Special issue: Advanced Programming Techniques for Construction of Robust, General and Evolutionary Programs

Research Paper

Modularising inductive families

Hsiang-Shang KO[1] and Jeremy GIBBONS[2]

[1,2]Department of Computer Science, University of Oxford

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. For example, sequences might be constrained by length or by an ordering on elements, giving rise to different datatypes "vectors" and "sorted lists" for the same underlying data structure of sequences. Modular implementation of common operations for these structurally similar datatypes has been a longstanding problem. We propose a datatype-generic solution, in which we axiomatise a family of isomorphisms between datatypes and their more refined versions as *datatype refinements*, and show that McBride's *ornaments* can be translated into such refinements. With the ornament-induced refinements, relevant properties of the operations can be separately proven for each constraint, and after the programmer selects several constraints to impose on a basic datatype and synthesises a new datatype incorporating those constraints, the operations can be routinely upgraded to work with the synthesised datatype.

KEYWORDS

Dependently typed programming, inductive families, datatype-generic programming, modularity

1 Introduction

Dependently typed programmers are encouraged to use *inductive families* [7], 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 quickly 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 datatype, with one predicate stating that a list has a certain length and another predicate asserting that a list is sorted. The list datatype is upgraded to the vector datatype, the sorted list datatype, or the sorted vector datatype by simply pairing the list datatype with the sortedness predicate, the length predicate, or the pointwise conjunction of the two predicates, respectively. The common operations are implemented for ordinary lists only, and their properties regarding ordering or length are separately

Received May 10, 2012; Revised October 16, 2012; Accepted December 17, 2012.

[1] *Hsiang-Shang.Ko@cs.ox.ac.uk, http://www.cs.ox.ac.uk/people/hsiang-shang.ko/*,

[2] *Jeremy.Gibbons@cs.ox.ac.uk, http://www.cs.ox.ac.uk/people/jeremy.gibbons/*

DOI: 10.2201/NiiPi.2013.10.5

proven and invoked when needed. Can we somehow introduce this beneficial composability 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 case study on function upgrading. The dependently typed language Agda [12] will be used throughout the paper; its syntax is explained in an appendix. We start with the following function *insert* on lists and try to upgrade it to work on more refined list datatypes:

```
insert : Val → List Val → List Val
insert y [ ] = y :: [ ]
insert y (x :: xs) with y ⩽? x
insert y (x :: xs) | yes _ = y :: x :: xs
insert y (x :: xs) | no  _ = x :: insert y xs
```

where Val is assumed to be a datatype on which there is a decidable total order _⩽_. We might need to be precise about how the length of the list changes. The internalist would use vectors, and reimplement a new version

```
vinsert :
  Val → ∀ {n} → Vec Val n → Vec Val (suc n)
```

whose body is exactly the same as that of *insert*. On the other hand, the externalist might define a relation

```
data Length {A : Set} : Nat → List A → Set
  where
  nil  : Length zero [ ]
  cons : ∀ x {n xs} →
         Length n xs → Length (suc n) (x :: xs)
```

such that a list *xs* has length *n* if and only if there is a proof of type Length *n xs*, and then prove:

```
insert-length : ∀ y {n xs} →
                  Length n xs →
                  Length (suc n) (insert y xs)
insert-length y nil = cons y nil
insert-length y (cons x l)
  with y ⩽? x
insert-length y (cons x l)
  | yes _ = cons y (cons x l)
insert-length y (cons x l)
  | no  _ = cons x (insert-length y l)
```

Afterwards, the externalist can just pair lists with their length proofs and pass the pairs around:

```
insert-l : Val → ∀ {n} →
             Σ (List Val) (Length n) →
             Σ (List Val) (Length (suc n))
insert-l y = insert y × insert-length y
```

where _×_ is defined by $(f \times g)\ (x, y) = (f\ x, g\ y)$ (and is later also overloaded to denote the product type). The two approaches to type refinement are interchangeable, however, since for each *n* there is an isomorphism

```
Vec A n ≅ Σ (List A) (Length n)
```

whose two directions are

```
ℜVec-to :
  ∀ {A n} → Vec A n → Σ (List A) (Length n)
ℜVec-to [ ]       = [ ], nil
ℜVec-to (x :: xs) = (_::_ x × cons x) (ℜVec-to xs)
ℜVec-from :
  ∀ {A n} → Σ (List A) (Length n) → Vec A n
ℜVec-from (_, nil)        = [ ]
ℜVec-from (_, cons x l) = x :: ℜVec-from (_, l)
```

and we can prove that the two directions are indeed inverse to each other:

```
ℜVec-to-from-inverse :
  ∀ {A n} → {s : Σ (List A) (Length n)} →
  ℜVec-to (ℜVec-from s) ≡ s
ℜVec-to-from-inverse {s = (_, nil)}        = refl
ℜVec-to-from-inverse {s = (_, cons x l)} =
  cong (_::_ x × cons x)
       (ℜVec-to-from-inverse {s = _, l})
ℜVec-from-to-inverse :
  ∀ {A n} → {v : Vec A n} →
  ℜVec-from (ℜVec-to v) ≡ v
ℜVec-from-to-inverse {v = [ ]}      = refl
ℜVec-from-to-inverse {v = x :: xs} =
  cong (_::_ x) (ℜVec-from-to-inverse {v = xs})
```

With the help of this family of isomorphisms, *vinsert* and *insert-l* can be defined in terms of each other. For example, the externalist can get *vinsert* by

```
vinsert :
  Val → ∀ {n} → Vec Val n → Vec Val (suc n)
vinsert y xs = ℜVec-from (insert-l y (ℜVec-to xs))
```

which is, in effect, like supplying an additional proof *insert-length* to upgrade *insert* to the more precisely typed *vinsert*.

The same story is repeated when we wish to say that *insert* produces a sorted list if the input list is sorted. The internalist would define another version of lists

```
data SList : Val → Set where
  snil   : ∀ {b} → SList b
```

```
scons : (x : Val) → ∀ {b} → b ⩽ x →
        SList x → SList b
```

which are sorted lists indexed by a lower bound, and reimplement the *insert* function on this datatype.

```
sinsert :
  (y : Val) → ∀ {b} → SList b → SList (b ⊓ y)
```

The relation that the externalist uses this time might be

```
data Sorted : Val → List Val → Set where
  nil   : ∀ {b} → Sorted b [ ]
  cons  :
    ∀ {x b} → b ⩽ x →
    ∀ {xs} → Sorted x xs → Sorted b (x :: xs)
```

They need to prove

```
insert-sorted : ∀ y {b xs} → Sorted b xs →
                Sorted (b ⊓ y) (insert y xs)
```

to get their function

```
insert-s : (y : Val) → ∀ {b} →
           Σ (List Val) (Sorted b) →
           Σ (List Val) (Sorted (b ⊓ y))
insert-s y = insert y × insert-sorted y
```

Again, the internalist and externalist datatypes are intimately related: for each *b* there is an isomorphism

```
SList b ≅ Σ (List Val) (Sorted b)
```

so the externalist can define the internalist version *sinsert* in terms of the externalist version *insert-s*, and vice versa for the internalist.

Things get more interesting when we move on to dealing with ordering and length information simultaneously. The internalist would repeat the story for a third time, defining yet another new version of lists

```
data SVec : Val → Nat → Set where
  svnil   : ∀ {b} → SVec b zero
  svcons  : (x : Val) → ∀ {b} → b ⩽ x →
            ∀ {n} → SVec x n → SVec b (suc n)
```

and reimplement *insert* as

```
svinsert : (y : Val) → ∀ {b n} →
           SVec b n → SVec (b ⊓ y) (suc n)
```

The externalist, however, needs no more new datatypes or proofs this time. To them, a sorted vector is simply a list with proofs that it both has a particular length and is sorted, so they can reuse and assemble the previous proofs to get

```
insert-sv :
  (y : Val) → ∀ {b n} →
  Σ [xs : List Val]
    Sorted b xs × Length n xs →
  Σ [xs : List Val]
    Sorted (b ⊓ y) xs × Length (suc n) xs
insert-sv y =
  insert y × (insert-sorted y × insert-length y)
```

Furthermore, through the family of isomorphisms

```
SVec b n ≅
  Σ [xs : List Val] Sorted b xs × Length n xs
```

they can get the internalist version *svinsert* without additional effort.

This case study suggests that we can switch between internalist and externalist representations to modularly synthesise internalist functions from externalist proofs, making use of the relevant representation-changing isomorphisms. Without the excursion into the externalist world, it would have been less straightforward for the internalist to synthesise *svinsert* from *vinsert* and *sinsert*. The reusability problem is thus reduced to writing the representation-changing isomorphisms. Based on previous work on *ornaments* by McBride and Dagand [6], [10], we propose in this paper a framework in which such isomorphisms can be synthesised *datatype-generically*. We axiomatise the isomorphisms between internalist and externalist datatypes as *refinements*, and show that ornaments[1] translate into a particular class of refinements, so the isomorphisms can be generated by inspecting the ornamental structure of datatypes. Ornaments also help to reveal the same composable structure of internalist datatypes corresponding to that of their externalist brethren — new internalist datatypes can be computed by composing the ornaments about existing internalist datatypes. For example, we would be able to synthesise SVec from the ornaments that describe how Vec and SList differ from List, and obtain all the isomorphisms relating the four datatypes for free, including the one saying that SVec is isomorphic to the externalist representation and allowing us to get *svinsert* from its modularly produced externalist version.

Here is an outline of the paper. Section 2 defines refinements and gives a motivation for a finer analysis of refinements, which is achieved by ornaments. Before ornaments and their (parallel) composition are defined in Section 4, we first introduce *index-first datatypes* [5], [6], which can result in more efficient representations of data, and construct a universe for them

[1] Readers familiar with previous developments on ornaments should note that our terminologies deviate from those in previous works. For a comparison and justification of the deviation, see Section 7.

in Section 3. The main result of this paper is presented in Section 5, where ornaments are translated into refinements and parallel composition of ornaments is shown to give rise to useful composable structure of refinements, enabling modular function upgrading. We give an extended example — *leftist heaps* [13] — in Section 6. Finally, Section 7 discusses related work and some future directions. Our Agda source code is available at http://www.cs.ox.ac.uk/people/hsiang-shang.ko/pcOrn/.

2 Refinements

From the case study in Section 1, we see that isomorphisms such as

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

are the key to moving between internalist and externalist datatypes. In this section we axiomatise these isomorphisms as *refinements*.

2.1 Definition of refinements

We say that a type family $Y : J \to \mathsf{Set}$ refines another type family $X : I \to \mathsf{Set}$ if the members of Y (i.e., the individual types $Y\ j$ where $j : J$) are partitioned such that each partition refines a member of X, say $X\ i$ for some $i : I$, which means that an object of type $X\ i$ can possibly and only be promoted to a type in that partition. The partitioning is specified by a function $e : J \to I$ from finer indices to coarser ones, assigning to (the index of) each member of Y (the index of) a member of X which it refines. We can put this more formally with the help of the inverse image datatype:

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

If X is refined by Y, an object of type $X\ i$ can possibly and only be promoted to $Y\ (und\ j)$ for some $j : e^{-1}\ i$, where the function

```
und : ∀ {J I} {e : J → I} {i} → e ⁻¹ i → J
und (ok j) = j
```

extracts the underlying index that is guaranteed to be mapped to i by e. The possibility of promotion is captured by the *promotion predicate*

```
P : ∀ {i} (j : e ⁻¹ i) → X i → Set
```

which states the condition under which an object x of type $X\ i$ can be converted to one of type $Y\ (und\ j)$ — a “promotion proof” of type $P\ j\ x$ contains necessary information that augments x to an object of type $Y\ (und\ j)$. The conversion, then, is an isomorphism $\mathfrak{R}$ between $Y\ (und\ j)$ and $\Sigma\ (X\ i)\ (P\ j)$, and a refinement consists of the index transformation e, the promotion predicate P, and the refinement isomorphism $\mathfrak{R}$:

```
record Refinement {I J : Set}
  (X : I → Set) (Y : J → Set) : Set₁ where
  field
    e : J → I
    P : ∀ {i} (j : e ⁻¹ i) → X i → Set
    ℜ : ∀ {i} (j : e ⁻¹ i) →
          Iso (Y (und j)) (Σ (X i) (P j))
```

where the type of isomorphisms is defined as an inverse pair of functions, as usual:

```
record Iso (A B : Set) : Set where
  field
    to   : A → B
    from : B → A
    to-from-inverse : ∀ {y} → to (from y) ≡ y
    from-to-inverse : ∀ {x} → from (to x) ≡ x
```

When the more refined type family in a refinement is an inductive family, i.e., an internalist datatype, the refinement then provides a lossless conversion between the internalist datatype and its externalist representation, which is all one needs in order to achieve function upgrading, as illustrated in Section 1. For example, we have all the ingredients for a refinement from $const\ (\mathsf{List}\ A) : \top \to \mathsf{Set}$ (where $const = \lambda\ X\ \_ \to X : \mathsf{Set} \to \top \to \mathsf{Set}$) to $\mathsf{Vec}\ A : \mathsf{Nat} \to \mathsf{Set}$ in Section 1, and we can just put them together:

```
List-Vec : (A : Set) →
             Refinement (const (List A)) (Vec A)
List-Vec A =
  record
    { e  = !
    ; P  = λ {(ok n) → Length n}
    ; ℜ  = λ {(ok n) →
               record
                 { to   = ℜVec-to
                 ; from = ℜVec-from
                 ; to-from-inverse =
                     ℜVec-to-from-inverse
                 ; from-to-inverse =
                     ℜVec-from-to-inverse}}}
```

where the partitioning function is

```
! : {A : Set} → A → ⊤
! _ = tt
```

As the partitioning is trivial, a list $xs : \mathsf{List}\ A$ can be promoted to a vector of type $\mathsf{Vec}\ A\ n$ for “any” n, provided that *P* (ok *n*) *xs*, i.e., Length *n xs*, has a proof. Given this refinement, the *vinsert* function in Section 1 can be reimplemented as

```
vinsert :
  Val → ∀ {n} → Vec Val n → Vec Val (suc n)
vinsert y {n} xs =
  Iso.from (Refinement.ℜ r (ok (suc n)))
    (insert-l y
      (Iso.to (Refinement.ℜ r (ok n)) xs))
  where
    r : Refinement (const (List Val)) (Vec Val)
    r = List-Vec Val
```

where $\mathfrak{R}_{\mathsf{Vec}}$-*to* and $\mathfrak{R}_{\mathsf{Vec}}$-*from* are simply replaced with appropriate fields in *List-Vec* Val.

It is worth noting that the notion of refinements is in general proof-relevant — different promotion proofs can lead to different completed objects. A classic example is the refinement from natural numbers to lists,

```
Nat-List : (A : Set) →
              Refinement (const Nat) (const (List A))
```

in which the promotion predicate is *λ* _ → Vec *A*, meaning that to augment a natural number *n* : Nat to a list of type List *A* we need to supply a vector of type Vec *A n*, i.e., *n* elements of type *A*, and the isomorphism is the usual one between List *A* and Σ Nat (Vec *A*). A natural number *n* can be promoted to different lists of length *n*, which is determined by the choice of promotion proof, i.e., the vector specifying what elements are to be associated with the suc nodes in *n*.

2.2 Predicate swapping

Sometimes we want to swap the promotion predicate *P* in a refinement for an isomorphic one that better suits our needs. For example, instead of the predicate Length, it is more economical to use

```
λ n xs → length xs ≡ n
```

which does not have a recursive structure, and so a proof about the length of a list need not incorpotate proofs about each of its. We hence define a record Swap containing a new predicate *Q* and a proof that it is isomorphic to the old one, i.e., that *P j x* is isomorphic to *Q j x* for all *j* and *x*.

```
record Swap {I J : Set}
  {X : I → Set} {Y : J → Set}
  (r : Refinement X Y) : Set₁ where
  field
    Q : ∀ {i} (j : Refinement.e r ⁻¹ i) →
          (x : X i) → Set
    s : ∀ {i} (j : Refinement.e r ⁻¹ i) →
          (x : X i) →
          Iso (Refinement.P r j x) (Q j x)
```

A new refinement can then be obtained by chaining the isomorphisms together:

$$Y\ (und\ j)\ \cong\ \Sigma\ (X\ i)\ (P\ j)\ \cong\ \Sigma\ (X\ i)\ (Q\ j)$$

This is implemented by

```
toRefinement :
  ∀ {I J} {X : I → Set} {Y : J → Set}
  {r : Refinement X Y} →
  Swap r → Refinement X Y
```

There is an identity swap which simply takes *Q* = *P* and uses the identity isomorphism, whose type is

```
idSwap :
  ∀ {I J} {X : I → Set} {Y : J → Set}
  {r : Refinement X Y} → Swap r
```

For example, we can define a predicate swap for the refinement *List-Vec A* as follows:

```
LengthSwap : (A : Set) → Swap (List-Vec A)
LengthSwap A =
  record
    {Q = λ {(ok n) xs → length xs ≡ n}
    ;s = λ {(ok n) xs →
           record
             {to   = to
             ;from = from
             ;to-from-inverse = UIP
             ;from-to-inverse = ULP}}}
  where
    to : ∀ {n xs} →
          Length n xs → length xs ≡ n
    to nil         = refl
    to (cons x l) = cong suc (to l)
    from : ∀ {xs n} →
             length xs ≡ n → Length n xs
    from {[ ]}      refl = nil
    from {x :: xs} refl = cons x (from refl)
    ULP : ∀ {n} {xs : List A} →
             {l l′ : Length n xs} → l ≡ l′
    ULP {l = nil}         {l′ = nil}          = refl
    ULP {l = cons x l} {l′ = cons .x l′} =
      cong (cons x) ULP
```

where the term

```
UIP : {A : Set} {x y : A}
        {eq eq′ : x ≡ y} → eq ≡ eq′
UIP {eq = refl} {refl} = refl
```

is uniqueness of identity proofs. Then

$$\mathit{toRefinement}\ (\mathit{LengthSwap}\ A)$$

is a refinement that gives us for each n an isomorphism

$$\mathsf{Vec}\ A\ n\ \cong\ \Sigma\,[\mathit{xs} : \mathsf{List}\ A]\ \mathit{length}\ \mathit{xs} \equiv n$$

This predicate swapping mechanism will be used in Section 5.2.

2.3 Problems with refinements

All we have done so far is merely identify the essential ingredients for modular function upgrading and axiomatise them as refinements. Refinements still have to be prepared individually and manually, which requires considerable effort. Moreover, although it is possible to define some sort of refinement composition directly, this approach would not go very far. In Section 1, we get externalist modularity for the internalist datatype SVec because the promotion predicate from lists to sorted vectors is the pointwise conjunction of the promotion predicates from lists to vectors and sorted lists. In general, given two refinements r : Refinement X Y and s : Refinement X Z, we wish to construct a new type family W and a refinement of type Refinement X W whose promotion predicate is the pointwise product of the promotion predicates of r and s. Without knowing the internal structure of Y and Z, all one can do is, roughly speaking, take W to be the pullback of the two maps from Y and Z to X. But this is a very inefficient representation. For example, let X, Y, and Z be *const* (List Val), Vec Val, and SList, respectively. Then an object of type W k for some k would be a pair of a vector and a sorted list with the same elements, meaning that the recursive structure and the elements are duplicated. To avoid such duplication, we need to somehow extract the parts that encode length and ordering information in Vec Val and SList and bake them into a single datatype, but this cannot be done if we work solely with refinements. Hence in the rest of the paper we seek to exploit the structure of datatypes to induce nontrivial refinements systematically — in particular, refinements whose promotion predicate is the pointwise product of the promotion predicates of some other refinements. Such structure can be exposed by *ornaments*, which provide a datatype-generic framework for talking about the relationship between structurally similar datatypes.

3 Index-first 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 [9], 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 done directly in such languages, so datatype-generic programming becomes just ordinary programming in the dependently typed world [1]. In this section we construct a universe of *index-first datatypes* [5], [6], on which a second universe of ornaments, to be constructed in Section 4, will depend.

3.1 An introduction to index-first datatypes

Traditionally, the index in the type of an object is synthesised in a bottom-up fashion following the construction of the object. Consider vectors as an example: the constructor _::_ takes a vector at some index n and constructs a vector at suc n — the final index is computed from the index of the sub-object. This approach, however, can yield redundant representations. For example, the _::_ constructor for vectors has to store the index of the sub-vector, so the representation of a vector would be cluttered with all the intermediate lengths. If we switch to the opposite perspective, determining from the targeted index what data should be supplied, then the representations can usually be significantly cleaned up. For a vector, if the targeted index is given as suc n for some n, then we know that the constructor choice can only be _::_, and that the index of the sub-vector must be n. All we need to supply is the head element and the sub-vector; everything else is determined from the targeted index. This is exactly what Brady's *detagging* optimisation does [4]. With index-first datatypes, however, detagged representations are available directly, rather than arising from a compiler optimisation.

Dagand and McBride [6] designed a new notation for index-first datatypes to reflect this fundamental change to the notion of datatypes. For reasons of presentation, we describe here a slightly more Agda-like variation of their notation. Here is the index-first vector datatype in the new notation:

```
indexfirst data Vec (A : Set) : Nat → Set
  where
  Vec A zero    ∋ []
  Vec A (suc n) ∋ _::_ (x : A) (xs : Vec A n)
```

The header remains the same except for the keyword **indexfirst**. For the constructor part, since constructor choices and what data to supply are now determined by the indices of the requested types, we write the types first. We do pattern matching on the targeted index to determine the constructor choice. If a Vec A zero is requested, the only thing that can be supplied is the nil constructor; if a Vec A (suc n) is requested, it can only be constructed by a cons, which takes a head element x of type A and a vector xs of type Vec A n. Another example is the datatype of sorted lists, which is also more cleanly expressed index-first:

```
indexfirst data SList : Val → Set where
  SList b
    ∋ snil
    | scons (x : Val) (le : b ⩽ x) (xs : SList x)
```

This time the targeted index b is not analysed, and there are always two constructor choices snil and scons. We can also describe the traditional bottom-up vector datatype in this new notation:

```
indexfirst data Vec (A : Set) : Nat → Set
  where
  Vec A n ∋ [ ]   {_ : n ≡ zero}
          | _::_  {m : Nat} {_ : n ≡ suc m}
                  (x : A) (xs : Vec A m)
```

When a vector of type Vec A n is demanded, we are "free" to choose between supplying a nil or a cons regardless of the index n — however, the two constructors now require implicit proofs of equality constraints, indirectly forcing us into a particular choice.

Later on in this paper, the indexfirst data definitions are displayed along with the elements of the universe defined in Section 3.2, i.e., the codes for index-first datatypes, to aid readability. They should not be confused with actual datatype definitions in Agda.

3.2 A universe for index-first datatypes

Now we proceed to construct the universe. An inductive family of type $I \to$ Set is constructed by taking the least fixed point of a base endofunctor on $I \to$ Set. For example, to get index-first vectors, we would define a (parametrised) base functor

```
VecF : Set → (Nat → Set) → (Nat → Set)
VecF A X zero    = ⊤
VecF A X (suc n) = A × X n
```

and take its least fixed point. If we flip the order of arguments of $VecF$ A,

```
VecF : Set → Nat → (Nat → Set) → Set
VecF A zero    = λ X → ⊤
VecF A (suc n) = λ X → A × X n
```

we see that $VecF$ A consists of two different "responses" to the index request, each of type (Nat $\to$ Set) $\to$ Set. It suffices to construct for such responses a universe

```
data RDesc (I : Set) : Set₁
```

with decoding function

```
⟦_⟧ : ∀ {I} → RDesc I → (I → Set) → Set
```

The codes for the responses are called *response descriptions*. A function of type $I \to$ RDesc I, then, can be decoded to an endofunctor on $I \to$ Set, so the type $I \to$ RDesc I acts as a universe for index-first datatypes.

We now define the datatype of response descriptions and its decoding function:

```
data RDesc (I : Set) : Set₁ where
  ▮   : RDesc I
  v   : (i : I) → RDesc I
  σ   : (S : Set) (D : S → RDesc I) → RDesc I
  _∗_ : (D E : RDesc I) → RDesc I

⟦_⟧ : ∀ {I} → RDesc I → (I → Set) → Set
⟦ ▮     ⟧ X = ⊤
⟦ v i   ⟧ X = X i
⟦ σ S D ⟧ X = Σ [s : S] ⟦ D s ⟧ X
⟦ D ∗ E ⟧ X = ⟦ D ⟧ X × ⟦ E ⟧ X
```

Given $X : I \to$ Set, we are allowed to produce the unit type (via the description ▮, suggesting a terminal), fetch a member of X (via v, suggesting a variable position in the base functor), or form a dependent sum (σ) or a binary product (_∗_). As for the actual universe of datatypes $I \to$ RDesc I, to aid type inference in Agda, we wrap the function type in a datatype

```
data Desc (I : Set) : Set₁ where
  wrap : (I → RDesc I) → Desc I
```

and define a deconstructor for it:

```
_at_ : ∀ {I} → Desc I → I → RDesc I
(wrap D) at i = D i
```

Inhabitants of type Desc I will be called *datatype descriptions*, or *descriptions* for short. Least fixed points can then be taken by

```
data μ {I} (D : Desc I) : I → Set where
  con : 𝓕 D (μ D) ⇒ μ D
```

where $\mathscr{F}$ decodes a description of type Desc I to an endofunctor on $I \to$ Set,

```
𝓕 : ∀ {I} → Desc I → (I → Set) → (I → Set)
𝓕 D X i = ⟦ D at i ⟧ X
```

and $X \Rightarrow Y$ is a collection of arrows between corresponding components of X and Y,

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

For example, the code for the base functor of the index-first vector datatype would be

```
VecD : Set → Desc Nat
VecD A = wrap λ { zero    → ■
                ; (suc n) → σ [ _ : A ] v n}
```

and μ (*VecD A*) : Nat → Set gives us the actual datatype to program with.

We can define functions on such vectors by pattern matching. For example,

```
head : ∀ {A n} → μ (VecD A) (suc n) → A
head (con (x, xs)) = x
```

To improve readability, we frequently substitute sugared names of datatypes and constructors for their encodings in function definitions. For example, the above function is sugared into

```
head : ∀ {A n} → Vec A (suc n) → A
head (x :: xs) = x
```

Direct function definitions by pattern matching work fine for individual datatypes, but later when we need to define operations and to state properties for all the datatypes encoded by the universe, it is necessary to have a generic *fold* operator parametrised by the codes. 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 two operators are shown in Figure 1. Their implementations are adapted for the index-first universe from those in McBride's original work [10] but they are essentially the same as those original versions. Note the two-level structure of the definitions of the two operators: the top-level *fold* and *induction* are parametrised by D : Desc I, and the actual analysis of D *at* i : RDesc I happens in a helper function after i is known. This is of course due to the two-level construction of Desc, and this pattern will be followed by all related definitions later.

It is helpful to form a two-dimensional image of our datatype manufacturing scheme: we manufacture a datatype by first defining a base functor, and then recursively duplicating the structure of the functor by taking its least fixed point. The shape of the base functor can be imagined to stretch horizontally, whereas the recursive structure generated by the least fixed point grows vertically. This image works directly when the recursive structure is linear, like lists. (Otherwise one resorts to the abstraction of functor composition.) For example, we can typeset a list two-dimensionally like

```
con (true, a,
con (true, a′,
con (false, tt)))
```

Things following con on each line are shaped by the base functor of lists, whereas the con nodes, aligned vertically, are generated by the least fixed point. This two-dimensional metaphor will be used in later explanations.

4 Ornaments

To establish relationships between datatypes, one idea that comes to mind is to write conversion functions. For some kinds of simple structural conversion like projecting away or assigning default values to fields, however, we may instead state the conversion at the level of datatypes and later translate the statement to the actual conversion function on values that we need. For example, a list is a Peano-style natural number whose successor nodes are decorated with elements, and to convert a list to its length, one simply discards those elements. To be more precise: given the descriptions of the two datatypes,

```
indexfirst data Nat : Set where
  Nat ∋ zero
      | suc (n : Nat)
NatD : Desc ⊤
NatD = wrap λ _ →
           σ Bool λ { false → ■
                    ; true  → v tt}
indexfirst data List (A : Set) : Set where
  List A ∋ []
         | _::_ (x : A) (xs : List A)
ListD : Set → Desc ⊤
ListD A = wrap λ _ →
            σ Bool λ { false → ■
                     ; true  → σ [ _ : A ] v tt}
```

to state the conversion from a list to its length, the essential information is just that the elements associated with cons nodes should be discarded, which is described by the following natural transformation between the two base functors $\mathscr{F}$ (*ListD A*) and $\mathscr{F}$ *NatD*:

```
erase : ∀ {A} → ∀ {X} →
          𝓕 (ListD A) X ⇒ 𝓕 NatD X
erase (false, tt)     = false, tt   -- unchanged
erase (true , (a, x)) = true , x    -- a discarded
```

The transformation can then be lifted to work on the least fixed points.

```
length : ∀ {A} → μ (ListD A) ⇒ μ NatD
length = fold (con ∘ erase {X = μ NatD})
```

```
mutual
  fold : ∀ {I X} {D : Desc I} → 𝓕 D X ⇒ X → μ D ⇒ X
  fold {D = D} φ {i} (con ds) = φ (mapFold D (D at i) φ ds)

  mapFold : ∀ {I} (D : Desc I) (D′ : RDesc I) →
            ∀ {X} → (𝓕 D X ⇒ X) → ⟦ D′ ⟧ (μ D) → ⟦ D′ ⟧ X
  mapFold D ■          φ _          = tt
  mapFold D (v i)      φ d          = fold φ d
  mapFold D (σ S D′)   φ (s, ds)    = s, mapFold D (D′ s) φ ds
  mapFold D (D′ ∗ D″)  φ (ds, ds′)  = mapFold D D′ φ ds, mapFold D D″ φ ds′

All : ∀ {I} (D : RDesc I) {X : I → Set} (P : ∀ {i} → X i → Set) → ⟦ D ⟧ X → Set
All ■          P _          = ⊤
All (v i)      P x          = P x
All (σ S D)    P (s, xs)    = All (D s) P xs
All (D ∗ E)    P (xs, xs′)  = All D P xs × All E P xs′

mutual
  induction :
    ∀ {I} (D : Desc I) (P : ∀ {i} → μ D i → Set) →
    (ih : ∀ {i} (ds : 𝓕 D (μ D) i) → All (D at i) P ds → P (con ds)) →
    ∀ {i} (d : μ D i) → P d
  induction D P ih {i} (con ds) = ih ds (everywhereInduction D (D at i) P ih ds)

  everywhereInduction :
    ∀ {I} (D : Desc I) (D′ : RDesc I) (P : ∀ {i} → μ D i → Set) →
    (ih : ∀ {i} (ds : 𝓕 D (μ D) i) → All (D at i) P ds → P (con ds)) →
    (ds : ⟦ D′ ⟧ (μ D)) → All D′ P ds
  everywhereInduction D ■          P ih _          = tt
  everywhereInduction D (v i)      P ih d          = induction D P ih d
  everywhereInduction D (σ S D′)   P ih (s, ds)    = everywhereInduction D (D′ s) P ih ds
  everywhereInduction D (D′ ∗ D″)  P ih (ds, ds′)  = everywhereInduction D D′ P ih ds,
                                                     everywhereInduction D D″ P ih ds′
```

Fig. 1 The *fold* and *induction* operators.

Our goal in this section is to construct a second universe for such natural transformations between the base functors that arise as decodings of descriptions. The inhabitants of this second universe are called *ornaments*. By encoding the relationship between datatype descriptions as a universe, we will not only be able to derive conversion functions between datatypes, but even compute new datatypes that are related to old ones in prescribed ways, which is something we cannot do if we simply write the conversion functions directly.

4.1 The universe of ornaments

The definition of ornaments, shown in Figure 2, has the same two-level structure as that of datatype descriptions: we have an upper-level datatype Orn of ornaments that refers to a lower-level datatype ROrn of *response ornaments*, which contains the actual encoding details and is decoded by the function *erase*. Parametrised by a partitioning function $e : J \rightarrow I$, the datatype Orn relates two datatype descriptions D : Desc I and E : Desc J such that from an inhabitant O : Orn $e\ D\ E$ we can derive a forgetful map

$$\mathit{forget}\ O : \mu\ E \Rightarrow \mu\ D \circ e$$

By design, this forgetful map necessarily preserves the recursive structure of its input. In terms of the two-dimensional metaphor mentioned at the end of Section 3, an ornament describes only how the horizontal shapes change, and the forgetful map simply applies the changes to each vertical level by a *fold* — it never alters the vertical structure. For example, the *length* function discards elements associated with cons nodes, shrinking the list horizontally to a natural number, but keeps the vertical structure — the con nodes — intact. Look more closely: given $y : \mu\ E\ j$, we should transform it into an object of type $\mu\ D\ (e\ j)$. Deconstructing y into con ys where $ys : [\![\ E\ \mathit{at}\ j\ ]\!]\ (\mu\ E)$ and assuming that the $(\mu\ E)$–objects in ys have been inductively transformed into $(\mu\ D \circ e)$–objects, we horizontally modify the resulting structure of type $[\![\ E\ \mathit{at}\ j\ ]\!]\ (\mu\ D \circ e)$ to one of type $[\![\ D\ \mathit{at}\ (e\ j)\ ]\!]\ (\mu\ D)$, which can then be wrapped by con to an object of type $\mu\ D\ (e\ j)$.

The above steps are performed by the *ornamental algebra* induced by O, whose implementation is shown as *ornAlg* in Figure 2, where the horizontal modification — a transformation from $[\![\ E\ at\ j\ ]\!]\ (X \circ e)$ to $[\![\ D\ at\ (e\ j)\ ]\!]\ X$, natural in X — is decoded by *erase* from a response ornament relating $D\ at\ (e\ j)$ and $E\ at\ j$. Hence an inhabitant of Orn e D E contains for each requested index j a response ornament of type ROrn e $(D\ at\ (e\ j))$ $(E\ at\ j)$ to cope with all possible horizontal structures that can occur in a $(\mu\ E)$–object.

Now we look at each case of the definitions of ROrn and *erase*. The v case says that $[\![\ \mathsf{v}\ j\ ]\!]\ (X \circ e)$ can be transformed into $[\![\ \mathsf{v}\ i\ ]\!]\ X$ if $e\ j\ \equiv\ i$ — since the former type reduces to $X\ (e\ j)$ and the latter to $X\ i$, their indices had better be equal. There are three other cases ■, σ, and _∗_ mirroring the rest of the response description constructors, each of which declares that the same constructor is present in the two related response descriptions, and the structure of the constructor is preserved by *erase*. The remaining two cases deal with addition and deletion of fields inserted by σ and prompt *erase* to perform nontrivial transformations. The Δ case says that the more refined response description, $\sigma\ T\ E$, has an additional field of type T with respect to the response description D being refined. The Δ case of *erase* should transform $[\![\ \sigma\ T\ E\ ]\!]\ (X \circ e)$ — which expands to $\Sigma\ [t : T]\ [\![\ E\ t\ ]\!]\ (X \circ e)$ — into $[\![\ D\ ]\!]\ X$, and it discards the value t of the additional field and continues to transform the remaining structure of type $[\![\ E\ t\ ]\!]\ (X \circ e)$ into $[\![\ D\ ]\!]\ X$, which is guaranteed to succeed since the Δ constructor also demands that D is related to the trailing response description $E\ t$ for every possible value $t : T$ of the additional field. Conversely, the ∇ case says that $\sigma\ S\ D$, having a field of type S, can be refined to E by deleting the field, if E refines $D\ s$ for some $s : S$. This s acts as a default value to be installed into the field when the field is restored by *erase*.

For an example, the ornament from natural numbers to lists is

```
NatD-ListD :
  (A : Set) → Orn ! NatD (ListD A)
NatD-ListD A =
  wrap λ _ → σ Bool λ {false → ■
                       ;true  → Δ [ _ : A ] v refl}
```

The Δ constructor is used to indicate that the field of type A is new in *ListD A*, whereas the other parts are copied from *NatD* as indicated by the mirroring constructors. The forgetful map induced by this ornament discards the field in every cons node of a list, and is exactly *length*. Another example is the ornament from lists to vectors, in which deletion is involved.

```
ListD-VecD :
  (A : Set) → Orn ! (ListD A) (VecD A)
ListD-VecD A =
  wrap λ {zero    → ∇ false ■
         ;(suc n) → ∇ true (σ [ _ : A ] v refl)}
```

We analyse the targeted index: if it is zero, then the constructor choice should be false, so we install that choice with ∇; if it is suc n for some n, then we install the constructor choice true by ∇, copy the element with σ, and finally affirm by v refl that a request of a sub-vector at index n is legitimate with respect to the (trivial) partitioning function ! : Nat → ⊤.

4.2 Ornamental descriptions

The apparent similarity between the description *ListD* and the ornament *NatD-ListD* is typical: frequently we define a new datatype, intending it to be a more refined version of an existing one, and then immediately write an ornament from the latter to the former. The structures of the new datatype and of the ornament are essentially the same, however, so the effort is duplicated. It would be more efficient if we could just write one "relative" description with respect to the existing description, specifying the "patches" that need to be made, and afterwards from this relative description extract a new description and an ornament from the existing description to it. We call such relative descriptions *ornamental descriptions*; their definition is shown in Figure 3 and again has a two-level structure. The lower-level ROrnDesc datatype almost looks like a copy of the ROrn datatype, except that ROrnDesc is indexed by only one response description rather than two — it does not connect two response descriptions like ROrn does, but creates a new response description whose structure is guided by an existing one. From an ornamental description O : OrnDesc J e D, we can extract a new description $\lfloor O \rfloor$: Desc J, which is a more refined version of D, and an ornament $\lceil O \rceil$: Orn e D $\lfloor O \rfloor$ from the reference description D to the new description $\lfloor O \rfloor$. For example, rather than defining *ListD* and then *NatD-ListD*, we can simply write

```
ListO : Set → OrnDesc ⊤ ! NatD
ListO A =
  wrap λ _ →
    σ Bool λ {false → ■
             ;true  → Δ [ _ : A ] v (ok tt)}
```

Then $\lfloor$ *ListO A* $\rfloor$: Desc ⊤ is a description of the list datatype and $\lceil$ *ListO A* $\rceil$: Orn ! *NatD* $\lfloor$ *ListO A* $\rfloor$ is an ornament from natural numbers to lists. By defining the list datatype in a more informative language that allows us to mark the differences between lists and natural numbers, we get the *length* function — the forgetful map induced by the ornament $\lceil$ *ListO A* $\rceil$ — for

```
data ROrn {I J} (e : J → I) : RDesc I → RDesc J → Set₁ where
  ▮    : ROrn e ▮ ▮
  v    : ∀ {j i} (idx : e j ≡ i) → ROrn e (v i) (v j)
  σ    : (S : Set) → ∀ {D E} (O : ∀ s → ROrn e (D s) (E s)) → ROrn e (σ S D) (σ S E)
  Δ    : (T : Set) → ∀ {D E} (O : ∀ t → ROrn e D (E t)) → ROrn e D (σ T E)
  ∇    : {S : Set} (s : S) → ∀ {D E} (O : ROrn e (D s) E) → ROrn e (σ S D) E
  _*_ : ∀ {D E} (O : ROrn e D E) → ∀ {D′ E′} (O′ : ROrn e D′ E′) → ROrn e (D * D′) (E * E′)
erase : ∀ {I J} {e : J → I} {D E} → ROrn e D E → ∀ {X} → ⟦ E ⟧ (X ∘ e) → ⟦ D ⟧ X
erase ▮         _        = tt
erase (v refl)  x        = x
erase (σ S O)   (s, xs)  = s, erase (O s) xs
erase (Δ T O)   (t, xs)  = erase (O t) xs
erase (∇ s O)   xs       = s, erase O xs
erase (O * O′)  (x, x′)  = erase O x, erase O′ x′
data Orn {I J : Set} (e : J → I) (D : Desc I) (E : Desc J) : Set₁ where
  wrap : (∀ j → ROrn e (D at (e j)) (E at j)) → Orn e D E
unwrap : ∀ {I J} {e : J → I} {D E} → Orn e D E → ∀ j → ROrn e (D at (e j)) (E at j)
unwrap (wrap O) = O
ornAlg : ∀ {I J} {e : J → I} {D E} (O : Orn e D E) → 𝓕 E (μ D ∘ e) ⇒ μ D ∘ e
ornAlg {D = D} (wrap O) {j} = con ∘ erase (O j)
forget : ∀ {I J} {e : J → I} {D E} (O : Orn e D E) → μ E ⇒ μ D ∘ e
forget O = fold (ornAlg O)
```

Fig. 2 The universe of ornaments.

free. For another example, we can define sorted lists by making modifications to lists,

```
SListO : OrnDesc Val ! (ListD Val)
SListO =
  wrap λ b →
    σ Bool λ { false → ▮
             ; true  → σ [x : Val]
                       Δ [_ : b ≤ x] v (ok x)}
```

An ornament ⌈ *SListO* ⌉ from *ListD* Nat to ⌊ *SListO* ⌋ can then be decoded from the ornamental description, and subsequently we obtain a forgetful map

```
forget ⌈ SListO ⌉ : ∀ {b} → SList b → List Val
```

that converts a sorted list to a plain list.

4.3 Parallel composition of ornaments

Functions are not the only entities that can be computed from ornaments. Since we have built a universe for datatypes, we can also compute new datatypes from ornaments by computing codes for the new datatypes. A particularly powerful construction is *parallel composition* of ornaments, which plays a central role in this paper. The generic scenario is illustrated in Figure 4: given three descriptions D : Desc I, E : Desc J, and F : Desc K and two ornaments O : Orn e D E and P : Orn e D F independently specifying how D is refined to E and F, we can compute an ornamental description

```
O ⊗ P : OrnDesc (e ⋈ f) pull D
```

incorporating all the modifications to D recorded in O and P. Also we get two *difference ornaments* from E and F to the new description ⌊ $O \otimes P$ ⌋ computed by *diffOrn-l* O P and *diffOrn-r* O P, through which we can partially forget the modifications. For example, the ornament from lists to vectors adds length information, while the ornament from lists to sorted lists enforces ordering; composing the two ornaments in parallel, we get a datatype of lists that keep track of their length and stay ordered at the same time — that is, we get sorted vectors, which can be demoted to vectors or to sorted lists by the forgetful maps induced by the two difference ornaments.

The new index set $e \bowtie f$ is the pullback of e and f. (See the left half of Figure 4 for the commutative diagram.) Set-theoretically, the elements are pairs of the form (j, k) such that $e\ j$ equals $f\ k$, or putting it another way, for which there exists i such that j is in the inverse image of i under e and k is in the inverse image of i under f. Hence we define pullbacks using the inverse image datatype from Section 2:

```
data _⋈_ {I J K : Set}
  (e : J → I) (f : K → I) : Set where
```

```
data ROrnDesc {I : Set} (J : Set) (e : J → I) : RDesc I → Set₁ where
  ∎   : ROrnDesc J e ∎
  v   : ∀ {i} (j : e ⁻¹ i) → ROrnDesc J e (v i)
  σ   : (S : Set) → ∀ {D} (O : ∀ s → ROrnDesc J e (D s)) → ROrnDesc J e (σ S D)
  Δ   : (S : Set) → ∀ {D} (O : S → ROrnDesc J e D) → ROrnDesc J e D
  ∇   : {S : Set} (s : S) → ∀ {D} (O : ROrnDesc J e (D s)) → ROrnDesc J e (σ S D)
  _*_ : ∀ {D} (O : ROrnDesc J e D) → ∀ {D′} (O′ : ROrnDesc J e D′) → ROrnDesc J e (D * D′)
data OrnDesc {I : Set} (J : Set) (e : J → I) (D : Desc I) : Set₁ where
  wrap : (∀ j → ROrnDesc J e (D at (e j))) → OrnDesc J e D
toRDesc : ∀ {I J} {e : J → I} {D} → ROrnDesc J e D → RDesc J
toRDesc ∎          = ∎
toRDesc (v (ok j)) = v j
toRDesc (σ S O)    = σ [s : S] toRDesc (O s)
toRDesc (Δ S O)    = σ [s : S] toRDesc (O s)
toRDesc (∇ s O)    = toRDesc O
toRDesc (O * O′)   = toRDesc O * toRDesc O′
⌊_⌋ : ∀ {I J} {e : J → I} {D} → OrnDesc J e D → Desc J
⌊wrap O⌋ = wrap λ j → toRDesc (O j)
toROrn : ∀ {I J} {e : J → I} {D} → (O : ROrnDesc J e D) → ROrn e D (toRDesc O)
toROrn ∎          = ∎
toROrn (v (ok j)) = v refl
toROrn (σ S O)    = σ [s : S] toROrn (O s)
toROrn (Δ S O)    = Δ [s : S] toROrn (O s)
toROrn (∇ s O)    = ∇ s (toROrn O)
toROrn (O * O′)   = toROrn O * toROrn O′
⌈_⌉ : ∀ {I J} {e : J → I} {D} → (O : OrnDesc J e D) → Orn e D ⌊O⌋
⌈wrap O⌉ = wrap λ i → toROrn (O i)
```

Fig. 3 Ornamental descriptions.

$$\begin{tikzcd}
 & e \bowtie f \arrow[dl, "\pi_1"'] \arrow[dr, "\pi_2"] \arrow[dd, "pull"] & \\
J \arrow[dr, "e"'] & & K \arrow[dl, "f"] \\
 & I &
\end{tikzcd}$$

$$\begin{tikzcd}
 & \lfloor O \otimes P \rfloor : \mathsf{Desc}\ (e \bowtie f) & \\
E : \mathsf{Desc}\ J \arrow[ur, "diffOrn\text{-}l\ O\ P"] & & F : \mathsf{Desc}\ K \arrow[ul, "diffOrn\text{-}r\ O\ P"'] \\
 & D : \mathsf{Desc}\ I \arrow[ul, "O"] \arrow[ur, "P"'] \arrow[uu, "\lceil O \otimes P \rceil"'] &
\end{tikzcd}$$

Fig. 4 Parallel composition of ornaments.

```
_,_ : {i : I} → e ⁻¹ i → f ⁻¹ i → e ⋈ f
```

We have a function *pull* which extracts the common value

```
pull : ∀ {I J K} {e : J → I} {f : K → I} →
         e ⋈ f → I
pull (_,_ {i} _ _) = i
```

and projections

```
π₁ : ∀ {I J K} {e : J → I} {f : K → I} →
       e ⋈ f → J
π₁ (j, _) = und j
π₂ : ∀ {I J K} {e : J → I} {f : K → I} →
       e ⋈ f → K
π₂ (_, k) = und k
```

It is interesting to think about why the new index set is a pullback: the differences recorded in O are only between corresponding responses of D and E as specified by e, and they are indexed by J — for each $j : J$ we get a difference between E *at* j and D *at* $(e\ j)$. The same goes for P. Now, parallel composition computes an ornamental description based on D by mixing O and P. To retrieve the differences recorded in O and P, we need a pair of indices (j, k) to access both ornaments. Not all pairs would do, however, since the two differences retrieved must be based on a common description, otherwise they would have no common structure and could not be mixed. By requiring that $e\ j$ equals $f\ k$, we ensure that the two differences have a common base description. Hence the use of pullbacks.

The full definition of parallel composition is shown in Figure 5, again possessing a two-level structure. The definition of left difference ornaments is shown in Figure 6, which is similar to that of parallel composition but records only modifications from the right-hand side ornament; right difference ornaments have an analogous definition, which is therefore omitted. We look at some representative cases of *pcROrn*. When both ornaments use σ, both of them retain the field in the common base description — no modification is made. Consequently, the field is retained in the resulting ornamental description as well.

```
pcROrn (σ S O) (σ .S P) =
  σ [s : S] pcROrn (O s) (P s)
```

When one of the ornaments uses Δ to mark the addition of a field, that additional field would be inserted into the resulting ornamental description, like in

```
pcROrn (Δ T O) P = Δ [t : T] pcROrn (O t) P
```

If one of the ornaments copies a field by σ and the other deletes it, then the field is deleted in the resulting ornamental description, like in

```
pcROrn (σ S O) (∇ s P) =
  ∇ s (pcROrn (O s) P)
```

The most interesting case is when both ornaments perform deletion: we would put in an equality field demanding that the default values supplied in the two ornaments be equal,

```
pcROrn (∇ s O) (∇ s′ P) =
  Δ (s ≡ s′) (pcROrn-double∇ O P)
pcROrn-double∇ {s = s} O P refl =
  ∇ s (pcROrn O P)
```

and then *pcROrn-double*∇ puts the deletion into the resulting ornamental description after matching the proof of the equality field with refl. It might seem bizarre that two deletions results in an insertion (and a deletion), but consider this informally described scenario: in a base description there is a field $\sigma\ S$, which is refined by two independent ornaments

```
Δ [t : T] ∇ (g t)        and        Δ [u : U] ∇ (h u)
```

That is, instead of S-values, the two ornaments use T- and U-values at this position, which can be erased to an underlying S-value by $g : T \to S$ and $h : U \to S$. Composing these two ornaments in parallel, we get

```
Δ [t : T] Δ [u : U] Δ [_ : g t ≡ h u] ∇ (g t)
```

where the added equality field completes the construction of a pullback of g and h. Here indeed we need a pullback: when we have an actual value for the field $\sigma\ S$, which gets refined to values of types T and U, the easiest way to mix the two refining values is to store them both, as a product. If we wish to retrieve the underlying value of type S, we can either extract the value of type T and apply g to it or extract the value of type U and apply h to it, and through either path we should get the same underlying value. So the product should really be a pullback to ensure this.

For an example, we mentioned that sorted vectors arise out of the parallel composition of the ornaments from lists to vectors and sorted lists. The datatype declaration for index-first sorted vectors is

```
indexfirst data SVec : Val → Nat → Set where
  SVec b zero      ∋  svnil
  SVec b (suc n)   ∋  svcons (x : Val) (le : b ⩽ x)
                             (xs : SVec x n)
```

and the ornamental description from lists to sorted vectors would simply be

```
SVecO : OrnDesc (! ⋈ !) pull (ListD Val)
SVecO = ⌈SListO⌉ ⊗ ListD-VecD Val
```

where the first ! has type Val $\to$ ⊤ and the second Nat $\to$ ⊤ (and hence the index set is essentially just a plain product Val × Nat, justifying the way we index the sugared datatype SVec). Expanding the definition of *SVecO*, we get

```
wrap λ { (ok b, ok zero)      →  ∇ false ▮
       ; (ok b, ok (suc n))  →
           ∇ true (σ [x : Val]
           Δ [_ : b ⩽ x] v ( ok x , ok n ))}
```

where a lighter box indicates modifications recorded in ⌈*SListO*⌉ and a darker box in *ListD-VecD* Val.

5 Refinement semantics of ornaments

In this section we present the main result of this paper: *every ornament* O : Orn $e\ D\ E$ *induces a refinement from* $\mu\ D$ *to* $\mu\ E$. That is, we can construct a function

```
from≡ : ∀ {J I} {e : J → I} {j i} → e j ≡ i → e ⁻¹ i
from≡ {j = j} refl = ok j
to≡ : ∀ {J I} {e : J → I} {i} → (j : e ⁻¹ i) → e (und j) ≡ i
to≡ (ok j) = refl
mutual
  pcROrn : ∀ {I J K} {e : J → I} {f : K → I} {D E F} →
           ROrn e D E → ROrn f D F → ROrnDesc (e ⋈ f) pull D
  pcROrn ▮         ▮         = ▮
  pcROrn ▮         (Δ T P)   = Δ [t : T] pcROrn ▮ (P t)
  pcROrn (v idx)   (v idx′)  = v (ok (from≡ idx, from≡ idx′))
  pcROrn (v idx)   (Δ T P)   = Δ [t : T] pcROrn (v idx) (P t)
  pcROrn (σ S O)   (σ .S P)  = σ [s : S] pcROrn (O s) (P s)
  pcROrn (σ f O)   (Δ T P)   = Δ [t : T] pcROrn (σ f O) (P t)
  pcROrn (σ S O)   (∇ s P)   = ∇ s (pcROrn (O s) P)
  pcROrn (Δ T O) P           = Δ [t : T] pcROrn (O t) P
  pcROrn (∇ s O)   (σ S P)   = ∇ s (pcROrn O (P s))
  pcROrn (∇ s O)   (Δ T P)   = Δ [t : T] pcROrn (∇ s O) (P t)
  pcROrn (∇ s O)   (∇ s′ P)  = Δ (s ≡ s′) (pcROrn-double∇ O P)
  pcROrn (O * O′) (Δ T P)    = Δ [t : T] pcROrn (O * O′) (P t)
  pcROrn (O * O′) (P * P′)   = pcROrn O P * pcROrn O′ P′
  pcROrn-double∇ : ∀ {I J K S} {e : J → I} {f : K → I} {D E F} {s s′ : S} →
    ROrn e (D s) E → ROrn f (D s′) F → s ≡ s′ → ROrnDesc (e ⋈ f) pull (σ S D)
  pcROrn-double∇ {s = s} O P refl = ∇ s (pcROrn O P)
_⊗_ : ∀ {I J K} {e : J → I} {f : K → I} {D E F} →
      Orn e D E → Orn f D F → OrnDesc (e ⋈ f) pull D
_⊗_ {e = e} {f} {D} {E} {F} (wrap O) (wrap P) =
  wrap λ {(j, k) → pcROrn (subst (λ i → ROrn e (D at i) (E at (und j))) (to≡ j) (O (und j)))
                         (subst (λ i → ROrn f (D at i) (F at (und k))) (to≡ k) (P (und k)))}
```

Fig. 5 Parallel composition.

```
RSem : ∀ {I J} {e : J → I} {D E} →
       Orn e D E → Refinement (μ D) (μ E)
```

which is called the *refinement semantics* of ornaments — broadly speaking, we are treating ornaments as a universe for refinements, with *RSem* as the decoding function. We construct in Section 5.1 a *canonical predicate* for every ornament, which is crafted to allow promotion proofs to have efficient representations, and prove that the associated isomorphism holds. When an ornament is a parallel composition, say $O \otimes P$, its canonical predicate can be shown to be isomorphic to the pointwise conjunction of the canonical predicates for O and P — this decomposition of a canonical predicate into existing ones is key to modular function upgrading like the one from *insert* to *svinsert* in Section 1. We express this decomposition as a predicate swap (introduced in Section 2.2) for the refinement *RSem* ($O \otimes P$) in Section 5.2.

5.1 Canonical predicates

We start with constructing a promotion predicate

```
[_]_⊩_ : ∀ {I J} {e : J → I} {D E} →
         ∀ {i} (j : e ⁻¹ i) (x : μ D i) →
         (O : Orn e D E) → Set
```

which is called the *canonical predicate* for the ornament O. Given $x : \mu\ D\ i$, a proof of type $[j]\ x \Vdash O$ would provide the necessary data for complementing x and forming an object y of type $\mu\ E\ (und\ j)$ with the same recursive structure — the proof is the "horizontal" difference between the two objects y and x, speaking in terms of the two-dimensional metaphor sketched in Section 4.1. Such proofs should have the same vertical recursive structure as that of x, and at each recursive node store horizontally only those data marked as modified by the ornament. For example, if we are promoting the natural number

```
two = con (true,
      con (true,
      con (false, tt))) : μ NatD tt
```

to a list, a promotion proof should look like

```
mutual
  diffROrn-l : ∀ {I J K} {e : J → I} {f : K → I} {D E F}
               (O : ROrn e D E) (P : ROrn f D F) → ROrn π₁ E (toRDesc (pcROrn O P))
  diffROrn-l ∎          ∎         = ∎
  diffROrn-l ∎          (Δ T P)   = Δ [t : T] diffROrn-l ∎ (P t)
  diffROrn-l (v refl)   (v idx′)  = v refl
  diffROrn-l (v refl)   (Δ T P)   = Δ [t : T] diffROrn-l (v refl) (P t)
  diffROrn-l (σ S O)    (σ .S P)  = σ [s : S] diffROrn-l (O s) (P s)
  diffROrn-l (σ S O)    (Δ T P)   = Δ [t : T] diffROrn-l (σ S O) (P t)
  diffROrn-l (σ S O)    (∇ s P)   = ∇ s (diffROrn-l (O s) P)
  diffROrn-l (Δ T O) P            = σ [t : T] diffROrn-l (O t) P
  diffROrn-l (∇ s O)    (σ S P)   = diffROrn-l O (P s)
  diffROrn-l (∇ s O)    (Δ T P)   = Δ [t : T] diffROrn-l (∇ s O) (P t)
  diffROrn-l (∇ s O)    (∇ s′ P)  = Δ (s ≡ s′) (diffROrn-l-double∇ O P)
  diffROrn-l (O * O′)   (Δ T P)   = Δ [t : T] diffROrn-l (O * O′) (P t)
  diffROrn-l (O * O′)   (P * P′)  = diffROrn-l O P * diffROrn-l O′ P′

  diffROrn-l-double∇ :
    ∀ {I J K} {e : J → I} {f : K → I} {S} {D E F} {s s′ : S} →
    (O : ROrn e (D s) E) (P : ROrn f (D s′) F) (eq : s ≡ s′) →
    ROrn π₁ E (toRDesc (pcROrn-double∇ {D = D} O P eq))
  diffROrn-l-double∇ O P refl = diffROrn-l O P

diffOrn-l : ∀ {I J K} {e : J → I} {f : K → I} {D E F}
            (O : Orn e D E) (P : Orn f D F) → Orn π₁ E ⌊ O ⊗ P ⌋
diffOrn-l {e = e} {f} {D} {E} {F} (wrap O) (wrap P) =
  wrap λ {(j, k) → diffROrn-l (subst (λ i → ROrn e (D at i) (E at (und j))) (to≡ j) (O (und j)))
                              (subst (λ i → ROrn f (D at i) (F at (und k))) (to≡ k) (P (und k)))}
```

Fig. 6 Left difference ornaments.

```
r = con (a,
      con (a′,
      con tt)) : [ok tt] two ⊩ ⌈ListO A⌉
```

where a and a' are some elements of type A, so we get a list by zipping together *two* and r node by node:

```
con (true, a,
con (true, a′,
con (false, tt))) : μ ⌊ListO A⌋ tt
```

Note that r contains only values of the field marked as additional by Δ in the ornament ⌈ *ListO A* ⌉. The boolean constructor choices are essential for determining the recursive structure of r, but instead of being stored in r, they are derived from *two*, which is part of the index of the type of r. So, in general, here is how we compute an ornamental description of the base functor for such proofs relative to D: we incorporate the modifications made by O, and delete the fields that already exist in D, whose default values are derived in the index-first fashion from the object that we are promoting, which appears in the index of the type of a proof. The deletion is independent of O and can be performed by the *singleton ornament* for D, whose definition *singOrn D* is shown below, so the desired ornamental description is the parallel composition of O and *singOrn D*:

```
cpD : ∀ {I J} {e : J → I} {D E} →
      (O : Orn e D E) → Desc (e ⋈ proj₁)
cpD {D = D} O = ⌊O ⊗ ⌈singOrn D⌉⌋
```

where $proj_1$ here has type $\Sigma\ I\ (\mu\ D) \rightarrow I$. The canonical predicate, then, is the least fixed point of the described base functor.

```
[_]_⊩_ : ∀ {I J} {e : J → I} {D E} →
         ∀ {i} (j : e ⁻¹ i) (x : μ D i) →
         (O : Orn e D E) → Set
[j] x ⊩ O = μ (cpD O) (j, (ok (_, x)))
```

Now we define the singleton ornament *singOrn D* for a description D, which describes a datatype additionally indexed by $\mu\ D$.

```
singOrn : ∀ {I} (D : Desc I) →
          OrnDesc (Σ I (μ D)) proj₁ D
singOrn D =
  wrap λ {(i, con xs) → erode (D at i) xs}
erode :
  ∀ {I} (D : RDesc I) →
```

```
    ∀ {J} → ⟦ D ⟧ J → ROrnDesc (Σ I J) proj₁ D
erode ∎         _          = ∎
erode (v i)      j          = v (ok (i, j))
erode (σ S D) (s, js)     = ∇ s (erode (D s) js)
erode (D ∗ E) (js, js′)  = erode D js ∗ erode E js′
```

An inhabitant of the new datatype is devoid of any horizontal contents, which are deleted by *erode* — only the vertical structure remains. For any type μ ⌊*singOrn D*⌋ (i, x), there is only one single inhabitant (which has the same recursive structure as x), hence the name of the ornament [11].

For an example, the promotion predicate for the ornament *NatD-ListD A* from μ *NatD* to μ (*ListD A*) would be the datatype of index-first vectors. Expanding the definition of the ornamental description *NatD-ListD A* ⊗ ⌈*singOrn NatD*⌉,

```
wrap λ { (ok tt, ok (tt, zero))  →
             ∇ false ∎
         ; (ok tt, ok (tt, suc n)) →
             ∇ true
               ( Δ [ _ : A ] v ( ok tt , ok (tt, n) ))}
```

where lighter box indicates modifications from the ornament *NatD-ListD A* and darker box from the singleton ornament ⌈ *singOrn NatD* ⌉, we see that it indeed yields the datatype of index-first vectors (indexed by a more heavy-weight datatype of natural numbers).

We have just determined the promotion predicate for the refinement semantics of ornaments.

```
RSem : ∀ {I J} {e : J → I} {D E} →
           Orn e D E → Refinement (μ D) (μ E)
RSem {e = e} O =
  record
    { e  = e
    ; P  = λ j x → [j] x ⊩ O
    ; ℜ  = ?}
```

The next step is to prove that μ E (*und* j) and Σ $[x : \mu$ D $i]$ $[j]$ $x \Vdash O$ are isomorphic for any $j : e^{-1}$ i. The backward direction is easy: the canonical predicate datatype $[j]$ $x \Vdash O$ is defined as a parallel composition with O as a component, so there is a difference ornament from the description E, which is the more refined end of O, to the canonical predicate datatype. Hence we define

```
cpOrn :
  ∀ {I J} {e : J → I} {D E} →
  (O : Orn e D E) → Orn π₁ E (cpD O)
cpOrn {D = D} O = diffOrn-l O ⌈singOrn D⌉
```

and the map *forget* (*cpOrn O*) ∘ $proj_2$ does the job. For the forward direction, from an object y : μ E j we need to compute an object x : μ D i and a proof of [ok j] $x \Vdash O$. We take x to be *forget O y*, and the proof is computed by a separate function

```
remember :
  ∀ {I J} {e : J → I} {D E} →
  (O : Orn e D E) →
  ∀ {j} (y : μ E j) → [ok j] forget O y ⊩ O
```

whose implementation is by *induction*. The translation can be completed after proving that the two directions are indeed inverse to each other, again by *induction*. The proofs are tedious but standard, and hence are omitted from the paper.

```
RSem : ∀ {I J} {e : J → I} {D E} →
           Orn e D E → Refinement (μ D) (μ E)
RSem {e = e} O =
  record
    { e  = e
    ; P  = λ j x → [j] x ⊩ O
    ; ℜ  = λ {{._} (ok j) →
              record
                {to        =
                   ⟨forget O, remember O⟩
                ;from =
                   forget (cpOrn O) ∘ proj₂
                ; to-from-inverse =
                   remember-forget-inverse O
                ;from-to-inverse =
                   forget-remember-inverse O}}}
```

5.2 Predicate swap for parallel composition

An ornament describes differences between two datatypes, and the canonical predicate for the ornament is the datatype of differences between objects of the two datatypes. To promote an object from the coarser end to the more refined end of the ornament using its refinement semantics, we give a promotion proof that the object satisfies the canonical predicate for the ornament. If, however, the ornament is a parallel composition, say ⌈ O ⊗ P ⌉, then the differences recorded in the ornament are simply collected from the component ornaments O and P. Consequently, it should suffice to give proofs that the object satisfies the canonical predicates for O and P, instead of the canonical predicate directly induced by ⌈O ⊗ P⌉. We are thus led to prove that the canonical predicate for ⌈ O ⊗ P ⌉ amounts to the pointwise conjunction of the canonical predicates for O and P. In the language of refinements, we provide a predicate swap (introduced in Section 2.2) that allows us to use the pointwise conjunction of the canonical predicates for O and P as the promotion predicate in *RSem* $\lceil O \otimes P \rceil$, instead of the canonical predicate for $\lceil O \otimes P \rceil$. We should allow predicate swapping to propagate, though: the canonical predicate for $\lceil O \otimes P \rceil$ can be swapped for the pointwise conjunction of any predicates that are isomorphic to the canonical predicates for O and P, so, for example, the canonical predicate for $\lceil O \otimes \lceil P \otimes Q \rceil \rceil$ can be swapped for the pointwise conjunction of the canonical predicates for O, P, and Q. Hence the predicate swap we provide is:

```
Swap-⊗ :
  ∀ {I J K} {e : J → I} {f : K → I} {D E F}
  (O : Orn e D E) (P : Orn f D F) →
  Swap (RSem O) → Swap (RSem P) →
  Swap (RSem ⌈O ⊗ P⌉)
Swap-⊗ O P s t =
  record
    { Q = λ {{._} (ok (j, k)) x →
                  Swap.Q s j x × Swap.Q t k x}
    ; s  = ?}
```

For the field s, we need only prove that the canonical predicate for $\lceil O \otimes P \rceil$ is isomorphic to the pointwise conjunction of the canonical predicates for O and P, whose forward direction is

```
project :
  ∀ {I J K}
  {e : J → I} {f : K → I} {D E F} →
  (O : Orn e D E) (P : Orn f D F) →
  ∀ {i} (x : μ D i) {j : e ⁻¹ i} {k : f ⁻¹ i} →
  [ok (j, k)] x ⊩ ⌈O ⊗ P⌉ →
  [j] x ⊩ O × [k] x ⊩ P
```

The implementation proceeds by *induction* on x and distributes the data in the composite proof to the two component proofs that we are constructing. The function *project* can be shown to be injective and surjective, so we get an isomorphism which we can then chain with the product of the two given isomorphisms Swap.$s\ s\ j\ x$ and Swap.$s\ t\ k\ x$ by *transIso*. That is, we can indeed form an isomorphism

```
[ok (j, k)] x ⊩ ⌈O ⊗ P⌉
  ≅ [j] x ⊩ O × [k] x ⊩ P
  ≅ Swap.Q s j x × Swap.Q t k x
```

which is what we use for the field s of *Swap*-⊗.

For an example, the key isomorphisms used to modularly upgrade *insert* to *svinsert* in Section 1

```
SVec b n ≅
  Σ [xs : List Val] Sorted b xs × Length n xs
```

can be provided by the refinement

```
toRefinement
  (Swap-⊗ ⌈SListO⌉ (ListD-VecD Val)
          idSwap idSwap)
```

If, instead of the inductive predicate Length $n\ xs$, we wish to program with the equality $length\ xs \equiv n$, then we use the refinement

```
toRefinement
  (Swap-⊗ ⌈SListO⌉ (ListD-VecD Val)
          idSwap (LengthSwap Val))
```

which gives us the family of isomorphisms

```
SVec b n ≅
  Σ [xs : List Val] Sorted b xs × length xs ≡ n
```

6 Example: leftist heaps

In this section we give an extended example: *leftist heaps*. In Okasaki's words [13], "[l]eftist heaps [...] are heap-ordered binary trees that satisfy the *leftist property*: the rank of any left child is at least as large as the rank of its right sibling. The rank of a node is defined to be the length of its *right spine* (i.e., the rightmost path from the node in question to an empty node)." From this description we can immediately decompose the concept of leftist heaps into three: leftist heaps (i) are binary trees that (ii) are heap-ordered and (iii) satisfy the leftist property. This suggests that there is a basic datatype of binary trees together with two ornamentations. The datatype of binary trees is

```
indexfirst data Tree : Set where
  Tree ∋ tip
       | fork (t : Tree) (u : Tree)

TreeD : Desc ⊤
TreeD = wrap λ _ →
          σ Bool λ { false → ■
                   ; true  → v tt ∗ v tt}
```

Leftist trees — binary trees satisfying the leftist property — are then an ornamented version of Tree.

```
indexfirst data LTree : Nat → Set where
  Tree zero    ∋ tip
  Tree (suc r) ∋ fork (l : Nat) (r⩽l : r ⩽ l)
                      (t : Tree l) (u : Tree r)

LTreeO : OrnDesc Nat ! TreeD
LTreeO =
  wrap λ { zero    → ∇ false ■
         ; (suc r) →
             ∇ true ( Δ [l : Nat] Δ [_ : r ⩽ l]
                      v (ok l) ∗ v (ok r))}
```

Independently, heap-ordered trees are also an ornamented version of Tree.

```
indexfirst data Heap : Val → Set where
  Heap b ∋ tip
         | fork (x : Val) (b≤x : b ≤ x)
                (t : Heap x) (u : Heap x)
HeapO : OrnDesc Val ! TreeD
HeapO =
  wrap λ b →
    σ Bool λ { false → ■
             ; true  → Δ [x : Val] Δ [_ : b ≤ x]
                       v (ok x) ∗ v (ok x)}
```

(One can see from the indexing pattern that heap-ordered trees can be regarded as a generalisation of sorted lists: in a heap-ordered tree, every path from the root to a tip is a sorted list.) Composing the two ornaments in parallel gives us exactly leftist heaps.

```
indexfirst data LHeap : Val → Nat → Set
  where
  LHeap b zero ∋ tip
  LHeap b (suc r)
    ∋ fork (x : Val) (b≤x : b ≤ x)
           (l : Nat) (r≤l : r ≤ l)
           (t : Heap x l) (u : Heap x r)
LHeapD : Desc (! ⋈ !)
LHeapD = ⌊⌈HeapO⌉ ⊗ ⌈LTreeO⌉⌋
```

The decomposition gives us the ability to talk about heap-ordering and the leftist property of leftist heaps independently. For example, a useful operation on heap-ordered trees is to relax the lower bound. If we implement it in predicate form, stating explicitly in the type that the underlying binary tree structure is unchanged,

```
relax : ∀ {b b′} → b′ ≤ b →
        ∀ {t} → [ok b] t ⊩ ⌈HeapO⌉ →
        [ok b′] t ⊩ ⌈HeapO⌉
relax b′≤b {tip}       p = con tt
relax b′≤b {fork _ _} (con (x, b≤x, t, u)) =
  con (x, ≤-trans b′≤b b≤x, t, u)
```

where ≤-*trans* is transitivity of _≤_, then we can lift it so as to modify only the heap-ordering portion of a leftist heap:

```
lhrelax : ∀ {b b′} → b′ ≤ b →
          ∀ {r} → LHeap b r → LHeap b′ r
lhrelax {b} {b′} b′≤b {r} =
  Iso.from (Refinement.ℜ re (ok (ok b′, ok r))) ∘
    (id × (relax b′≤b × id)) ∘
      Iso.to (Refinement.ℜ re (ok (ok b, ok r)))
  where
    re : Refinement (μ TreeD) (μ LHeapD)
    re = toRefinement
           (Swap-⊗ ⌈HeapO⌉ ⌈LTreeO⌉
                   idSwap idSwap)
```

In general, non-modifying heap operations do not depend on the leftist property and can be implemented for heap-ordered trees and later lifted to work with leftist heaps, relieving us of the unnecessary work of dealing with the leftist property when it is simply to be ignored. For another example, converting a leftist heap to a list of its elements has nothing to do with the leftist property. In fact, it even has nothing to do with heap-ordering, but only with the internal labelling. Hence we define the *internally labelled trees*

```
indexfirst data ITree (A : Set) : Set where
  ITree A ∋ tip
          | fork (x : A)
                 (t : ITree A) (u : ITree A)
ITreeO : Set → OrnDesc ⊤ ! TreeD
ITreeO A =
  wrap λ _ →
    σ Bool λ { false → ■
             ; true  →
                 Δ [_ : A] v (ok tt) ∗ v (ok tt)}
```

on which we can do pre-order traversal:

```
preorder : ∀ {A} → ITree A → List A
preorder tip          = []
preorder (fork x t u) =
  x :: preorder t ++ preorder u
```

We have an ornament from internally labelled trees to heap-ordered trees:

```
ITreeD-HeapD : Orn ! ⌊ITreeO Val⌋ ⌊HeapO⌋
ITreeD-HeapD =
  wrap λ b →
    σ Bool λ { false → ■
             ; true  → σ [x : Val] Δ [_ : b ≤ x]
                       v refl ∗ v refl}
```

So, to get a list of the elements of a leftist heap (with the first element of the list, if any, being the minimum one in the heap), we convert the leftist heap to an internally labelled tree and then invoke *preorder*.

```
toList : ∀ {b r} → LHeap b r → List Val
toList = preorder ∘ forget ITreeD-HeapD ∘
           forget (diffOrn-l ⌈HeapO⌉ ⌈LTreeO⌉)
```

For modifying operations, however, we need to consider both heap-ordering and the leftist property at the same time, so we should program directly with the composite datatype of leftist heaps. For example, the key modifying operation is merging two heaps,

```
merge : ∀ {b r} → LHeap b r →
        ∀ {b′ r′} → LHeap b′ r′ →
        Σ Nat (LHeap (b ⊓ b′))
```

with which we can easily implement insertion of a new element and deletion of the minimum element. The definition of *merge* is shown in Figure 7. It is a more precisely typed version of Okasaki's implementation, split into two mutually recursive functions to make the two-level induction clear to Agda's termination checker, and conversions are added to establish the correct bounds.

Another advantage of separating the leftist property and heap-ordering is that we can swap the leftist property for another balancing property. The non-modifying operations, previously defined for heap-ordered trees, can be upgraded to work with the new balanced heap datatype in the same way, while the modifying operations are reimplemented with respect to the new balancing property. For example, the leftist property requires that the *rank* of the left subtree is at least that of the right one; we can replace "rank" with "size" in its statement and get the *weight-biased leftist property*. This is again codified as an ornamentation of binary trees

```
indexfirst data WLTree : Nat → Set where
  WLTree zero  ∋  tip
  WLTree (suc n)
      ∋  fork (l : Nat) (r : Nat)
              (r⩽l : r ⩽ l) (n≡l+r : n ≡ l + r)
              (t : WLTree l) (u : WLTree r)
WLTreeO : OrnDesc Nat ! TreeD
WLTreeO =
  wrap λ { zero     → ∇ false ∎
         ; (suc n) →
             ∇ true
               (Δ [l : Nat] Δ [r : Nat]
                Δ [_ : r ⩽ l] Δ [_ : n ≡ l + r]
                v (ok l) ∗ v (ok r))}
```

which can be composed in parallel with the heap-ordering ornament and give us weight-biased leftist heaps.

```
indexfirst data WLHeap : Val → Nat → Set
  where
  WLHeap b zero  ∋  tip
  WLHeap b (suc n)
      ∋  fork (x : Val) (b⩽x : b ⩽ x)
              (l : Nat) (r : Nat)
              (r⩽l : r ⩽ l) (n≡l+r : n ≡ l + r)
              (t : WLHeap x l) (u : WLHeap x r)
```

```
WLHeapD : Desc (! ⋈ !)
WLHeapD = ⌊⌈HeapO⌉ ⊗ ⌈WLTreeO⌉⌋
```

Switching to the weight-biased leftist property makes it possible to reimplement *merge* in a single, top-down pass rather than two passes: with the original rank-biased leftist property, recursive calls to *merge* are determined top-down by comparing root elements, and the helper function *makeT* swaps the recursive result with the other subtree if the rank of the former is larger; the rank of the result, however, is not known before the recursive call returns, so during the whole merging process *makeT* does the swapping in a second bottom-up pass. On the other hand, with the weight-biased leftist property, the size of the recursive result is known before the merging is actually performed, so *makeT* can determine whether to do the swapping or not before the recursive call, and the whole merging process requires only one top-down pass. The new implementation is similar to the one for rank-biased leftist heaps and is thus omitted from the paper.

7 Discussion

This paper is a heavily revised version of the one that the authors previously published in the Workshop of Generic Programming (WGP) [8]. The WGP version was the first to use the terms "internalism" and "externalism" for naming different ways of expressing constraints known by the dependently typed programming community, the former using inductive families with fancy indices and the latter using separately defined predicates, and to show that there is a connection between internalism and externalism: whereas externalist constraints are expressed by predicates, internalist constraints can be expressed by ornaments, and we can derive a predicate from every ornament, thereby translating internalist constraints to externalist ones. This connection is axiomatised in this paper in terms of refinements. The axiomatisation greatly streamlines the presentation, as it makes a clear logical separation between how (modular) function upgrading can be achieved by having isomorphisms between internalist and externalist datatypes and how a particular class of such isomorphisms can be induced by capturing structural similarities between datatypes with ornaments.

We might say that ornaments form a universe for refinements (in a broader sense). Even though it is obvious that ornaments encode only a small collection of refinements, what we have achieved is typical of universe constructions: refinements on their own do not have a very useful compositional structure, but we can identify a collection of more composable refinements by reflecting their deeper structure as codes, i.e., ornaments. This collection of ornament-induced refine-

```
    -- We assume the existence of the function ≰-invert : ∀ {x y} → x ≰ y → y ⩽ x
    -- (which makes _⩽_ a total ordering).
    -- Various proof terms about equalities/inequalities are not essential and
    -- thus omitted; instead, the holes {! !} are filled with the expected types only.
makeT : (x : Nat) → ∀ {r} (t : LHeap x r) → ∀ {r′} (t′ : LHeap x r′) → Σ Nat (LHeap x)
makeT x {r} t {r′} t′ with r ⩽? r′
makeT x {r} t {r′} t′ | yes r⩽r′ = suc r , fork x ⩽-refl r′ r⩽r′ t′ t
makeT x {r} t {r′} t′ | no  r≰r′ = suc r′, fork x ⩽-refl r (≰-invert r≰r′) t t′
mutual
  merge : ∀ {b r} → LHeap b r → ∀ {b′ r′} → LHeap b′ r′ → Σ Nat (LHeap (b ⊓ b′))
  merge {b} {zero } h {b′} h′ = _, lhrelax {! b ⊓ b′ ⩽ b′ !} h′
  merge {b} {suc r} h {b′} h′ = merge′ h h′
  merge′ : ∀ {b r} → LHeap b (suc r) → ∀ {b′ r′} → LHeap b′ r′ → Σ Nat (LHeap (b ⊓ b′))
  merge′ {b} {r} h {b′} {zero} h′ =
    _, lhrelax {! b ⊓ b′ ⩽ b !} (subst (LHeap b) {! suc r ≡ suc r + zero !} h)
  merge′ {b} {r} (fork x b⩽x l r⩽l t u) {b′} {suc r′} (fork x′ b′⩽x′ l′ r′⩽l′ t′ u′)
    with x ⩽? x′
  merge′ {b} {r} (fork x b⩽x l r⩽l t u) {b′} {suc r′} (fork x′ b′⩽x′ l′ r′⩽l′ t′ u′)
    | yes x⩽x′ = _, lhrelax (⩽-trans {! b ⊓ b′ ⩽ b !} b⩽x)
                             (proj₂ (makeT x t (lhrelax {! x ⩽ x ⊓ x !}
                               (proj₂ (merge u (fork x′ x⩽x′ l′ r′⩽l′ t′ u′))))))
  merge′ {b} {r} (fork x b⩽x l r⩽l t u) {b′} {suc r′} (fork x′ b′⩽x′ l′ r′⩽l′ t′ u′)
    | no x≰x′ = _, lhrelax (⩽-trans {! b ⊓ b′ ⩽ b′ !} b′⩽x′)
                             (proj₂ (makeT x′ t′ (lhrelax {! x′ ⩽ x′ ⊓ x′ !}
                               (proj₂ (merge′ (fork x (≰-invert x≰x′) l r⩽l t u) u′)))))
```

Fig. 7 Merging two leftist heaps.

ments can be composed at the level of ornaments by parallel composition so the resulting promotion predicate is the pointwise conjunction of the promotion predicates of the component refinements. Such composable structure is the key to modular function upgrading, and is made possible because we can manipulate the deeper structure of refinements through ornaments. (Parallel composition is not an initial structure of ornaments, however, so strictly speaking we will need to construct a higher universe for an algebra of ornaments, one of whose constructors is parallel composition. It is premature to carry out this higher universe construction, though, before such an algebra of ornaments is properly studied.)

Parallel composition has been fully implemented in this paper, whereas the WGP version merely implemented a specialised version. We are thus able to give canonical predicates a concise definition and to define leftist heaps by composing the heap-ordering ornament in parallel with the leftist ornament, neither of which could have been done without the full power of parallel composition. Also we give projection an efficient implementation by directly distributing the content of a composite promotion proof, as opposed to the inefficient composition of forgetful and remembering maps used in the WGP version.

The idea of viewing vectors as promotion predicates was first proposed by Bernardy [2 p 82], who refers to the realisability transformation defined for pure type systems by Bernardy and Lasson [3]. 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 WGP paper can be seen as an adaptation of Bernardy’s idea into the language of ornaments without introducing levels, but also adopting the realisability terminology. We have abandoned the realisability terminology in this paper, though, as we feel that the departure from the theory of realisability is now so great that an explicit analogy seems inappropriate.

Ornaments were first proposed by McBride [10] and later adapted to index-first datatypes by Dagand and McBride [6], who also proposed *reornaments* as a more efficient representation of promotion predicates, taking full advantage of index-first datatypes. Following their suggestion, we have also adapted our work to index-first datatypes. Their reornaments are reimplemented in this paper as canonical predicates using parallel composition. Dagand and McBride [6] also extended the notion of ornaments to *functional ornaments*. Our axiomatisation of refinements and their functional ornaments are complementary and await integration: their functional ornaments can be seen as a universe for refinements generalised for function types, which will automate the insertion of isomorphisms for function upgrading as shown in their work and make the refinement approach truly worthwhile.

We have redefined ornaments to be relations between descriptions, whereas what are called "ornaments" in both works above correspond to our ornamental descriptions. Separation of ornaments from ornamental descriptions gives us the ability to state ornamental relationships between two *existing* datatypes. This ability is essential to forming the "pullback square" for parallel composition — in the WGP version we had only ornamental descriptions, and thus were forced to make the two difference ornaments produce two redundant new datatypes that are isomorphic to the one manufactured by parallel composition. Separating ornaments from ornamental descriptions also opens up the possibility of structuring descriptions and ornaments as a category with descriptions as objects and ornaments as arrows: after defining *sequential composition* of ornaments

```
_⊙_ :
  ∀ {I J K}
  {e : J → I} {f : K → J} {D E F} →
  Orn e D E → Orn f E F → Orn (e ∘ f) D F
```

and determining a suitable equivalence for ornaments, we should then be able to formulate parallel composition as a pullback in this category. Then, for example, we can take advantage of the fact that the canonical predicates are defined by parallel composition, so as to derive operations and properties about canonical predicates easily from the universal property of pullbacks. We should also be able to show that μ and *RSem* constitute a pullback-preserving functor, completing the theory.

Practically, how do we structure our libraries with ornaments and refinements for better reusability? As McBride suggested [10], the datatypes should be delivered as codes and ornaments. The datatypes on which operations are defined should be as general as possible, and other versions of the operations on more specialised types should be implemented in the form of promotion predicates. For example, *insert* should be defined for plain lists, and implemented for sorted lists and vectors as functions on proofs about ordering and length respectively. Delivered in this way, then, *insert* for sorted lists, vectors, and sorted vectors can all be derived routinely by the refinement mechanism, 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 datatypes, so datatype constraints can be manipulated with data in an integrated fashion and guide the implementation, an example being the *merge* operation for leftist heaps. This is due to the precision offered by internalism. So here is the development pattern we have in mind: once a rich collection of ornaments is 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 datatype. Existing operations are upgraded to work with the new datatype routinely by refinements. And then, operations specific to the new datatype can be programmed directly on it, benefiting from the precision of programming with inductive families.

Acknowledgements

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*. The authors would like to thank Conor McBride for offering an introduction to index-first datatypes in a *Reusability and Dependent Types* project meeting, where Thorsten Altenkirch and Peter Morris also provided very helpful comments, and the anonymous reviewers for their valuable suggestions.

References

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

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

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

[4] E. Brady, *Practical Implementation of a Dependently Typed Functional Programming Language*. PhD thesis, University of Durham, 2005.

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

[6] P.-É. Dagand and C. McBride, "Transporting functions across ornaments," In *International Conference on Functional Programming*, *ICFP '12*, pp.103–114. ACM, Sept. 2012.

[7] P. Dybjer, "Inductive families," *Formal Aspects of Computing*, vol.6, pp.440–465, 1994.

[8] H.-S. Ko and J. Gibbons, "Modularising inductive families," In Jaakko Järvi and Shin-Cheng Mu, editors, *Workshop on Generic Programming*, *WGP '11*, pp.13–24. ACM, Sept. 2011.

[9] P. M.-Löf. *Intuitionistic Type Theory*. Bibliopolis, Napoli, 1984.

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

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

[12] U. Norell, "Dependently typed programming in Agda," In P. Koopman, R. Plasmeijer, and D. Swierstra, editors, *Advanced Functional Programming*, vol.5832 of *Lecture Notes in Computer Science*, pp.230–266. Springer-Verlag, 2009.

[13] C. Okasaki. *Purely functional data structures*. Cambridge University Press, 1999.

Appendix: Agda syntax

This appendix provides a whistle-stop tour of Agda syntax, for those familiar with dependently typed programming in general but ono Agda specifically.

Function types

Let us look at a practical example of simplifying the type of the elimination (induction) principle for lists, which should help the reader to grasp the Agda syntax for function types. (The datatype definition of lists will be shown and explained later.)

1. In dependent function types, we give names to parameters, so the result type can refer to the values of those parameters. If a parameter is not referred to later, its name can be omitted. Thus, the first argument A : Set below (where Set is the type of all small types) needs to be named, because its value A is used in the result type, but in the type of the fourth parameter *ind-case*, the third argument of type $P\ xs$ need not be named, because nothing depends on its value.

```
list-elim :
  (A : Set) → (P : List A → Set) →
    (base-case : P [ ]) →
    (ind-case : (x : A) → (xs : List A) →
                  P xs → P (x :: xs)) →
      (xs : List A) → P xs
```

We sometimes give names even to parameters that are not referred to later in the code, just so that we can mention the parameters in the text.

2. Arrows between named parameters can be abbreviated, forming a *telescope*, highlighted below.

```
list-elim :
  (A : Set) (P : List A → Set) →
    (base-case : P [ ]) →
    (ind-case : (x : A) (xs : List A) →
                  P xs → P (x :: xs)) →
      (xs : List A) → P xs
```

If parameters in a telescope are of the same type, e.g., $(x : A)\ (y : A)$, then the telescope can be further condensed into $(x\ y : A)$.

3. Inferrable parameters can be marked as *implicit* by putting them into curly braces.

```
list-elim :
  {A : Set} {P : List A → Set} →
    (base-case : P [ ]) →
    (ind-case : (x : A) (xs : List A) →
                  P xs → P (x :: xs)) →
      (xs : List A) → P xs
```

A function with implicit parameters can be applied as if the implicit parameters were ignored. For example, when applying *list-elim*, we do not have to mention A and P if they are truly inferrable. If Agda cannot infer the argument to an implicit parameter, the programmer can explicitly supply an argument by putting it in curly braces, like *list-elim* $\{A\}\ \{P\}$. If we only wish to supply P and let Agda infer A, we can write *list-elim* $\{P = P\}$, in which the first P is the name of the formal parameter and the second P is the actual parameter we supply. On the other hand, if an explicit argument can be inferred, we can place an underscore to instruct Agda to infer it.

4. Parameters whose type is inferrable can be quantified by ∀ and subsequently omit their type. ∀-quantified parameters can also be collected in a telescope, and their type can still be displayed if needed.

```
list-elim :
  ∀ {A} {P : List A → Set} →
```

```
(base-case : P [ ]) →
(ind-case : ∀ x xs →
            P xs → P (x :: xs)) →
  ∀ xs → P xs
```

Datatype definitions

Agda datatype definitions employ the syntax of generalised algebraic datatypes (GADTs), the most notable feature being that the types of constructors are explicitly written. For example, the booleans are defined by

```
data Bool : Set where
  false : Bool
  true  : Bool
```

and the natural numbers by

```
data Nat : Set where
  zero : Nat
  suc  : Nat → Nat
```

The definition of lists is slightly more interesting:

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

The cons constructor is given a name _::_ which contains two underscores indicating where its two arguments can go — we can write x :: xs for _::_ x xs. This *mixfix operator* syntax works for any name, be it the name of a constructor, a function, or a datatype. There are very few restrictions on what constitutes a name in Agda — almost all unicode characters are allowed, with just a few exceptions like whitespace and parentheses. The highlighted (A : Set), which appears to the left of the colon, is a "uniform" parameter which can be used throughout the declaration. Compare this with the declaration of vectors,

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

in which the highlighted Nat, appearing to the right of the colon, is a type whose elements are used as indices of the types in the inductive family Vec A. Constructor names can be overloaded for different datatypes.

The dependent pair type is defined by

```
data Σ (A : Set) (B : A → Set) : Set where
  _,_ : (x : A) → B x → Σ A B
```

An element of Σ A B is a pair where the type of the second component depends on the value of the first component. Projections are then defined by

```
π₁ : {A : Set} {B : A → Set} → Σ A B → A
π₁ (x, y) = x
```

and

```
π₂ : {A : Set} {B : A → Set} →
     (p : Σ A B) → B (π₁ p)
π₂ (x, y) = y
```

The usual non-dependent pair type is a special case of Σ.

```
_×_ : Set → Set → Set
A × B = Σ A (λ _ → B)
```

Frequently we write types of the form Σ A (λ x → E) where the second argument is a λ-expression (in which the body E is an expression that can refer to x). We can sugar such types into Σ [x : A] E if we provide the following syntax declaration

```
syntax Σ A (λ x → E) = Σ [x : A] E
```

With this syntax, we can regard Σ [x : A] as a binder, whose scope extends as far as possible, so Σ [x : A] B x is parsed as Σ [x : A] (B x). In general such syntax declarations can be provided for the application of any (simple) names to λ-expressions.

The propositional equality type is defined by

```
data _≡_ {A : Set} (x : A) : A → Set where
  refl : x ≡ x
```

The type x ≡ y has a proof if and only if x and y can somehow be shown to be equal, as demanded by the type of its only constructor refl.

Function definitions

Functions can be defined by pattern matching as usual. For example,

```
not : Bool → Bool
not false = true
not true  = false
```

What is unusual is that performing pattern matching on a variable whose type depends on another variable may determine the value of the latter variable. For example,

```
sym : {A : Set} {x y : A} →
        (eq : x ≡ y) → y ≡ x
sym {x = x′} {.x′} refl = refl
```

First we see that implicit parameters can be explicitly mentioned if needed. We skip A and match the parameter x with the pattern variable x′. Then notice that the value of y is determined to be x' because eq is matched with refl, causing x' and y to be unified. This fact is shown by the *dot pattern* $.x'$ appearing in y's position — it indicates that the value of y is determined by unification instead of pattern matching. The goal type is thus $x' \equiv x'$ and can be solved simply by refl. (This example can actually be completed without mentioning the implicit parameters; we mention them for the purpose of illustration.)

To perform pattern matching on intermediate terms, we use the **with** construct. For example, let us look at the *insert* function used in Section 1:

```
insert : Val → List Val → List Val
insert y [ ] = y :: [ ]
insert y (x :: xs) with y ≤? x
insert y (x :: xs) | yes _ = y :: x :: xs
insert y (x :: xs) | no  _ = x :: insert y xs
```

In the $x :: xs$ case, we need to compare y and x to determine how to carry on, so we put the term $y \leqslant_? x$ after **with** as if adding it as a new argument, which is then matched with yes or no. The result of $y \leqslant_? x$ is either yes p for some $p : y \leqslant x$ or no q for some $q : y \nleqslant x$. We have no use of the proofs p and q, though, so underscores are placed after yes and no to save the trouble of giving names to the unused proofs.

Hsiang-Shang KO

Hsiang-Shang KO is a third-year doctoral student at the University of Oxford, UK, supervised by Professor Jeremy Gibbons. He is working on modularity and reusability issues in dependently typed programming, focusing particularly on ornamentation-based techniques. Previosly he finished his undergraduate studies in Computer Science and Information Engineering at National Taiwan University, Taiwan.

Geremy GIBBONS

Jeremy GIBBONS is Professor of Computing at the University of Oxford, UK, where he has been on the faculty since 1999. His research interests are in functional programming, design patterns, and domain-specific languages, and he is Director of and teaches on Oxford's part-time prefessional MSc in Software Engineering. He is also a Fellow of Kellogg College, Oxford; Chair of IFIP Working Group 2.1 on Algorithmic Languages and Calculi; Vice-Chair of ACM Special Interest Group on Programming Languages; and a Visiting Professor at the National Institute of Informatics, Japan.