(1), 143–184.

Hermida, Claudio, & Jacobs, Bart. (1998). Structural induction and coinduction in a fibrational setting. *Inf. comput.*, **145**(2), 107–152.

Huet, Gérard, & Plotkin, Gordon (eds). (1990). *Electronic Proceedings of the First Annual BRA Workshop on Logical Frameworks (Antibes, France)*.

Licata, Daniel R., Zeilberger, Noam, & Harper, Robert. (2008). Focusing on binding and computation. *Pages 241–252 of: Lics*. IEEE Computer Society.

Lovas, William, & Pfenning, Frank. (2010). Refinement types for logical frameworks and their interpretation as proof irrelevance. *Corr*, **abs/1009.1861**.

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

McBride, Conor, & McKinna, James. (2004). The view from the left. *J. funct. program.*, **14**(1), 69–111.

Miller, D. (1992). Unification under a mixed prefix. *J. symbolic computation*, **14**(4), 321–358.

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

Norell, Ulf. (2008). Dependently Typed Programming in Agda. *Pages 230–266 of:* Koopman, Pieter W. M., Plasmeijer, Rinus, & Swierstra, S. Doaitse (eds), *Advanced Functional Programming*. Lecture Notes in Computer Science, vol. 5832. Springer.

Parent-Vigouroux, Catherine. (1997). Verifying programs in the calculus of inductive constructions. *Formal asp. comput.*, **9**(5-6), 484–517.

Peyton Jones, Simon, Vytiniotis, Dimitrios, Weirich, Stephanie, & Washburn, Geoffrey. (2006). Simple unification-based type inference for GADTs. *Pages 50–61 of: Icfp '06*. New York, NY, USA: ACM.

Peyton Jones, Simon L. (2003). Haskell 98: Standard prelude. *J. funct. program.*, **13**(1), 103–124.

Pollack, Robert. (1992). *Implicit syntax*. An earlier version appeared in (Huet & Plotkin, 1990).

Sheard, Tim. (2004). Languages of the future. *Sigplan not.*, **39**(12), 119–132.

Sozeau, Matthieu. 2008 (December). *Un environnement pour la programmation avec types dépendants*. Ph.D. thesis, Université Paris 11, Orsay, France.

Viera, Marcos, Swierstra, S. Doaitse, & Swierstra, Wouter. (2009). Attribute grammars fly first-class: how to do aspect oriented programming in haskell. *Pages 245–256 of:* Hutton, Graham, & Tolmach, Andrew P. (eds), *Icfp*. ACM.

Xi, Hongwei, & Pfenning, Frank. (1999). Dependent types in practical programming. *Pages 214–227 of: Popl*.

Xi, Hongwei, Chen, Chiyan, & Chen, Gang. (2003). Guarded recursive datatype constructors. *Popl*.