*JFP* **27**, e2, 43 pages, 2016. © Cambridge University Press 2016
doi:10.1017/S0956796816000307

# *Programming with ornaments*

HSIANG-SHANG KO
*Information Systems Architecture Research Division, National Institute of Informatics, Japan*
(*e-mail:* hsiang-shang@nii.ac.jp)

JEREMY GIBBONS
*Department of Computer Science, University of Oxford, UK*
(*e-mail:* jeremy.gibbons@cs.ox.ac.uk)

**Abstract**

Dependently typed programming advocates the use of various indexed versions of the same shape of data, but the formal relationship amongst these structurally similar datatypes usually needs to be established manually and tediously. Ornaments have been proposed as a formal mechanism to manage the relationships between such datatype variants. In this paper, we conduct a case study under an ornament framework; the case study concerns programming binomial heaps and their operations — including insertion and minimum extraction — by viewing them as lifted versions of binary numbers and numeric operations. We show how current dependently typed programming technology can lead to a clean treatment of the binomial heap constraints when implementing heap operations. We also identify some gaps between the current technology and an ideal dependently typed programming language that we would wish to have for our development.

## 1 Introduction

Dependently typed programming is characterised by the use of more informative types, in particular inductive families (Dybjer, 1994) — or indexed datatypes — to guarantee program correctness by construction. Having various versions of the same kind of data, however, causes a management problem. For example, there are various kinds of list datatype, such as vectors (with embedded length constraints) and ordered lists; but formally these list datatypes and their operations are unrelated, and whenever a new variant of lists is invented, similar operations for the new variant must be developed from scratch, along with conversions to and from existing list-like datatypes. This work is tedious, and can make the programmer hesitant to employ informative types so as to avoid the hassle of managing them.

To address the variant-management problem, McBride (2011) proposed *ornaments* as a framework for relating structurally similar datatypes. An ornament specifies a particular way of relating two datatype definitions intensionally, and stating that one of them is more informative than the other; from an ornament, we can compute a forgetful function converting the more informative datatype to the less informative one. McBride also proposed *algebraic ornaments*, which provide a systematic way to synthesise new datatypes by adding indices to existing ones. McBride's work spawned a number of subsequent developments: The ornament-induced forgetful function was extended by Ko & Gibbons (2013a) to an isomorphism for promoting less informative datatypes to more informative ones; at the heart of the construction of the promotion isomorphism is a *parallel composition* operation on ornaments that, in general, can be used to combine multiple constraints on a datatype and synthesise a new variant of the datatype with all those constraints embedded at once. Dagand & McBride (2014) then proposed *functional ornaments*, generalising ornaments to work also for functions. Algebraic ornamentation has also been generalised to a relational setting, and shown to work with relational calculation techniques (Ko & Gibbons, 2013b).

In this paper, we set out to do an experiment to evaluate how effective the current ornament framework is in assisting the development of dependently typed programs, by working out a small yet complete library of binomial heaps in AGDA (Norell, 2007; Bove & Dybjer, 2009; Norell, 2009). Our experiment is tied to AGDA to some extent, as we employ AGDA-specific features like interactive programming support, pattern synonyms, and instance arguments (Devriese & Piessens, 2011), but we believe that the experiment has broader applicability since AGDA is representative of many dependently typed languages. The case study is based on Okasaki's (1999) idea of numerical representations, lifting operations on binary numbers such as increment and decrement to operations on binomial heaps such as insertion and extraction — this lifting is well supported by the ornament framework. Interestingly, the numerically developed extraction operation leads us to an algorithm for minimum extraction that is conceptually simpler than the standard one.

Before presenting the case study in Section 5, we will first give a tour of related constructions used in this paper, including index-first inductive families and their encodings in Section 2, ornaments for relating index-first inductive families in Section 3, and an extensible function-promoting mechanism that works in conjunction with ornaments in Section 4. We gloss over many technical details about the theory of ornaments and about alternative universes for them, since we intend this paper to provide a more operational/pragmatic understanding of ornaments without touching their theoretical side; more comprehensive discussions of such details can be found in the first author's DPhil thesis (Ko, 2014). Throughout the paper, there are special *Gap* paragraphs that suggest what still needs to be developed in the ornament framework or for dependently typed programming languages in general so as to give better support for dependently typed programming; Section 6 will give a summary of these gaps, and, along with other remarks, concludes this paper. Most of the code presented in this paper is collected in two supplementary AGDA source files, which have been checked with AGDA version 2.5.1.1 and standard library version 0.12.

## 2 Descriptions: encoding index-first inductive families

We first look at *index-first inductive families* (Chapman *et al.*, 2010), which are a variant of inductive families (Dybjer, 1994) that can naturally yield efficient representations. This ability will turn out to be important for the ornamental constructions in Section 3.3 to be useful in practice.

### *2.1 Index-first inductive families*

A good starting point for understanding index-first inductive families is a comparison between the usual inductive families (as supported by AGDA) and inductively defined type-computing functions. Consider vectors (length-indexed lists). In AGDA, there are two approaches to define vectors, one as an inductive family:

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

and the other as a type-computing function:

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

In the first definition, choice of constructor determines the index in the type of the constructed vector, whereas in the second definition, the index determines which constructors are valid (i.e., whether we can construct an empty vector of type $\top$ or a non-empty one of type $A \times \mathsf{Vec}\ A\ n$). The two approaches have their own pros and cons:

- The inductive family comes with an induction principle, which, roughly speaking, means that in AGDA the programmer can perform dependent pattern matching on vectors, making some subsequent programs like vector append as natural as their list counterparts. Without any optimisations (like those proposed by Brady *et al.* (2004)), however, the inductive family leads to an inefficient representation of vectors, having to store the constructor choices (whether a vector starts with nil or cons) and also the length of the tail in every cons constructor, despite the fact that the constructor choices and the tail lengths are determined by the indices.
- In contrast, the type-computing function yields an (almost) optimal representation of vectors — a vector of length $n$ is an $n$-tuple (ending with tt) and does not need to store constructor choices and tail lengths. Most interestingly, the optimisation of representation is achieved by directly expressing how the structure of a vector depends on its length, i.e., the index in its type (instead of relying on any compiler optimisations): If the index is zero, the vector must be empty; if the index is suc $n$, the vector must have a head element and a tail indexed by $n$. The approach has two drawbacks, though: This definition of vectors is dependent on the length, so whenever we want to analyse a vector we need to analyse its length first, making vector programs more verbose to write. In other words, the vectors do not come with their own induction principle, and must be analysed by induction on the natural number index. More seriously, this approach works only for situations where the type

indices grow strictly as the data grow; otherwise, the recursive definition is not well-founded.

Index-first inductive families can be regarded as reconciling type-computing functions with the usual inductive families, allowing the index of a type to influence what information can be provided to construct an element of that type, and at the same time removing the restriction that the definition should be inductive on the index. Dagand & McBride (2013; 2014) proposed a new syntax for datatype declarations mimicking type-computing functions, with which we can (choose to) do case analysis on the index and decide what constructor(s) are available for each case. For example, they would define vectors as an index-first inductive family by

```
data Vec [A : Set] (n : Nat) : Set where
  Vec_A   n       ⇐ Nat-case n
    Vec_A 0       ∋ nil
    Vec_A (suc m) ∋ cons (a : A) (vs : Vec_A m)
```

This syntax strongly depends on Epigram's pattern matching mechanism (McBride & McKinna, 2004). In this paper, we informally adopt a more Agda-like variation. For example, we describe index-first vectors in the following syntax:

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

We start the definition with the keyword **indexfirst** to mark explicitly that this is not valid Agda. The definition says that a Vec $A$ zero can be constructed only by [], whilst a Vec $A$ (suc $n$) can be constructed only by _::_, which takes two arguments of types $A$ and Vec $A$ $n$. In effect, this is the same as the type-computing function, and does not really show the full power of index-first inductive families. A more compelling example is the type family of $\lambda$-terms indexed with the number of free variables:

```
indexfirst data Term : Nat → Set where
  Term n ∋ var (Fin n)
         | lam (Term (suc n))
         | app (Term n) (Term n)
```

This definition is valid even though $n$ never decreases. In contrast, the equivalent type-computing function

```
Term : Nat → Set
Term n = Fin n ⊎ Term (suc n) ⊎ (Term n × Term n)
```

is obviously non-terminating, and would be rejected by Agda.

## 2.2 *Universe construction*

We have not really explained what index-first inductive families are; below we will do so by constructing a model of index-first inductive families with the help of a *universe* (Martin-Löf, 1984). This makes it possible to carry out (part of) our experiment with index-first datatypes without extending Agda, but ultimately we would hope that such datatypes can be natively supported.

The standard way of constructing inductive families is to take the least fixed point of base functors of type $(I \to \mathsf{Set}) \to I \to \mathsf{Set}$, where $I : \mathsf{Set}$ is the index set. For example, the base functor we use for index-first vectors (parameterised by the element type) is

$$\begin{aligned}
&\mathit{VecF} : \mathsf{Set} \to (\mathsf{Nat} \to \mathsf{Set}) \to \mathsf{Nat} \to \mathsf{Set}\\
&\mathit{VecF}\ A\ X\ \mathsf{zero} \quad = \top\\
&\mathit{VecF}\ A\ X\ (\mathsf{suc}\ n) = A \times X\ n
\end{aligned}$$

This closely resembles the type-computing function except that the recursive reference to $\mathsf{Vec}\ A$ is replaced by an invocation of the extra argument $X$. $\mathsf{Vec}\ A$ is then defined as the least fixed point of $\mathit{VecF}\ A$, implying that $\mathsf{Vec}\ A$ and $\mathit{VecF}\ A\ (\mathsf{Vec}\ A)$ are isomorphic. In particular, $\mathsf{Vec}\ A\ \mathsf{zero}$ is isomorphic to $\mathit{VecF}\ A\ (\mathsf{Vec}\ A)\ \mathsf{zero} = \top$, and $\mathsf{Vec}\ A\ (\mathsf{suc}\ n)$ is isomorphic to $\mathit{VecF}\ A\ (\mathsf{Vec}\ A)\ (\mathsf{suc}\ n) = A \times \mathsf{Vec}\ A\ n$, exactly corresponding to how we defined index-first vectors previously.

Not all functions of type $(I \to \mathsf{Set}) \to I \to \mathsf{Set}$ are valid base functors, and we will constrain ourselves by fixing the ways in which functions of type $(I \to \mathsf{Set}) \to I \to \mathsf{Set}$ can be constructed. This is where universe construction can help: By defining a datatype whose elements encode ways to construct valid base functors, and later quantifying over this datatype, we will be able to carry out our subsequent constructions for only the encoded base functors. For index-first inductive families, it turns out that we can encode just the "response" part of base functors: Using the base functor for vectors as an example, after swapping the index and recursive reference arguments:

$$\begin{aligned}
&\mathit{VecF}' : \mathsf{Set} \to \mathsf{Nat} \to (\mathsf{Nat} \to \mathsf{Set}) \to \mathsf{Set}\\
&\mathit{VecF}'\ A\ \mathsf{zero} \quad = \lambda X \mapsto \top\\
&\mathit{VecF}'\ A\ (\mathsf{suc}\ n) = \lambda X \mapsto A \times X\ n
\end{aligned}$$

the parts on the right-hand side of the equations are referred to as "responses", suggesting an interactive interpretation of base functors — $\mathit{VecF}'\ A$ receives and analyses an index before it responds with what information is required to construct an element whose type has the given index. We will shortly define a universe RDesc whose elements — called *response descriptions* — encode a range of such responses:

$$\begin{aligned}
&\mathsf{RDesc} : \mathsf{Set} \to \mathsf{Set}_1\\
&[\![\_]\!] \quad : \{I : \mathsf{Set}\} \to \mathsf{RDesc}\ I \to (I \to \mathsf{Set}) \to \mathsf{Set}
\end{aligned}$$

Base functors will then be encoded by

$$\begin{aligned}
&\mathsf{Desc} : \mathsf{Set} \to \mathsf{Set}_1\\
&\mathsf{Desc}\ I = I \to \mathsf{RDesc}\ I\\
&\mathbb{F} : \{I : \mathsf{Set}\} \to \mathsf{Desc}\ I \to (I \to \mathsf{Set}) \to (I \to \mathsf{Set})\\
&\mathbb{F}\ D\ X\ i = [\![\ D\ i\ ]\!]\ X
\end{aligned}$$

and we can take the least fixed point of the encoded base functors by

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

where $X \rightrightarrows Y$ abbreviates $\{i : I\} \to X\ i \to Y\ i$.

The datatype of response descriptions and its decoding function determine the range of base functors we can describe. For this paper, we focus on inductive families that are finitely branching, captured by the following definition:

```
data RDesc (I : Set) : Set₁ where
  v : (is : List I) → RDesc I
  σ : (S : Set) (D : S → RDesc I) → RDesc I
⟦_⟧ : {I : Set} → RDesc I → (I → Set) → Set
⟦ v is    ⟧ X = ℙ is X    -- see below
⟦ σ S D ⟧ X = Σ[ s ∈ S ] ⟦ D s ⟧ X
```

The operator ℙ computes the product of a finite number of types in a type family, whose indices are given in a list:

```
ℙ : {I : Set} → List I → (I → Set) → Set
ℙ []       X = ⊤
ℙ (i :: is) X = X i × ℙ is X
```

Thus, in a response, given $X : I \to$ Set, we are allowed to form dependent sums (by σ) and the product of a finite number of types in $X$ (via v, suggesting variable positions in the base functor). We will informally refer to the index part of a σ as a *field* of the datatype. For convenience, we define an abbreviation:

```
syntax σ S (λ s ↦ D) = σ[ s ∈ S ] D
```

so as to use σ as a binder.

### 2.3 *Encoding datatypes as descriptions*

Let us now look at several kinds of datatype and how they can be encoded as descriptions.

**Non-indexed datatypes.** A non-indexed datatype can be seen as an inductive family trivially indexed by ⊤, and hence can be encoded as an inhabitant of Desc ⊤. For example, the datatype of natural numbers

```
indexfirst data Nat : Set where
  Nat ∋ zero
      | suc Nat
```

can be encoded as follows:

```
data ListTag : Set where
  ‘nil  : ListTag
  ‘cons : ListTag
```

$$\begin{aligned}
&\mathit{NatD}\ &&:\ \mathsf{Desc}\ \top \\
&\mathit{NatD}\ \mathsf{tt}\ &&=\ \sigma\ \mathsf{ListTag}\ \lambda\ \{\ \text{‘}\mathsf{nil}\ \ \ \mapsto\ \mathsf{v}\ [] \\
& && \qquad\qquad\qquad\ \ ;\ \text{‘}\mathsf{cons}\ \mapsto\ \mathsf{v}\ (\mathsf{tt} :: [])\ \}
\end{aligned}$$

The index request is necessarily tt, and we respond with a field of type ListTag representing the constructor choices. (Any two-element set can be used to represent the constructor choices; the reason that we choose the names ListTag, ‘nil, and ‘cons will become clear when we relate natural numbers with lists in Section 3.) A pattern-matching lambda function follows, which computes the trailing responses to the two possible values ‘nil and ‘cons for the field: If the field receives ‘nil, then we are constructing zero, which takes no recursive values, so we write v [] to end this branch; if the ListTag field receives ‘cons, then we are constructing a successor, which takes a recursive value at index tt, so we write v (tt :: []).

To see that we have indeed defined the type of natural numbers, we can try to construct an inhabitant of $\mu$ *NatD* tt interactively:

$$\begin{aligned}
&n\ :\ \mu\ \mathit{NatD}\ \mathsf{tt} \\
&n\ =\ \{\ \mu\ \mathit{NatD}\ \mathsf{tt}\ \}_0
\end{aligned}$$

The shaded box is called an *interaction point*, or informally a *hole* in the program. Shown in the hole is the goal type, for which we have to construct an appropriate value. The subscript number is used to disambiguate when there are multiple holes.

To construct a $\mu$-value, a natural — indeed, a forced — choice is to use the only constructor con:

$$\begin{aligned}
&n\ :\ \mu\ \mathit{NatD}\ \mathsf{tt} \\
&n\ =\ \mathsf{con}\ \{\ \mathbb{F}\ \mathit{NatD}\ (\mu\ \mathit{NatD})\ \mathsf{tt}\ \}_1
\end{aligned}$$

The goal type computes to $[\![\ \mathit{NatD}\ \mathsf{tt}\ ]\!]\ (\mu\ \mathit{NatD})$, which, if we compute under lambda, normalises to

$$\begin{aligned}
&\Sigma\ \mathsf{ListTag}\ \lambda\ \{\ \text{‘}\mathsf{nil}\ \ \ \mapsto\ \top \\
&\qquad\qquad\quad\ ;\ \text{‘}\mathsf{cons}\ \mapsto\ \mu\ \mathit{NatD}\ \mathsf{tt}\ \times\ \top\ \}
\end{aligned}$$

Goal 1 thus expects a (dependent) pair whose first component has type ListTag, indicating whether this con node is a zero node or a successor node. Here, we choose to construct a successor by supplying ‘cons, which determines the type of the second component:

$$\begin{aligned}
&n\ :\ \mu\ \mathit{NatD}\ \mathsf{tt} \\
&n\ =\ \mathsf{con}\ (\text{‘}\mathsf{cons}\ ,\ \{\ \mu\ \mathit{NatD}\ \mathsf{tt}\ \times\ \top\ \}_2)
\end{aligned}$$

Now, we should construct an inner inhabitant of $\mu$ *NatD* tt (paired with a tt) by going through the above process again. We can choose to supply ‘cons several times, but eventually we should choose to supply ‘nil to end the construction:

$$\begin{aligned}
&n\ :\ \mu\ \mathit{NatD}\ \mathsf{tt} \\
&n\ =\ \mathsf{con}\ (\text{‘}\mathsf{cons}\ ,\ \mathsf{con}\ (\text{‘}\mathsf{cons}\ ,\ \mathsf{con}\ (\text{‘}\mathsf{cons}\ ,\ \mathsf{con}\ (\text{‘}\mathsf{nil}\ ,\ \{\ \top\ \}_3)\ ,\ \mathsf{tt})\ ,\ \mathsf{tt})\ ,\ \mathsf{tt})
\end{aligned}$$

Filling tt into Goal 3 ends the construction. It should be evident now that inhabitants of $\mu$ *NatD* tt indeed represent natural numbers, with con (‘nil , tt) : $\mu$ *NatD* tt being zero and $\lambda\, m \mapsto \mathsf{con}\,(\text{‘}\mathsf{cons}\ ,\ m\ ,\ \mathsf{tt}) : \mu\ \mathit{NatD}\ \mathsf{tt} \to \mu\ \mathit{NatD}\ \mathsf{tt}$ being the successor function.

**Parameterised datatypes.** Parameterised datatypes can simply be encoded as functions mapping parameters to descriptions. For example, the datatype of lists is parameterised by the element type:

```
indexfirst data List (A : Set) : Set where
  List A ∋ []
          | _::_ A (List A)
```

This can be encoded as a function mapping element types to descriptions:

```
ListD     : Set → Desc ⊤
ListD A tt = σ ListTag λ { ‘nil    ↦ v []
                          ; ‘cons  ↦ σ[ _ ∈ A ] v (tt :: []) }
```

*ListD A* is the same as *NatD* except that, in the ‘cons case, we use σ to insert a field of type *A* for storing an element.

**Indexed datatypes.** When a datatype is non-trivially indexed, we can perform arbitrary computation on the index request to produce suitable responses, but for most situations, the kind of computation we need is just pattern matching. For example, the datatype of vectors is encoded as

```
VecD : Set → Desc Nat
VecD A zero    = v []
VecD A (suc n) = σ[ _ ∈ A ] v (n :: [])
```

which is directly comparable to the index-first base functor $\mathit{VecF}'$.

There are more sophisticated scenarios requiring extra information to be supplied in the descriptions. For example, consider the following variant of finite sets:

```
indexfirst data Fin′ : Nat → Set where
  Fin′ n       ∋ zero
  Fin′ (suc n) ∋ suc (Fin′ n)
```

The inhabitants of Fin′ $n$ are natural numbers less than or equal to $n$: Regardless of what $n$ is, zero is always an available constructor; if we know that $n$ is a successor, then we can also choose to construct an inhabitant by suc. One way to encode this kind of datatype is to distinguish the responses that are always available from those that are available subject to the result of pattern matching, and the distinction can be encoded as an implicit constructor tag (not appearing explicitly in the datatype declaration):

```
data ConMenu : Set where
  ‘always  : ConMenu
  ‘indexed : ConMenu
Fin′D   : Desc Nat
Fin′D n = σ ConMenu λ { ‘always  ↦ v []
```

```
                              ; ‘indexed ↦ case n of λ { zero     ↦ Σ ⊥ λ ()
                                                      ; (suc m) ↦ v (m :: []) } }
```

Note that the empty response $\Sigma \perp \lambda$ () produced in the case where $n$ is pattern-matched with zero is also implicitly inserted to make the function total.

*Gap 1* (*elaboration of datatype declarations to descriptions*)
Dagand & McBride (2013) presented an elaboration mechanism for encoding index-first datatype declarations to descriptions, but that mechanism needs to be further extended to deal with more sophisticated cases like always-available constructors and incomplete pattern matching on the index. To circumvent the gap in this paper, we will present an index-first datatype declaration side by side with its encoding as a description, leaving the formal connection between them unspecified.

### *2.4 Functions on index-first datatypes*

There is no problem defining functions on the encoded datatypes, except that it has to be done with the raw representation. This is easy when the datatypes are not defined by pattern matching on their index. For example, list append can be defined by

```
_++_ : μ (ListD A) tt → μ (ListD A) tt → μ (ListD A) tt
con (‘nil   ,          tt) ++ ys  =  ys
con (‘cons , x , xs , tt) ++ ys  =  con (‘cons , x , xs ++ ys , tt)
```

To improve readability, we can define the following abbreviations, exploiting Agda’s ability to define “pattern synonyms”:

```
List : Set → Set
List A  =  μ (ListD A) tt
pattern [] = con (‘nil , tt)
pattern _::_ a as  =  con (‘cons , a , as , tt)
```

List append can then be rewritten in the usual form:

```
_++_ : List A → List A → List A
[]         ++ ys  =  ys
(x :: xs) ++ ys  =  x :: (xs ++ ys)
```

The situation gets trickier when the datatypes are defined by pattern matching on their index. Consider vector append, for example:

```
_++_ : {m n : Nat} → μ (VecD A) m → μ (VecD A) n → μ (VecD A) (m + n)
con xs ++ ys  =  { }0
```

We cannot further decompose $xs$ because its type gets stuck at $[\![$ *VecD A m* $]\!]$ ($\mu$ (*VecD A*)). In other words, before $m$ is known to have a more specific form, we cannot determine which vector constructor is applicable here. We thus need to perform pattern matching on $m$ before we can proceed:

```
_++_ : {m n : Nat} → μ (VecD A) m → μ (VecD A) n → μ (VecD A) (m + n)
_++_ {m = zero  } (con tt           ) ys = ys
_++_ {m = suc m} (con (x , xs , tt)) ys = con (x , xs ++ ys , tt)
```

This is obviously inferior to what we can already do with AGDA's native datatypes: The append program for the vectors defined as a native AGDA datatype is exactly the same as the list append program, using patterns corresponding to the declared constructors, whereas the above definition has to deal with indices first.

*Gap 2* (*elaboration of pattern matching on index-first data*)
Type-theoretically, pattern matching can be explained in terms of the elimination rule, i.e., the induction principle, as done by McBride & McKinna (2004), Goguen *et al.* (2006), and Cockx *et al.* (2014). There is a generic induction principle for index-first datatypes (see, e.g., Section 5.1 of Chapman *et al.* (2010)), but that induction principle does not directly correspond to "proper" pattern matching (against declared constructors, rather than having to analyse the index first), resulting in a gap to be bridged. It is possible to simulate in AGDA proper pattern matching for the encoded index-first datatypes to some extent by, e.g., employing McBride and McKinna's view idiom, but the syntax is still far from transparent. In Section 5.5, we will experiment with writing more complex programs; to employ a natural pattern matching notation (and also get more readable type information at interaction points), these programs are actually implemented on AGDA's native versions of the datatypes in the experimental code.

## 3 Ornaments: relating structurally similar datatypes

Informally, ornaments relate one datatype with another that has the same recursive structure but is more informative. Here, "more informative" refers to two things: having more fields and more refined indices. For example, there is an ornament from natural numbers to lists, as the latter has an extra field associated with the cons constructor; and there is also an ornament from lists to vectors, as the latter partitions the type of lists into a family of types indexed by length. On the other hand, there is no ornament from Peano-style natural numbers to binary trees, because they have different recursive structures, one being linear and the other branching.

More formally, ornaments are typed as

```
Orn : {I J : Set} (e : J → I) (D : Desc I) (E : Desc J) → Set₁
```

An ornament of type Orn $e$ $D$ $E$ is a relation from a less informative description $D$ : Desc $I$ to a more informative description $E$ : Desc $J$. Logically prior to forming the ornaments, a refining relation on the index sets $I$ and $J$ is first specified, in the form of a function $e$ of type $J \to I$ (which appears in the type of the ornaments), so every $J$-index is related to exactly one $I$-index. Since $D$ and $E$ are collections of response descriptions, an ornament from $D$ to $E$ is actually a $J$-indexed collection of refining relations from $D\ (e\ j)$ to $E\ j$. For technical (and partly historical) reasons, we use an auxiliary datatype Inv (defined in Figure 1) to index this collection, such that Inv $e$ $i$ contains those elements of $J$ that are mapped to $i$ by $e$:

```
data Inv {A B : Set} (f : A → B) : B → Set where
  ok : (x : A) → Inv f (f x)

und : {A B : Set} {f : A → B} {y : B} → Inv f y → A
und (ok x) = x

data 𝔼 {I J : Set} (e : J → I) : List J → List I → Set where
  []    : 𝔼 e [] []
  _::_ : {j : J} {i : I} (eq : e j ≡ i)
          {js : List J} {is : List I} (eqs : 𝔼 e js is) → 𝔼 e (j :: js) (i :: is)

data ROrn {I J : Set} (e : J → I) : RDesc I → RDesc J → Set₁ where
  v : {js : List J} {is : List I} (eqs : 𝔼 e js is) → ROrn e (v is) (v js)
  σ : (S : Set) {D : S → RDesc I} {E : S → RDesc J}
      (O : (s : S) → ROrn e (D s) (E s)) → ROrn e (σ S D) (σ S E)
  Δ : (T : Set) {D : RDesc I} {E : T → RDesc J}
      (O : (t : T) → ROrn e D (E t)) → ROrn e D (σ T E)
  ∇ : {S : Set} (s : S) {D : S → RDesc I} {E : RDesc J}
      (O : ROrn e (D s) E) → ROrn e (σ S D) E
```

Fig. 1. Definitions for response ornaments.

```
Orn : {I J : Set} (e : J → I) (D : Desc I) (E : Desc J) → Set₁
Orn {I} e D E = {i : I} (j : Inv e i) → ROrn e (D i) (E (und j))
```

The datatype ROrn of *response ornaments*, which is where we place the actual definition of how two datatypes are related, is shown in Figure 1. (This figure is intended to provide type information for the constructors. In Section 3.1 below, we will explain the constructors by going through concrete examples and showing how to use the constructors to relate datatypes, when the type information will become helpful.) One construction derived from an ornament is a forgetful function that turns an inhabitant of the more informative datatype to one of the less informative datatype, retaining the recursive structure:

```
ornForget : {I J : Set} {e : J → I} {D : Desc I} {E : Desc J}
            (O : Orn e D E) → μ E ⇉ (μ D ∘ e)
```

For example, the forgetful function derived from the ornament from natural numbers to lists is the *length* function, and the forgetful function derived from the ornament from lists to vectors computes the underlying list of a vector. Note that the type of *ornForget* gives a reason for the index-refining relation to be specified as a function $e$: For any index $j$, an inhabitant of type $\mu\ E\ j$ can be transformed into $\mu\ D\ (e\ j)$.

### *3.1 Understanding ornaments by examples*

The most interesting feature of ornaments is their intensional (syntactic) nature: An ornament can be seen as a field-by-field comparison between two datatypes, akin to a "diff" between two descriptions of types Desc $I$ and Desc $J$. When writing an ornament, imagine that we are consuming the two descriptions being compared, relating them field-by-field and index-by-index. Below, we will go through some examples to bring out this intuition, and also explain how the constructors of ROrn are intended to be used.

Our first example is an ornament from natural numbers to lists whose elements are of a given type $A$. To specify the type of the ornament, first we need to determine the refining relation on the index sets; in this case, there is only one choice, namely $! = \lambda\_ \mapsto$ tt, because naturals are trivially indexed.

```
NatD-ListD   : (A : Set) → Orn ! NatD (ListD A)
NatD-ListD A = { Orn ! NatD (ListD A) }0
```

An ornament is a collection of response ornaments indexed by the more informative indices; in this case, again, there is only one such index, namely tt.

```
NatD-ListD : (A : Set) → Orn ! NatD (ListD A)
NatD-ListD A (ok tt) = { ROrn ! (NatD tt) (ListD A tt) }1
```

Now, we should relate the response descriptions *NatD* tt and *ListD A* tt, shown as indices in the goal type. Both response descriptions (shown in Section 2.3) start with a common field of type ListTag. To indicate that the field is a common one in both response descriptions, we use the σ constructor of ROrn:

```
NatD-ListD : (A : Set) → Orn ! NatD (ListD A)
NatD-ListD A (ok tt) = σ ListTag { (t : ListTag) → ROrn ! (D′ t) (E′ t) }2
    -- where D′ = λ {‘nil ↦ v [] ; ‘cons ↦ v (tt :: [])}
    -- and E′ = λ {‘nil ↦ v [] ; ‘cons ↦ σ[ _ ∈ A ] v (tt :: [])}
```

To make $D'\ t$ and $E'\ t$ reduce, we pattern-match on $t$, splitting the goal into two:

```
NatD-ListD : (A : Set) → Orn ! NatD (ListD A)
NatD-ListD A (ok tt) =
  σ ListTag λ { ‘nil  ↦ { ROrn ! (v []) (v []) }3
              ; ‘cons ↦ { ROrn ! (v (tt :: [])) (σ[ _ ∈ A ] v (tt :: [])) }4 }
```

Let us look at Goal 4 first. There is a field of type $A$ (for storing list elements) only on the more informative side and not on the less informative side; to indicate this, we use the Δ constructor:

```
NatD-ListD : (A : Set) → Orn ! NatD (ListD A)
NatD-ListD A (ok tt) =
  σ ListTag λ { ‘nil  ↦ { ROrn ! (v []) (v []) }3
              ; ‘cons ↦ Δ[ _ ∈ A ] { ROrn ! (v (tt :: [])) (v (tt :: [])) }5 ) }
```

Here, again we regard Δ as a binder and write $\Delta[\, t \in T\,]\ O\ t$ for $\Delta\ T\ (\lambda t \mapsto O\ t)$. Now, both goals require us to relate variable positions, which is done by the v constructor of ROrn:

```
NatD-ListD : (A : Set) → Orn ! NatD (ListD A)
NatD-ListD A (ok tt) =
  σ ListTag λ { ‘nil  ↦ v { 𝔼 ! [] [] }6
              ; ‘cons ↦ Δ[ _ ∈ A ] v { 𝔼 ! (tt :: []) (tt :: []) }7 }
```

The datatype $\mathbb{E}\ e\ js\ is$ is just a more structured representation of the equality $map\ e\ js \equiv is$, guaranteeing that the recursive structure is the same (i.e., the two index lists have the same length) and that the corresponding pairs of indices respect the refining relation $e$. Here, the lists in each goal are indeed of the same length, and the indices in Goal 7 are trivial, so both goals can be automatically discharged.

```
NatD-ListD : (A : Set) → Orn ! NatD (ListD A)
NatD-ListD A (ok tt) =
  σ ListTag λ { ‘nil  ↦ v []
              ; ‘cons ↦ Δ[ _ ∈ A ] v (refl :: []) }
```

The forgetful function *ornForget* (*NatD*-*ListD* *A*) discards the $A$-typed element associated with each cons node (because the field is marked using Δ as being exclusive to the description of lists) and keeps everything else, including the recursive structure (a series of ‘cons nodes ending with a ‘nil node). That is, it computes the length of a list.

Our second example is an ornament from lists to vectors, starting from the following ornament type:

```
ListD-VecD : (A : Set) → Orn ! (ListD A) (VecD A)
ListD-VecD A (ok n) = { ROrn ! (ListD A tt) (VecD A n) }0
```

An ornament is a collection of response ornaments, one for each index of the more informative datatype — in this case, vectors, which are indexed by natural numbers.

To make *VecD A n* reduce, we pattern-match on $n$:

```
ListD-VecD : (A : Set) → Orn ! (ListD A) (VecD A)
ListD-VecD A (ok zero   ) = { ROrn ! (ListD A tt) (v []) }1
ListD-VecD A (ok (suc n)) = { ROrn ! (ListD A tt) (σ[ _ ∈ A ] v (n :: [])) }2
```

This time there is a field of type ListTag that appears only on the less informative side, namely in *ListD A* tt. The value for this field, however, should be fixed for both goals: For Goal 1, it should be ‘nil, and for Goal 2 it should be ‘cons (otherwise, the goals will be unachievable). To indicate so, we use the ∇ constructor:

```
ListD-VecD : (A : Set) → Orn ! (ListD A) (VecD A)
ListD-VecD A (ok zero   ) = ∇[‘nil ]  { ROrn ! (v []) (v []) }3
ListD-VecD A (ok (suc n)) = ∇[‘cons]  { ROrn ! (σ[ _ ∈ A ] v (tt :: []))
                                                (σ[ _ ∈ A ] v (n :: [])) }4
```

Even though ∇ is not a binder, we write ∇[ $s$ ] $O$ for ∇ $s$ $O$ to avoid the parentheses around $O$ when $O$ is a complex expression. Goal 3 is easily discharged using v. As for Goal 4, we use σ to relate the element field on both sides, and then use v to end the ornament.

```
ListD-VecD : (A : Set) → Orn ! (ListD A) (VecD A)
ListD-VecD A (ok zero   ) = ∇[‘nil ]  v []
ListD-VecD A (ok (suc n)) = ∇[‘cons]  σ[ _ ∈ A ] v (refl :: [])
```

The forgetful function *ornForget* (*ListD*-*VecD* *A*) erases the index, reinstalls the constructor tags ‘nil and ‘cons, and keeps everything else. That is, it computes the underlying list of a vector.

```
OrnDesc : {I : Set} (J : Set) (e : J → I) (D : Desc I) → Set₁
OrnDesc {I} J e D = {i : I} (j : Inv e i) → ROrnDesc J e (D i)

data ROrnDesc {I : Set} (J : Set) (e : J → I) : RDesc I → Set₁ where
  v : {is : List I} (js : ℙ is (Inv e)) → ROrnDesc J e (v is)
  σ : (S : Set) {D : S → RDesc I}
      (O : (s : S) → ROrnDesc J e (D s)) → ROrnDesc J e (σ S D)
  Δ : (T : Set) {D : RDesc I}
      (O : T → ROrnDesc J e D) → ROrnDesc J e D
  ∇ : {S : Set} (s : S) {D : S → RDesc I}
      (O : ROrnDesc J e (D s)) → ROrnDesc J e (σ S D)
```

Fig. 2. Definitions for ornamental descriptions.

### 3.2 Ornamental descriptions

Often we want to define a more informative datatype that is an ornamentation of an existing datatype. *Ornamental descriptions* are provided for conveniently writing both the description and the ornament in a single definition. If an ornament is a “diff” between two descriptions, then an ornamental description is a “patch” specified using an existing description as a template, and this patch can be “applied” to produce a more informative description. The definitions are shown in Figure 2. Note that OrnDesc is indexed by only one template description. Historically, ornamental descriptions were in fact McBride’s (2011) original formulation of ornaments, but we needed to formulate ornaments as binary relations between descriptions (Ko & Gibbons, 2013a) in order to define parallel composition (Section 3.3) as a categorical pullback.

Suppose we wish to write an ornamental description of ordered lists indexed by a lower bound, using the description of lists as a template. Also, suppose that the list elements are of type *Val*, on which there is an ordering relation _≤_ : *Val* → *Val* → Set. Eventually, this ornamental description should give us the following datatype:

```
indexfirst data OrdList : Val → Set where
  OrdList b ∋ nil
            | cons (x : Val) (b ≤ x) (OrdList x)
```

OrnDesc, like Orn, is an indexed collection of response ornamental descriptions of type ROrnDesc. We start by assuming that the lower bound is *b*:

```
OrdListOD : OrnDesc Val ! (ListD Val)
OrdListOD (ok b) = { ROrnDesc Val ! (ListD Val tt) }₀
```

The first field of *ListD Val* tt is ListTag, which we should keep by using the σ constructor of ROrnDesc:

```
OrdListOD : OrnDesc Val ! (ListD Val)
OrdListOD (ok b) = σ ListTag
  { (t : ListTag) →
    ROrnDesc Val ! ((λ { ‘nil ↦ v [] ; ‘cons ↦ σ[_ ∈ Val] v (tt :: []) }) t) }₁
```

Again we should pattern-match on $t$ to reveal the two cases for the two list constructors:

```
OrdListOD  : OrnDesc Val ! (ListD Val)
OrdListOD (ok b)  =
   σ ListTag λ { ‘nil    ↦  { ROrnDesc Val ! (v []) }2
               ; ‘cons   ↦  { ROrnDesc Val ! (σ[ _ ∈ Val ] v (tt :: [])) }3 }
```

Goal 2 can be met using the v constructor of ROrnDesc. Goal 3, which corresponds to the refinement of the cons constructor, is more interesting. We first use σ to keep the element field:

```
OrdListOD  : OrnDesc Val ! (ListD Val)
OrdListOD (ok b)  =
   σ ListTag λ { ‘nil    ↦  v tt
               ; ‘cons   ↦  σ[ x ∈ Val ]  { ROrnDesc Val ! (v (tt :: [])) }4 }
```

We should not close Goal 4 just yet, because we should require that the element $x$ is bounded below by $b$. This is done by inserting a new field of type $b \leqslant x$ by using the Δ constructor of ROrnDesc:

```
OrdListOD  : OrnDesc Val ! (ListD Val)
OrdListOD (ok b)  =
   σ ListTag λ { ‘nil    ↦  v tt
               ; ‘cons   ↦  σ[ x ∈ Val ]  Δ[ _ ∈ (b ⩽ x) ]
                                { ROrnDesc Val ! (v (tt :: [])) }5 }
```

Now, we can refine the index for the tail to $x$, meaning that the tail should be bounded below by $x$:

```
OrdListOD  : OrnDesc Val ! (ListD Val)
OrdListOD (ok b)  =
   σ ListTag λ { ‘nil    ↦  v tt
               ; ‘cons   ↦  σ[ x ∈ Val ]  Δ[ _ ∈ (b ⩽ x) ]  v (ok x , tt) }
```

Note that the argument to v is a tuple containing exactly as many indices as those in the index list in the template description, keeping the recursive branching structure by construction.

We can interpret an ornamental description using ⌊_⌋ to extract a description of type Desc *Val* of the new datatype:

```
⌊OrdListOD⌋ b  =
   σ ListTag λ { ‘nil    ↦  v []
               ; ‘cons   ↦  σ[ x ∈ Val ]  σ[ _ ∈ (b ⩽ x) ]  v (x :: []) }
```

Alternatively, we can interpret the ornamental description using ⌈_⌉ to extract an ornament of type Orn ! (*ListD Val*) ⌊*OrdListOD*⌋ from the template description to the new description:

$\lceil \mathit{OrdListOD} \rceil\ (\mathsf{ok}\ b) =$
$\quad \sigma\ \mathsf{ListTag}\ \lambda\ \{\ \text{‘}\mathsf{nil} \mapsto \mathsf{v}\ []$
$\qquad\qquad\qquad ;\ \text{‘}\mathsf{cons} \mapsto \sigma[x \in \mathit{Val}]\ \Delta[\_ \in (b \leqslant x)]\ \mathsf{v}\ (\mathsf{refl} :: [])\ \}$

*Gap 3* (*interactive support for constructing of ornaments and ornamental descriptions*) Both Dagand & McBride (2014) and Williams *et al.* (2014) propose concrete syntax for defining ornamental descriptions, and Williams *et al.* also devise an "`ornament from`" mechanism for defining ornaments (between two existing datatypes, rather than creating a new datatype on top of an existing one) in terms of forgetful functions, which should conform to some syntax restrictions. We have shown that the construction of ornaments (and ornamental descriptions) can be performed in an interactive manner, so perhaps ornaments can also be constructed in a special tactic language, in which the programmer can go through the interactive construction process by giving instructions linking two existing datatype declarations, and get valid ornaments by construction.

## 3.3 Ornamental promotion isomorphisms

The syntactic nature of ornaments allows us to compute from them things other than the forgetful function. Indeed, the forgetful function can be seen as part of an isomorphism connecting the two datatypes related by an ornament: From an ornament $O : \mathsf{Orn}\ e\ D\ E$, we can compute an *optimised promotion predicate* $\mathsf{OptP}\ O : \{i : I\}\ (j : \mathsf{Inv}\ e\ i) \to \mu\ D\ i \to \mathsf{Set}$ that captures the necessary information for augmenting an inhabitant of the less informative datatype to an inhabitant of the more informative datatype. More precisely, for every $j : \mathsf{Inv}\ e\ i$ there is an isomorphism

$$\mu\ E\ (\mathit{und}\ j) \cong \Sigma\ (\mu\ D\ i)\ (\mathsf{OptP}\ O\ j)$$

which says that an inhabitant of the more informative datatype is interconvertible with an inhabitant of the less informative datatype paired with a proof that it satisfies the optimised promotion predicate. A representative example is the ornament from lists to ordered lists. In this case, the optimised promotion predicate computed from the ornament is

$\textbf{indexfirst data}\ \mathsf{Ordered} : \mathit{Val} \to \mathsf{List}\ \mathit{Val} \to \mathsf{Set}\ \textbf{where}$
$\quad \mathsf{Ordered}\ b\ [] \qquad\quad \ni\ \mathsf{nil}$
$\quad \mathsf{Ordered}\ b\ (x :: xs) \ni\ \mathsf{cons}\ (\mathit{leq} : b \leqslant x)\ (\mathit{ord} : \mathsf{Ordered}\ x\ xs)$

An inhabitant of $\mathsf{Ordered}\ b\ xs$ is exactly a series of inequality proofs that chains together the lower bound $b$ and all the elements of $xs$ in order. Given such a series of inequality proofs, we can promote $xs$ to an $\mathsf{OrdList}\ b$; conversely, any $\mathsf{OrdList}\ b$ can be separated into a list $xs$ and a series of inequality proofs of type $\mathsf{Ordered}\ b\ xs$. That is, for every $b$, there is an isomorphism

$$\mathsf{OrdList}\ b \cong \Sigma\ (\mathsf{List}\ \mathit{Val})\ (\mathsf{Ordered}\ b)$$

We call this the *ornamental promotion isomorphism.*

We could construct the optimised promotion predicate and the isomorphism directly, as Dagand & McBride (2014) do with "reornaments", but in fact there is a more general construction — *parallel composition* of ornaments. If we think of ornaments as diffs, then parallel composition is analogous to three-way merging: Given two ornaments $O$ : Orn $e$ $D$ $E$ and $P$ : Orn $f$ $D$ $F$ with the same less informative end $D$, we can compute an ornamental description $O \otimes P$ relative to $D$ that incorporates all modifications to $D$ recorded in $O$ and in $P$. For example, we can compose in parallel the ornaments from lists to ordered lists and to vectors to get ordered vectors:

```
indexfirst data OrdVec : Val → Nat → Set where
  OrdVec b zero    ∋ nil
  OrdVec b (suc n) ∋ cons (x : Val) (leq : b ⩽ x) (xs : OrdVec x n)
```

With parallel composition, the optimised promotion predicate for an ornament $O$ : Orn $e$ $D$ $E$ can easily be defined as the parallel composition of $O$ and the *singleton* ornamentation for $D$. Here, we give only a quick and intuitive explanation. (For a more detailed account, see Section 3.3.1 of Ko (2014).) The singleton ornamentation for $D$ gives the singleton datatype (Sheard & Linger, 2007) indexed by $\mu$ $D$, and each type in the family has exactly one inhabitant. For example, the singleton datatype for lists is

```
indexfirst data ListS (A : Set) : List A → Set where
  ListS A []        ∋ nil
  ListS A (x :: xs) ∋ cons (ListS A xs)
```

This datatype can be obtained from the following ornamental description:

```
ListSOD : (A : Set) → OrnDesc (List A) ! (ListD A)
ListSOD A (ok []        ) = ∇[‘nil]       v tt
ListSOD A (ok (x :: xs)) = ∇[‘cons] ∇[x] v (ok xs , tt)
```

Note that all the fields are deleted. What ornamental description do we get if we compose this ornament in parallel with the one from lists to ordered lists? The extra field for ordered lists storing inequality proofs will be inserted, but the fields for lists storing constructor choices and elements will be deleted. Thus, the resulting datatype Ordered is left with only the extra field for ordered lists, and inhabitants of Ordered store only the inequality proofs.

We do not give the construction of the ornamental isomorphism in this paper, but instead refer the reader to Chapter 4 of Ko (2014), where parallel composition is shown to be a categorical pullback. The ornamental isomorphism can then be constructed from the pullback property, independently of the choice of universe encoding. It is worth mentioning that an arbitrary choice of encoding may make it impossible to establish the pullback property: Indeed, the encoding used in Ko & Gibbons (2013a) does not satisfy the pullback property because there is a product constructor _*_ : RDesc $I$ → RDesc $I$ → RDesc $I$ that turns out to be problematic; we thus switched to the version presented by Ko (2014) in this paper, in particular enforcing that all the recursive positions are grouped in a list, so that the pullback property can be proved.

## 4 Promotions and upgrades: lifting ornaments for function types

Ornaments give rise to constructions for promoting an element of a basic datatype to a structurally similar element of a more informative datatype. Having such constructions only for datatypes, however, is not enough — we need to promote operations on these datatypes too. Lifting of ornaments for function types has been considered by Ko & Gibbons (2013a) and more systematically developed by Dagand & McBride (2014). Here, we propose another systematic lifting mechanism, which is less complicated and more easily extensible than Dagand and McBride's approach.

### 4.1 Promotions: axiomatising the ornamental relationship

Since the ornamental constructions can be summarised as the promotion isomorphisms — for instance, OrdList $b \cong \Sigma$ (List $Val$) (Ordered $b$) — it is natural to axiomatise promotability as the existence of such isomorphisms. It turns out that we require only a part of the inverse properties to make the function-promoting mechanism work. Hence, we dismantle the promotion isomorphism and include only the necessary inverse property in the axiomatisation:

```
record Promotion (X Y : Set) : Set₁ where
  field
    Predicate  : X → Set
    forget     : Y → X
    complement : (y : Y) → Predicate (forget y)
    promote    : (x : X) → Predicate x → Y
    coherence  : (x : X) (p : Predicate x) → forget (promote x p) ≡ x
```

In prose, when we say that $X$ : Set can be promoted to $Y$ : Set, we mean that there is a *promotion predicate* $P : X \rightarrow$ Set such that $Y$ and $\Sigma\ X\ P$ are interconvertible by $forget \vartriangle complement : Y \rightarrow \Sigma\ X\ P$ (where $\_\vartriangle\_$ is defined by $(f \vartriangle g)\ x = (f\ x\ ,\ g\ x)$) and $uncurry\ promote : \Sigma\ X\ P \rightarrow Y$. We might think of every element $y : Y$ as consisting of a "core" element $forget\ y : X$ extended with some additional information of type $P\ (forget\ y)$. Consequently, when we construct an element $y : Y$ by promoting $x : X$, we expect that its core, $forget\ y$, is exactly $x$ — this is the *coherence* property that we impose on *promote* and *forget*. From every ornamental promotion isomorphism, we can construct a promotion for which *Predicate* is the optimised promotion predicate, *forget* is the forgetful function, and the remaining three components are drawn from the appropriate parts of the isomorphism. For example, the promotion induced by the ornament from lists to ordered lists with lower bound $b$ has type Promotion (List $Val$) (OrdList $b$); it uses Ordered $b$ as its *Predicate* component and the forgetful function of the ornament as its *forget* component.

Whilst promotions are defined to be an abstraction of the ornamental relationship, we do not strive to strengthen the definition so as to rule out other unintended instantiations. What matters to us is that the function-promoting mechanism works as expected when the basic promotions used are induced by ornaments. To see the generality of promotions, note that promotions can be seen as a variant of *bidirectional transformations*, a subject that has been extensively explored. (See, e.g., Czarnecki *et al.* (2009) for a comprehensive survey.) In a promotion from $X$ to $Y$, $Y$ is the source type and $X$ is the view type, with *forget* and *promote* being respectively the forward (*get*) and backward (*put*) transformations. The promotion predicate corresponds to the idea of *view complement*, which represents the information that is present in the source but not in the view. (In fact, we chose the name *complement* with this analogy in mind.) Finally, the *coherence* property is exactly the *PutGet* property, one of the two inverse-like properties that constitute *well-behavedness* of bidirectional transformations.

### *4.2 Towards promotions for function types*

Let us first explain the idea of function promotion with a small example. Since there is an ornament from natural numbers to lists, we can induce a promotion $p$ : Promotion Nat (List $A$) (for some fixed $A$ : Set), which we will use throughout this example. The promotion predicate of $p$ is Vec $A$: To promote a natural number $n$ to a list, we need to supply $n$ elements, i.e., a vector of type Vec $A$ $n$. With respect to this promotion $p$, we can promote the following function:

$$\begin{aligned}
&\mathit{double} &&: \mathsf{Nat} \to \mathsf{Nat}\\
&\mathit{double}\ \mathsf{zero} &&= \mathsf{zero}\\
&\mathit{double}\ (\mathsf{suc}\ n) &&= \mathsf{suc}\ (\mathsf{suc}\ (\mathit{double}\ n))
\end{aligned}$$

to a “similarly behaving” function $g$ : List $A$ → List $A$. By “similarly behaving”, we mean that $g$ and *double* map similar arguments to similar results. Since, as specified by $p$, we regard natural numbers and lists as similar when they have the same recursive structure, $g$ should process the recursive structure (i.e., nil and cons nodes) of its input in the same way as *double* , i.e.,

$$(xs : \mathsf{List}\ A) \to \mathit{double}\ (\mathit{length}\ xs) \equiv \mathit{length}\ (g\ xs)$$

or as a commutative diagram:

$$\begin{tikzcd}
\mathsf{List}\ A \arrow[r, "g"] \arrow[d, "\mathit{length}"'] & \mathsf{List}\ A \arrow[d, "\mathit{length}"] \\
\mathsf{Nat} \arrow[r, "\mathit{double}"'] & \mathsf{Nat}
\end{tikzcd}$$

Dagand & McBride (2014) called this the *coherence property*. (Note that it is stated in terms of *length*, i.e., the *forget* function of $p$.)

By analogy with the data promotion mechanism offered by ornaments, rather than explicitly writing a coherent function $g$, we can instead show that *double* satisfies a promotion predicate. We already know that $g$ should produce lists whose lengths are computed by *double*; the missing information for constructing $g$, then, is how $g$ processes the list elements whilst respecting the length constraint. The natural promotion predicate to use in this case is thus

$$\lambda f \mapsto (n : \mathsf{Nat}) \to \mathsf{Vec}\ A\ n \to \mathsf{Vec}\ A\ (f\ n)$$

(Note, again, that this is defined in terms of the promotion predicate of $p$, i.e., $\mathsf{Vec}\ A$.) Given a promotion proof for *double*, say

$$\begin{array}{lll} \mathit{duplicate}' & : (n : \mathsf{Nat}) \to \mathsf{Vec}\ A\ n \to \mathsf{Vec}\ A\ (\mathit{double}\ n) & \\ \mathit{duplicate}'\ \mathsf{zero} & [] & = [] \\ \mathit{duplicate}'\ (\mathsf{suc}\ n)\ (x :: \mathit{xs}) & & = x :: x :: \mathit{duplicate}'\ n\ \mathit{xs} \end{array}$$

we can synthesise a function $\mathit{duplicate} : \mathsf{List}\ A \to \mathsf{List}\ A$ by

$$\begin{array}{l} \mathit{duplicate}\ : \mathsf{List}\ A \to \mathsf{List}\ A \\ \mathit{duplicate}\ = \mathit{uncurry}\ \mathit{promote} \circ (\mathit{double} * \mathit{duplicate}'\ \_) \circ (\mathit{forget} \bigtriangleup \mathit{complement}) \\ \quad \textbf{where open}\ \mathsf{Promotion}\ p \end{array}$$

(where $(f * g)\ (x\ ,\ y) = (f\ x\ ,\ g\ y)$). That is, we decompose the input list into the underlying natural number and a vector of elements, process the two parts separately with *double* and *duplicate*′, and finally combine the results back to a list.

The sketch above suggests that we can construct promotions on function types from smaller promotions. The actual story has a few twists, though, as we will show below.

#### 4.2.1 *A failed attempt*

A first attempt might be to write a promotion combinator of type

$$\mathsf{Promotion}\ X\ Y \to \mathsf{Promotion}\ Z\ W \to \mathsf{Promotion}\ (X \to Z)\ (Y \to W)$$

We can argue that this combinator does not make sense, however: Consider the ornament-induced promotion from $\mathsf{Nat}$ to $\mathsf{List\ Nat}$. Having the above combinator would mean that we can construct a promotion from $\mathsf{Nat} \to \mathsf{Nat}$ to $\mathsf{List\ Nat} \to \mathsf{List\ Nat}$. We have seen in the *double*–*duplicate* example that the natural choice of the promotion predicate is

$$\lambda f \mapsto (n : \mathsf{Nat}) \to \mathsf{Vec\ Nat}\ n \to \mathsf{Vec\ Nat}\ (f\ n)$$

The problem arises when we try to define *forget* and *complement*. Being able to define these two functions means that every function of type $\mathsf{List\ Nat} \to \mathsf{List\ Nat}$ can be decomposed into an underlying function of type $\mathsf{Nat} \to \mathsf{Nat}$ acting exclusively on the recursive structure and a complement function on the list elements. This is clearly not the case, though, as we can easily find functions which manipulate the recursive structure in a way that depends on the list elements — for example, we can sum the elements of the input list and then produce a list of zeros whose length is the sum (and does not depend on the length of the input list at all).

We should note that the above argument is not rigorous, since it does not really show that a combinator of the given type cannot exist — we could have chosen other promotion predicates and/or constructed "strange" yet passable *forget* and *complement* functions. (Note that there is no law in the definition of promotions that puts restrictions on *complement*.) The argument merely shows that a combinator of the type does not match well with our intention. Also note that taking variance into account (changing the type of the first argument to the combinator to Promotion $Y\ X$) does not help: In the case of *double* and *duplicate*, for example, we want to uniformly promote Nat to List Nat in the type of *double*, regardless of variance.

### 4.2.2 Promoting to evidently coherent functions only

The attempt in Section 4.2.1 fails because not every function of type List Nat → List Nat behaves like a function of type Nat → Nat. One way out is to restrict the promotion to only those functions on lists that do behave like functions on natural numbers. In general, we could aim for an alternative combinator of the following type:

$$\begin{aligned}&(p : \mathsf{Promotion}\ X\ Y)\,(q : \mathsf{Promotion}\ Z\ W) \to\\&\mathsf{Promotion}\ (X \to Z)\,(\Sigma[f \in Y \to W]\ \Sigma[g \in X \to Z]\ \mathit{FCoherent}\ p\ q\ f\ g)\end{aligned}$$

where

$$\begin{aligned}&\mathit{FCoherent} : \mathsf{Promotion}\ X\ Y \to \mathsf{Promotion}\ Z\ W \to\\&\qquad\qquad (Y \to W) \to (X \to Z) \to \mathsf{Set}\\&\mathit{FCoherent}\ p\ q\ f\ g = (y : Y) \to \mathsf{Promotion}.\mathit{forget}\\&q\ (f\ y) \equiv g\ (\mathsf{Promotion}.\mathit{forget}\ p\ y)\end{aligned}$$

That is, we promote $X \to Z$ to only the subset of $Y \to W$ that are evidently coherent with some function of type $X \to Z$. This combinator is easy to define, and works reasonably well in simple cases: The *promote* function now produces a triple consisting of the promoted function, the input core function, and a coherence proof for the two functions. After a promotion, we do a projection to get either the promoted function or the coherence proof.

This combinator does not work so well for curried functions, however. For example, composing three promotions respectively from basic types $X_0$, $X_1$, and $X_2$ to more informative types $Y_0$, $Y_1$, and $Y_2$ using the combinator in the right order, we get a promotion from $X_0 \to X_1 \to X_2$ to

$$\begin{aligned}&\Sigma[f \in Y_0 \to \Sigma[f' \in Y_1 \to Y_2]\ \Sigma[g' \in X_1 \to X_2]\ \mathit{FCoherent}\ \_\ \_\ f'\ g']\\&\quad \Sigma[g \in X_0 \to X_1 \to X_2]\ \mathit{FCoherent}\ \_\ \_\ f\ g\end{aligned}$$

with which it is inconvenient to extract even just the promoted functions, let alone the coherence proofs.

In general, this trick of "promoting to an evidently coherent subset" could be used to develop a variety of promotion combinators, but these combinators do not work well when they are used to construct promotions between more complex types. For a different example, sometimes we want to promote $X$ to $(i : I) \to Y\ i$ provided that $X$ can be promoted to $Y\ i$ for every $i : I$. This is useful when, for example, promoting List $A$ → List $A$ to $(n : \mathsf{Nat}) \to \mathsf{Vec}\ A\ n \to \mathsf{Vec}\ A\ n$, where the promoted function type has an additional argument $n$ for indexing the vectors. Again, there is no forgetful function from $(i : I) \rightarrow Y\ i$ to $X$, and we must restrict the promoted type to $\Sigma[f \in (i : I) \rightarrow Y\ i]\ \Sigma[x \in X]\ ((i : I) \rightarrow \mathit{forget}\ (f\ i) \equiv x)$. This combinator, like the one for function types above, can greatly complicate the extraction of promoted functions and coherence proofs. We will develop a better solution next, in which this trick will still play a role though.

### *4.3 Upgrades: lifting promotions for function types*

(Part of) the problem with promotions for function types in Section 4.2.1 was that we cannot define a sensible *forget*. In typical use cases, however, after we *promote* a basic function, there is no need to apply *forget* to the promoted function to retrieve the basic function — we already have the basic function in the first place. This suggests that we use a slightly different formulation of promotion for function types, which does not support *forget*:

```
record Upgrade (X Y : Set) : Set₁ where
  field
    Predicate  : X → Set
    Coherence  : X → Y → Set
    complement : (x : X) (y : Y) → Coherence x y → Predicate x
    promote    : (x : X) → Predicate x → Y
    coherence  : (x : X) (p : Predicate x) → Coherence x (promote x p)
```

An upgrade does not require the existence of a *forget* function, but includes a *Coherence* property which is used to reformulate the types of *complement* and *coherence*. The relationship between promotions and upgrades is most clearly shown by the following combinator *toUpgrade*, which says that promotions are special cases of upgrades, namely those whose *Coherence* property is stated in terms of *forget*:

```
toUpgrade : {X Y : Set} → Promotion X Y → Upgrade X Y
toUpgrade p = record
  { Predicate  = Promotion.Predicate p
  ; Coherence  = λ x y ↦ Promotion.forget p y ≡ x
  ; complement = λ {._ y refl ↦ Promotion.complement p y }
  ; promote    = Promotion.promote p
  ; coherence  = Promotion.coherence p }
```

We can then formulate in general terms the promoting construction in the *double–duplicate* example:

```
_⇀_ : {X Y Z W : Set} →
        Promotion X Y → Upgrade Z W → Upgrade (X → Z) (Y → W)
_⇀_ {X} {Y} p u = record
        {Predicate   =
          λ f ↦ (x : X) → Promotion.Predicate p x → Upgrade.Predicate u (f x)
        ; Coherence  =
          λ f g ↦ (y : Y) → Upgrade.Coherence u (f (Promotion.forget p y)) (g y)
```

```
    ; complement  =  ...
    ; promote     =
       λ f com ↦ uncurry (Upgrade.promote u) ∘ (f ∗ com _) ∘
                        (Promotion.forget p △ Promotion.complement p)
    ; coherence   =  ... }
```

In particular, *promote* does exactly what we did in the *double–duplicate* example: Given $f : X \to Z$ and its complement function *com*, we construct a function of type $Y \to W$ by separating the argument of type $Y$ into its core of type $X$ and a complement, processing them respectively using $f$ and *com*, and invoking *promote* of the upgrade $u$ to combine the processed pair. The *double–duplicate* example can now be handled by the upgrade:

```
Nat-List A ⇀ toUpgrade (Nat-List A) : Upgrade (Nat → Nat) (List A → List A)
```

where *Nat-List* : ($A$ : Set) → Promotion Nat (List $A$) encapsulates the promotion isomorphism induced by the ornament from natural numbers to lists. Note that the computed coherence property looks exactly as we want it to, and that computation works nicely for curried functions too, solving one problem mentioned in Section 4.2.2.

The promotion-upgrade mechanism is easily extensible, since the combinators are shallowly embedded (cf. Dagand & McBride's (2014) approach, which uses deep embedding). For example, the other problem mentioned in Section 4.2.2 is additional arguments in promoted function types. We can extend the mechanism to deal with additional arguments by writing a new combinator:

```
new : (I : Set) {X : Set} {Y : I → Set}
      (u : (i : I) → Upgrade X (Y i)) → Upgrade X ((i : I) → Y i)
new I u = record
  { Predicate   = λ x ↦ (i : I) → Upgrade.Predicate (u i) x
  ; Coherence   = λ x y ↦ (i : I) → Upgrade.Coherence (u i) x (y i)
  ; complement  = λ x y coh i ↦ Upgrade.complement (u i) x (y i) (coh i)
  ; promote     = λ x com i ↦ Upgrade.promote (u i) x (com i)
  ; coherence   = λ x com i ↦ Upgrade.coherence (u i) x (com i) }
syntax new I (λ i ↦ u) = ∀⁺[ i ∈ I ] u
```

The promotion mentioned in Section 4.2.2 can then be handled by the upgrade:

```
∀⁺[ n ∈ Nat ] (List-Vec A n ⇀ toUpgrade (List-Vec A n)) :
   Upgrade (List A → List A) ((n : Nat) → Vec A n → Vec A n)
```

where *List-Vec* : ($A$ : Set) ($n$ : Nat) → Promotion (List $A$) (Vec $A$ $n$) is induced by the ornament from lists to vectors. The upgrade itself looks like a dependent function type, thanks to the mixfix binder $\forall^+[\,n \in \mathsf{Nat}\,]$.

**Higher-order functions.** It might seem that we lose the ability to promote higher-order functions, as _⇀_ requires a promotion as its first argument, which does not seem to be able to relate function types. But in fact, we can deal with higher-order functions surprisingly straightforwardly, by injecting upgrades back into promotions:

```
fromUpgrade : {X Y : Set} (u : Upgrade X Y) →
                Promotion X (Σ[ y ∈ Y ] Σ[ x ∈ X ] Upgrade.Coherence u x y)
fromUpgrade u  = record
  { Predicate  = Upgrade.Predicate u
  ; forget     = proj₁ ∘ proj₂
  ; complement = λ { (y , x , coh) ↦ Upgrade.complement u x y coh }
  ; promote    = λ x com ↦ Upgrade.promote u x com , x ,
                           Upgrade.coherence u x com
  ; coherence  = λ _ _ ↦ refl }
```

This is a general formulation of the trick of “promoting to evidently coherent subsets” used in Section 4.2.2.

Here, we give a non-trivial but not perfect example of promoting a higher-order function. Consider a version of *concatMap* specialised to lists whose elements are themselves lists:

```
concatMap   : (List A  →  List B)  →  List (List A)  →  List B
concatMap f  =  concat ∘ map f
```

If we know that the function $f$ preserves a particular length, and that all lists in the input list have that length, then we can determine the length of the output list in advance. Formally:

```
(n  : Nat) (f  : List A → List B) →
            ((xs  : List A) → length xs ≡ n → length (f xs) ≡ n) →
(m  : Nat) (xss  : List (List A)) →
            length xss ≡ m  ×  All (λ xs → length xs ≡ n) xss →
  length (concatMap f xss) ≡ m ∗ n
```

This is exactly the type of promotion proof computed by the upgrade

```
∀⁺[ n ∈ Nat ] (fromUpgrade (List-Vec A n ⇀ List-Vec A n) ⇀
                ∀⁺[ m ∈ Nat ] (List²-Vec² A m n ⇀ toUpgrade (List-Vec B (m ∗ n))) :
  Upgrade ((List A  →  List B)  →  List (List A)  →  List B)
          ((n  : Nat)  →  (Σ[ g ∈ Vec A n → Vec A n ] Σ[ f ∈ List A → List A ]
                              ((xs  : Vec A n)  →  toList (g xs) ≡ f (toList xs)))  →
           (m  : Nat)  →  Vec (Vec A n) m  →  Vec B (m ∗ n))
```

whose type shows the promoted function type, which works on vectors. (We omit the definition of $\mathit{List}^2\text{-}\mathit{Vec}^2 \; : \; (A \; : \; \mathsf{Set}) \; (m \; n \; : \; \mathsf{Nat}) \; \to \; \mathsf{Promotion} \; (\mathsf{List} \; (\mathsf{List} \; A)) \; (\mathsf{Vec} \; (\mathsf{Vec} \; A \; n) \; m)$, which can be constructed either in an *ad hoc* way or somehow compositionally.) A promoted function, say *concatMapV*, will satisfy the coherence property:

```
(n  : Nat) (g  : Vec A n → Vec B n) (f  : List A → List B)
            (c  : (xs  : Vec A n) → toList (g xs) ≡ f (toList xs)) →
(m  : Nat) (xss  : Vec (Vec A n) m) →
  toList (concatMapV (g , f , c) xss) ≡ concatMap f (map toList (toList xss))
```

which is also computed, and proved, by the upgrade.

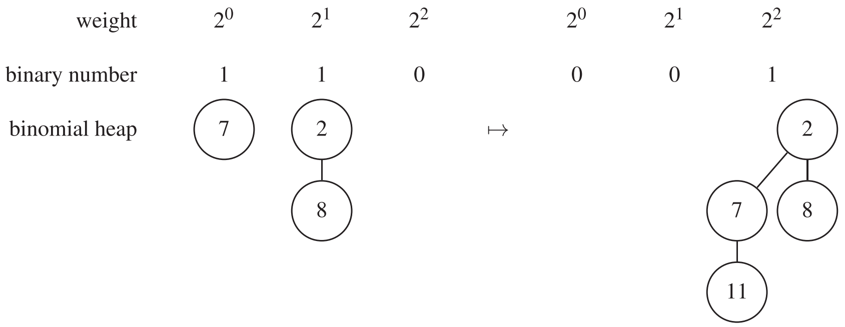

Fig. 3. *Left:* A binomial heap of size 3 consisting of two binomial trees storing elements 2, 5, and 8. *Right:* The result of inserting 11 into the heap. (Note that the digits of the underlying binary numbers are ordered with the least significant digit first.)

The $(g\ ,f\ ,c)$-triple argument to *concatMapV* looks rather bulky — why can it not just be $g$? This has to do with the semantic way in which we promote functions: Note that a function of type $\mathsf{Vec}\ A\ n\ \rightarrow\ \mathsf{Vec}\ A\ n$ (which should be distinguished from $\{n\ :\ \mathsf{Nat}\}\ \rightarrow\ \mathsf{Vec}\ A\ n\ \rightarrow\ \mathsf{Vec}\ A\ n$) is less general than a function of type $\mathsf{List}\ A\ \rightarrow\ \mathsf{List}\ A$. Since *concatMapV* computes by calling *concatMap*, it must somehow supply an extension of $g$ — i.e., a function $f\ :\ \mathsf{List}\ A\ \rightarrow\ \mathsf{List}\ A$ such that $\mathit{toList}\ (g\ \mathit{xs}) \equiv f\ (\mathit{toList}\ \mathit{xs})$ for all $\mathit{xs}\ :\ \mathsf{Vec}\ A\ n$ — to *concatMap*; here *concatMapV* simply requires such an extension as an additional argument. (This might not be as problematic as it seems: If we consistently use function promotion, that is, functional arguments to promoted functions are themselves promoted functions, then we can simply bundle the promoted functions with the basic functions and coherence proofs and pass the triples around.)

## 5 Case study: binomial heaps

Having gone through the previous sections on index-first inductive families, ornaments, and function promotion, we are now ready for our case study on implementing binomial heaps and their operations by relating them to binary numbers and numeric operations.

Every schoolchild is familiar with the idea of *positional number systems*, in which a number is represented as a list of digits. Each digit is associated with a weight, and the interpretation of the list is the weighted sum of the digits. For example, the weights used for binary numbers are powers of 2. Some container data structures and associated operations strongly resemble positional representations of natural numbers and associated operations. For example, a *binomial heap* (illustrated in Figure 3) can be thought of as a binary number in which every 1-digit stores a *binomial tree* — the actual vehicle for storing elements — whose size is exactly the weight of that digit. The number of elements stored in a binomial heap is therefore exactly the value of the underlying binary number. Inserting a new element into a binomial heap is analogous to incrementing a binary number,

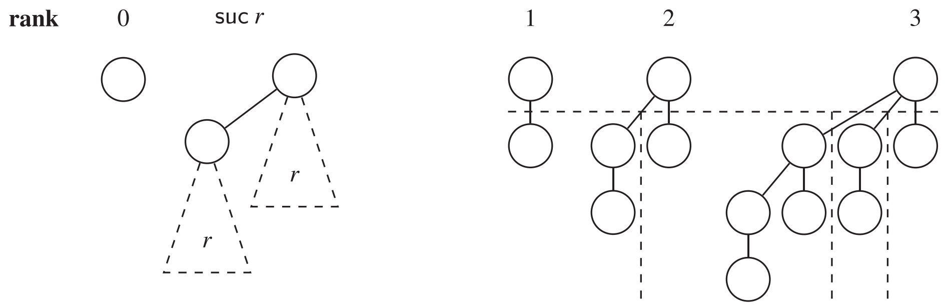

Fig. 4. *Left:* Inductive definition of binomial trees. *Right:* Decomposition of binomial trees of ranks 1 to 3.

with carries corresponding to combining smaller binomial trees into larger ones. Okasaki (1999) thus proposed to design container data structures by analogy with positional representations of natural numbers, and called such data structures *numerical representations*. Using an ornament, it is easy to express the relationship between a numerically represented container datatype (e.g., binomial heaps) and its underlying numeric datatype (e.g., binary numbers). But the ability to express the relationship alone is not too surprising. What is more interesting is that the ornament can give rise to upgrades such that

- the coherence properties of the upgrades semantically characterise the resemblance between container operations and corresponding numeric operations, and
- the promotion predicates give the precise types of the container operations that guarantee such resemblance.

In this section, we show how ornaments induce insertion and extraction operations on binomial heaps, given increment and decrement operations on binary numbers. The induced extraction operation is not quite what is usually wanted, though, since the element it extracts is not necessarily the minimum element. We also show how to build a derived operation to extract the minimum element, by identifying a more refined datatype of binomial heaps, and we show how ornaments induce the conversions to and from this more refined datatype.

## *5.1 Binomial trees*

The basic building blocks of binomial heaps are *binomial trees*, in which elements are stored. We assume that elements are drawn from a set *Val* on which there is a decidable ordering

$$\_\leqslant\_ \ : \ \mathit{Val} \ \rightarrow \ \mathit{Val} \ \rightarrow \ \mathsf{Set}$$

with comparison function

$$\_\leqslant_?\_ \ : \ (x \ y \ : \ \mathit{Val}) \ \rightarrow \ (x \leqslant y) \uplus (y \leqslant x)$$

Binomial trees are defined inductively on their *rank*, which is a natural number (see Figure 4):

- A binomial tree of rank 0 is a single node storing an element of type *Val*, and
- a binomial tree of rank suc $r$ consists of two binomial trees of rank $r$, with one attached under the other's root node.

From this definition, we can readily deduce that a binomial tree of rank $r$ has $2^r$ elements. To actually define binomial trees as a datatype, however, an alternative view is more helpful: A binomial tree of rank $r$ is a root node and a forest of binomial trees of ranks 0 to $r - 1$. (Figure 4 shows how binomial trees of ranks 1 to 3 can be decomposed according to this view.) We could thus define a datatype BTree : Nat → Set of binomial trees — which is indexed with the rank — as follows: For any rank $r$ : Nat, the type BTree $r$ has a field of type *Val*, namely the root node, and $r$ recursive positions indexed from $r - 1$ down to 0. For our experiment, it is not necessary to encode the BTree datatype as a description, since we do not need any ornamental constructions involving BTree; but BTree turns out to be an interesting example, so we present the encoding anyway:

```
BTreeD : Desc Nat
BTreeD r = σ[ _ ∈ Val ] v (descend r)
indexfirst data BTree : Nat → Set where
  BTree r ∋ node Val (ℙ (descend r) BTree)
```

This is an interesting example since we are exploiting the full computational power of Desc, computing the list of recursive indices from the index request $r$ by *descend* $r$, which yields a list from $r - 1$ down to 0:

```
descend : Nat → List Nat
descend zero    = []
descend (suc n) = n :: descend n
```

For use in min-heaps, however, we should also ensure that elements in binomial trees are in *heap order*, i.e., the root of any binomial tree (including subtrees) is the minimum element in the tree. As demonstrated earlier with ordered lists (Section 3.2), ordering invariants can be easily encoded with inductive families. Here, we go for a slightly more space-efficient variant: Instead of a lower bound, we index the BTree datatype by a given root stored elsewhere. A more appropriate name for the revised datatype is thus BForest, since an inhabitant of the datatype now defines only a *forest* of binomial trees that are considered attached under the root specified as the index (which is not stored in the inhabitant itself):

```
indexfirst data BForest : Nat → Val → Set where
  BForest r x ∋ forest (ℙ (descend r) (λ i → Σᴵ [ t ∈ Σ Val (BForest i) ]
                                                x ≤ proj₁ t))
```

($\Sigma^{\mathrm{I}}$ is a special dependent pair type former, which will be introduced below.) A binomial tree, then, is just a pair of its root and the forest of subtrees. We thus redefine BTree as a dependent pair type:

```
BTree : Nat → Set
BTree r = Σ Val (BForest r)
```

For clarity, we give alternative names to the two projection functions:

```
root     : {r : Nat} → BTree r → Val
root     = proj₁
children : {r : Nat} (t : BTree r) → BForest r (root t)
children = proj₂
```

As an aside, the encoding of the BForest datatype as a description requires a bit of twist, but can still be done fairly easily. We can even define BForest as an ornamentation of the BTree datatype from the previous paragraph, but (again) this is not essential since we will exclusively use the BForest and BTree datatypes defined in this paragraph from now on, making no use of the relationship between BForest and the first version of BTree.

The most important operation on binomial trees is combining two smaller binomial trees of the same rank into a larger one, corresponding to carries in positional arithmetic. Given two binomial trees of the same rank $r$, one can be *attach*ed under the root of the other, forming a single binomial tree of rank suc $r$, corresponding exactly to the inductive definition of binomial trees. To establish heap ordering, we should also require a proof that that the root of the first tree is no less than the root of the second tree, and put this proof into the new tree. Here, we adopt (a variation of) the technique (McBride, 2014) of using AGDA's *instance arguments* (Devriese & Piessens, 2011) to pass around such proofs implicitly. We define a special dependent pair type whose second component can be suppressed and automatically taken care of by AGDA's instance searching mechanism:

```
record Σᴵ (A : Set) (B : A → Set) : Set where
  constructor ⟨_⟩
  field
    fst   : A
    {{snd}} : B fst
syntax Σᴵ A (λ x → B) = Σᴵ [x ∈ A] B
```

Typically, we deconstruct a $\Sigma^{\mathrm{I}}$-typed element by matching it against the constructor ⟨_⟩ without naming (and thus explicitly using) its second component, which will nevertheless be considered as an available instance during instance searching. On the other hand, when we construct a new $\Sigma^{\mathrm{I}}$-typed element using the constructor ⟨_⟩, supplying only its first component, its second component will be automatically determined to be the unique available instance having the right type. (Typechecking fails unless there is exactly one available instance having the right type.) This special dependent pair type was used to define BForest, and allows ordering proofs in elements of BForest to be handled implicitly by instance searching. We can now define *attach*, which takes two trees of the same rank and a proof of ordering of their roots, by

```
attach : {r : Nat} (t u : BTree r) {{_ : root u ≤ root t}} → BTree (suc r)
attach t (y , forest us) = y , forest (⟨ t ⟩ , us)
```

Using *attach* requires that we know how the roots of two trees are ordered in advance. To write a more general combining function, we first define the following proof-relevant version of **if_then_else_**:

$\textbf{if\_then\_else\_} : \{A\ B\ C : \mathsf{Set}\} \to A \uplus B \to (\{\!\{\_ : A\}\!\} \to C) \to (\{\!\{\_ : B\}\!\} \to C) \to C$
$\textbf{if}\ \mathsf{inj}_1\ a\ \textbf{then}\ t\ \textbf{else}\ u \;=\; t\ \{\!\{a\}\!\}$
$\textbf{if}\ \mathsf{inj}_2\ b\ \textbf{then}\ t\ \textbf{else}\ u \;=\; u\ \{\!\{b\}\!\}$

and then we can compare the roots of two binomial trees to decide which will be attached under the other:

$\mathit{link} : \{r : \mathsf{Nat}\} \to \mathsf{BTree}\ r \to \mathsf{BTree}\ r \to \mathsf{BTree}\ (\mathsf{suc}\ r)$
$\mathit{link}\ t\ u \;=\; \textbf{if}\ \mathit{root}\ t \leqslant_?\ \mathit{root}\ u\ \textbf{then}\ \mathit{attach}\ u\ t\ \textbf{else}\ \mathit{attach}\ t\ u$

Instance searching manages all ordering proofs behind the scenes, and makes it impossible to get the two branches the wrong way around, as remarked by McBride (2014) for his development.

### *5.2 From binary numbers to binomial heaps*

The datatype Bin : Set of binary numbers is just a specialised datatype of lists of binary digits:

```
indexfirst data Bin : Set where
  Bin ∋ nil
      | zero Bin
      | one  Bin
```

This datatype can be encoded as follows:

```
data BinTag : Set where
  ‘nil  : BinTag
  ‘zero : BinTag
  ‘one  : BinTag
BinD : Desc ⊤
BinD tt = σ BinTag λ { ‘nil  ↦ v []
                     ; ‘zero ↦ v (tt :: [])
                     ; ‘one  ↦ v (tt :: []) }
```

The intended interpretation of binary numbers is given by

$\mathit{toNat} : \mathsf{Bin} \to \mathsf{Nat}$
$\mathit{toNat}\ \mathsf{nil} = 0$
$\mathit{toNat}\ (\mathsf{zero}\ b) = 0 + 2 \times \mathit{toNat}\ b$
$\mathit{toNat}\ (\mathsf{one}\ b) = 1 + 2 \times \mathit{toNat}\ b$

That is, binary numbers of type Bin are written least significant bit first, and the $i$th digit (counting from zero) has weight $2^i$. We refer to the position of a digit as its *rank*, i.e., the $i$th digit is said to have rank $i$.

As stated in the opening of Section 5, binomial heaps are binary numbers whose 1-digits are decorated with binomial trees of matching rank, and they can be expressed straightforwardly as an ornamentation of binary numbers. To ensure that the binomial trees in binomial heaps have the correct rank, the datatype BHeap : Nat → Set is indexed with the starting rank: If a binomial heap of type BHeap *r* is non-empty (i.e., not nil), then its first digit has rank *r* (and stores a binomial tree of rank *r* when the digit is one), and the rest of the heap is indexed with suc *r*:

```
indexfirst data BHeap : Nat → Set where
  BHeap r ∋ nil
          | zero (BHeap (suc r))
          | one  (BTree r) (BHeap (suc r))
```

This datatype can be encoded as

```
BHeapOD : OrnDesc Nat ! BinD
BHeapOD (ok r) = σ BinTag λ { ‘nil  ↦ v tt
                            ; ‘zero ↦ v (ok (suc r) , tt)
                            ; ‘one  ↦ Δ[ t ∈ BTree r ] v (ok (suc r) , tt) }
```

In applications, we would use binomial heaps of type BHeap zero, since this type encompasses binomial heaps of all sizes.

### 5.3 Increment and insertion

The increment operation on binary numbers is defined by

```
incr : Bin → Bin
incr nil      = one nil
incr (zero b) = one b
incr (one  b) = zero (incr b)
```

The corresponding operation on binomial heaps is insertion of a binomial tree into a binomial heap (of matching rank), whose direct implementation is

```
-- Later this function will be synthesised using function promotion
insT : {r : Nat} → BTree r → BHeap r → BHeap r
insT t nil        = one t nil
insT t (zero   h) = one t h
insT t (one u  h) = zero (insT (link t u) h)
```

Conceptually, *incr* puts a 1-digit into the least significant position of a binary number, triggering a series of carries, i.e., summing 1-digits of smaller ranks into 1-digits of larger ranks; *insT* follows the pattern of *incr*, but since 1-digits now have to store a binomial tree of matching rank, *insT* takes an additional binomial tree as input and *link*s binomial trees of smaller ranks into binomial trees of larger ranks whenever carries occur. Having defined *insT*, inserting a single element into a binomial heap of type BHeap zero is then inserting, by *insT*, a rank-0 binomial tree (i.e., a single node) storing the element into the heap.

$$\begin{aligned}&\mathit{insV} : \mathit{Val} \to \mathsf{BHeap\ zero} \to \mathsf{BHeap\ zero}\\ &\mathit{insV}\ x = \mathit{insT}\ (x\ ,\ \mathsf{forest\ tt})\end{aligned}$$

It is apparent that the program structure of *insT* strongly resembles that of *incr* — they manipulate the list-of-binary-digits structure in the same way. But can we characterise the resemblance semantically? It turns out that the coherence property of the following upgrade from the type of *incr* to that of *insT* is an appropriate answer:

$$\begin{aligned}&\mathit{upg} : \mathsf{Upgrade}\ (\qquad\qquad\qquad\qquad\qquad\ \ \mathsf{Bin} \to \mathsf{Bin}\qquad)\\ &\qquad\qquad\qquad(\{r : \mathsf{Nat}\} \to \mathsf{BTree}\ r \to \mathsf{BHeap}\ r \to \mathsf{BHeap}\ r)\\ &\mathit{upg} = \forall^{+}[[r \in \mathsf{Nat}]]\ \forall^{+}[\_ \in \mathsf{BTree}\ r]\ \mathit{Bin\text{-}BHeap}\ r \rightharpoonup \mathit{toUpgrade}\ (\mathit{Bin\text{-}BHeap}\ r)\end{aligned}$$

The upgrade *upg* says that, compared to the type of *incr*, the type of *insT* has two new arguments — the implicit argument $r$ : Nat (declared using $\forall^{+}[[r \in \mathsf{Nat}]]$, which is the same as $\forall^{+}[r \in \mathsf{Nat}]$ except that the new argument in the upgraded function type is implicit) and the explicit argument of type BTree $r$ — and that the two occurrences of BHeap $r$ in the type of *insT* refine the corresponding occurrences of Bin in the type of *incr* using the promotion

$$\begin{aligned}&\text{-- Induced from the ornament } \lceil \mathit{BHeapOD} \rceil\\ &\mathit{Bin\text{-}BHeap} : (r : \mathsf{Nat}) \to \mathsf{Promotion\ Bin}\ (\mathsf{BHeap}\ r)\end{aligned}$$

induced by the ornament $\lceil \mathit{BHeapOD} \rceil$ (ok $r$) from Bin to BHeap $r$. The upgrade computes a coherence property, which instantiates for *incr* and *insT* to

$$\begin{aligned}&\text{-- Computed by Upgrade.}\mathit{Coherence\ upg\ incr\ insT}\\ &\{r : \mathsf{Nat}\}\ (t : \mathsf{BTree}\ r)\ (b : \mathsf{Bin})\ (h : \mathsf{BHeap}\ r) \to\\ &(\mathit{toBin}\ h \equiv b) \to (\mathit{toBin}\ (\mathit{insT}\ t\ h) \equiv \mathit{incr}\ b)\end{aligned}$$

where *toBin* extracts the underlying binary number of a binomial heap:

$$\begin{aligned}&\mathit{toBin} : \{r : \mathsf{Nat}\} \to \mathsf{BHeap}\ r \to \mathsf{Bin}\\ &\mathit{toBin} = \mathit{ornForget}\ \lceil \mathit{BHeapOD} \rceil\end{aligned}$$

That is, given a binomial heap $h$ : BHeap $r$ whose underlying binary number is $b$ : Bin, after inserting a binomial tree into $h$ by *insT*, the underlying binary number of the result is *incr b*. This says exactly that *insT* manipulates the underlying binary number in the same way as *incr*.

We have seen that the coherence property computed by *upg* is appropriate for characterising the resemblance of *incr* and *insT*; proving that it holds for *incr* and *insT* is a separate matter, though. We can, however, avoid doing the implementation of insertion and the coherence proof separately: Instead of implementing *insT* directly, we can implement insertion with a more precise type in the first place such that, from this more precisely typed version, we can derive automatically a function *insT* that satisfies the coherence property. This process is fully supported by the mechanism of upgrades. Specifically, the more precise type for insertion is given by the promotion predicate of *upg* (applied to *incr*), the more precisely typed version of insertion acts as a promotion proof of *incr* (with respect to *upg*), and the promotion gives us *insT*, accompanied by a proof that *insT* is coherent with *incr*.

Let BHeap$^U$ be the optimised promotion predicate for the ornament from Bin to BHeap $r$:

```
indexfirst data BHeap^U : Nat → Bin → Set where
  BHeap^U r nil      ∋ nil
  BHeap^U r (zero b) ∋ zero (h : BHeap^U (suc r) b)
  BHeap^U r (one  b) ∋ one  (t : BTree r) (h : BHeap^U (suc r) b)
BHeap^U : Nat → Bin → Set
BHeap^U r b = OptP ⌈BHeapOD⌉ (ok r) b
```

Here, a more helpful interpretation is that BHeap$^U$ is a datatype of binomial heaps additionally indexed with the underlying binary number. The type of promotion proofs for *incr* then expands to

```
-- Computed by Upgrade.Predicate upg incr
{r : Nat} → BTree r → (b : Bin) → BHeap^U r b → BHeap^U r (incr b)
```

A function of this type is explicitly required to transform the underlying binary number structure of its input in the same way as *incr*. Insertion can now be implemented as

```
insT^U : {r : Nat} → BTree r → (b : Bin) → BHeap^U r b → BHeap^U r (incr b)
insT^U t nil      nil        = one t nil
insT^U t (zero b) (zero   h) = one t h
insT^U t (one  b) (one u  h) = zero (insT^U (link t u) b h)
```

which is very similar to the original *insT*. It is interesting to note that all the constructor choices for binomial heaps in *insT*$^U$ are actually completely determined by the types. This fact is easier to observe if we desugar *insT*$^U$ to its raw representation:

```
insT^U  : {r : Nat} → BTree r → (b : Bin) → BHeap^U r b → BHeap^U r (incr b)
insT^U t (con (‘nil   ,     tt)) (con          tt ) = con (t , con tt                 , tt)
insT^U t (con (‘zero  , b , tt)) (con (    h , tt)) = con (t , h                      , tt)
insT^U t (con (‘one   , b , tt)) (con (u , h , tt)) = con (    insT^U (link t u) b h , tt)
```

in which no constructor tags for binomial heaps are present. This means that the types would determine which constructors to use when programming *insT*$^U$, establishing the coherence property by construction. Indeed, when programming *insT*$^U$ in Agda interactively, most of *insT*$^U$ can be directly generated by Agda (upon instruction by the programmer). Finally, since *insT*$^U$ is a promotion proof for *incr*, we can invoke the promoting operation of *upg* and get *insT*:

```
insT : {r : Nat} → BTree r → BHeap r → BHeap r
insT = Upgrade.promote upg incr insT^U
```

which is automatically coherent with *incr*:

$$\begin{aligned}&\mathit{incr\text{-}insT\text{-}coherence} \;:\; \{r \;:\; \mathsf{Nat}\}\,(t \;:\; \mathsf{BTree}\; r) \to \mathit{toBin}\;(\mathit{insT}\;t\;h) \equiv \mathit{incr}\;(\mathit{toBin}\;h)\\ &\mathit{incr\text{-}insT\text{-}coherence} \;=\; \mathsf{Upgrade}.\mathit{coherence}\;\mathit{upg}\;\mathit{incr}\;\mathit{insT}^U\end{aligned}$$

To sum up: We define Bin, *incr*, and then BHeap as an ornamentation of Bin, describe in *upg* how the type of *insT* is an upgraded version of the type of *incr*, and implement $\mathit{insT}^U$, whose type is supplied by *upg*. We can then derive *insT*, the coherence property of *insT* with respect to *incr*, and its proof, all automatically using *upg*. Compared to Okasaki's implementation (1999), besides rank-indexing, which elegantly transfers the management of rank-related invariants to the type system, the only extra work is the straightforward marking of the differences between Bin and BHeap (in *BHeapOD*) and between the type of *incr* and that of *insT* (in *upg*). The reward is large in comparison: We get a coherence property that precisely characterises the structural behaviour of insertion with respect to increment, and an enriched function type that guides the implementation of insertion such that the coherence property is satisfied by construction.

Similar to lifting increment to insertion, addition on binary numbers can also be lifted to merging of two heaps. The lifting process is the same, though, and is thus omitted from this paper.

### *5.4 Decrement and extraction*

Binomial heaps would be useless if we could only insert but not *extract* elements. Thinking in terms of numerical representations, element extraction on binomial heaps corresponds to *decrement* on binary numbers. Decrement is slightly more complicated than increment since it is inherently a partial operation, applicable to non-zero numbers only. (Defining the result of decrementing zero as zero counts as a workaround at best, as it destroys the natural property that increment is left inverse to decrement.) We will explore the simplest approach, weakening the result type to Maybe Bin:

$$\begin{aligned}&\mathit{decr} \;:\; \mathsf{Bin} \to \mathsf{Maybe\;Bin}\\ &\mathit{decr}\;\mathsf{nil} &&= \mathsf{nothing}\\ &\mathit{decr}\;(\mathsf{zero}\;b) &&= \mathit{mapMaybe}\;\mathsf{one}\;(\mathit{decr}\;b)\\ &\mathit{decr}\;(\mathsf{one}\;\;b) &&= \mathsf{just}\;(\mathsf{zero}\;b)\end{aligned}$$

where $\mathit{mapMaybe} \;:\; \{A\;B \;:\; \mathsf{Set}\} \to (A \to B) \to \mathsf{Maybe}\;A \to \mathsf{Maybe}\;B$ is the usual functorial map for Maybe.

Following the route taken in Section 5.3, we will promote $\mathit{decr} \;:\; \mathsf{Bin} \to \mathsf{Maybe\;Bin}$ to a function:

$$\mathit{extract} \;:\; \{r \;:\; \mathsf{Nat}\} \to \mathsf{BHeap}\;r \to \mathsf{Maybe}\;(\mathsf{BTree}\;r \times \mathsf{BHeap}\;r)$$

which divides a non-empty binomial heap of given rank $r$ into a tree of rank $r$ and a smaller heap of the same rank; if the heap has a rank-$r$ zero digit, then the tree has to be "borrowed" from a higher rank one digit.

The first step is to write an upgrade relating the types of *decr* and *extract*. The language of promotions and upgrades presented in Section 4 cannot deal with the

```
_⁺×_ : (X : Set) {Y Z : Set} → Promotion Y Z → Promotion Y (X × Z)
X ⁺× p = record
  { Predicate  = λ y → X × Promotion.Predicate p y
  ; forget     = Promotion.forget p ∘ proj₂
  ; complement = λ { (x , z) ↦ x , Promotion.complement p z }
  ; promote    = λ { y (x , com) ↦ x , Promotion.promote p y com }
  ; coherence  = λ { y (x , com) ↦ Promotion.coherence p y com }
  }
data Maybe′ {A : Set} (X : A → Set) : Maybe A → Set where
  nothing : Maybe′ X nothing
  just    : {a : A} → X a → Maybe′ X (just a)
MaybeP : {A B : Set} → Promotion A B → Promotion (Maybe A) (Maybe B)
MaybeP p = record
  { Predicate  = Maybe′ (Promotion.Predicate p)
  ; forget     = mapMaybe (Promotion.forget p)
  ; complement = λ { nothing  ↦ nothing
                   ; (just b) ↦ just (Promotion.complement p b) }
  ; promote    = λ { nothing _           ↦ nothing
                   ; (just a) (just x)   ↦ just (Promotion.promote p a x) }
  ; coherence  = λ { nothing _           ↦ refl
                   ; (just a) (just x)   ↦ cong just (Promotion.coherence p a x) } }
```

Fig. 5. Two more promotion combinators for upgrading decrement.

sub-upgrade from Maybe Bin to Maybe (BTree *r* × BHeap *r*), but we can easily extend the language with two combinators:

```
_⁺×_   : (X : Set) {Y Z : Set} → Promotion Y Z → Promotion Y (X × Z)
MaybeP : {A B : Set} → Promotion A B → Promotion (Maybe A) (Maybe B)
```

to support that, as shown in Figure 5. (The most important information in Figure 5 is how the *Predicate* and *forget* fields are defined for the two combinators; the rest of the code is presented simply to show that it is easy to complete the definitions.) These may not be the most generic combinators that we can define, but they show that the framework can be easily extended. The upgrade we write is then

```
upg : Upgrade (                Bin        → Maybe                 Bin          )
               ({r : Nat} → BHeap r → Maybe (BTree r × BHeap r))
upg = ∀⁺[[ r ∈ Nat ]] (Bin-BHeap r ⇀
                            toUpgrade (MaybeP (BTree r ⁺× Bin-BHeap r)))
```

The promotion predicate for *decr* with respect to this upgrade, or rather, a more informative type for extraction, is

```
-- Computed by Upgrade.Predicate upg decr
{r : Nat} {b : Bin} →
BHeapᵁ r b → Maybe′ (λ b′ ↦ BTree r × BHeapᵁ r b′) (decr b)
```

where the result is nothing when *decr* *b* = nothing, or just *p* for some pair *p* : BTree *r* × BHeap$^{U}$ *r* *b′* when *decr* *b* = just *b′*. (See Figure 5 for the definition of Maybe′.) A function of this type is guaranteed to produce a tree and a heap (wrapped in just as a pair) if and only if *decr* executes successfully on the underlying number of the input heap, and when that is the case, the underlying number of the resulting heap will be the result of *decr*.

```
extractᵁ : {r : Nat} (b : Bin) →
             BHeapᵁ r b → Maybe′ (λ b′ ↦ BTree r × BHeapᵁ r b′) (decr b)
extractᵁ nil      nil       = nothing
extractᵁ (zero b) (zero h)  = mapMaybe′ (λ { ((x , t ◁ ts) , h) ↦ (x , forest ts),
                                                                  one t h })
                                          (extractᵁ b h)
extractᵁ (one b)  (one t h) = just (t , zero h)
```

where we introduce a pattern synonym **pattern** _◁_ *t ts* = forest (⟨ *t* ⟩ , *ts*) for separating the leftmost tree from a forest, and *mapMaybe′* is a carefully indexed version of functorial map on Maybe′:

```
mapMaybe′ : {A B : Set} {C : A → Set} {D : B → Set} {f : A → B} →
              ({a : A} → C a → D (f a)) →
              {ma : Maybe A} → Maybe′ C ma → Maybe′ D (mapMaybe f ma)
mapMaybe′ g nothing  = nothing
mapMaybe′ g (just c) = just (g c)
```

We now get a coherent-by-construction definition of *extract* via the upgrade:

```
extract : {r : Nat} → BHeap r → Maybe (BTree r × BHeap r)
extract = Upgrade.promote upg decr extractᵁ
```

where coherence is witnessed by

```
decr-extract-coherence :
  {r : Nat} (h : BHeap r) → mapMaybe (toBin ∘ proj₂) (extract h) ≡ decr (toBin h)
decr-extract-coherence = Upgrade.coherence upg decr extractᵁ
```

## 5.5 *Minimum extraction*

The type of the *extract* operation guarantees that it extracts from a heap of rank $r$ a subtree of rank $r$, but places no constraints on which particular subtree of that rank. (In fact, the implementation picks the subtree containing the root out of the "first" tree present in the heap, that is, the one with lowest rank; but that fact is not captured in the type.) A more useful operation on min-heaps is to extract the minimum element of the heap. Because of the heap ordering property, this minimum element will be the root of some tree, but not necessarily of the first tree. But we can preprocess the heap to ensure that the minimum element is the root of the first tree; then *extract* will indeed return a subtree containing the minimum element as its root.

We say that a heap is *normalised* if the minimum element is the root of the first tree. A normalised heap is of course a heap; we express the normalisation property as a refined datatype $\mathsf{BHeap}^{\mathsf{N}}$ of heaps, carrying an extra boolean index stating whether the heap is normalised. The relationship between ordinary heaps BHeap and this refinement is an ornamentation. Actually, we do the ornamentation in two steps: First to a datatype $\mathsf{BHeap}^{\mathsf{M}}$ that indexes a heap by its minimum element, and second to $\mathsf{BHeap}^{\mathsf{N}}$ to state whether the root of the first tree has this minimum value. We can normalise a heap by swapping the first tree with an equally sized subtree of the tree with the minimum root. Finally, we get for free the forgetful mapping back from $\mathsf{BHeap}^{\mathsf{N}}$ via $\mathsf{BHeap}^{\mathsf{M}}$ to BHeap, which allows us to apply *extract*.

Compare this with the standard algorithm for extracting the minimum element from a binomial heap, namely (i) to find the tree with minimum root, (ii) separate it from the heap, (iii) extract its root, and (iv) merge its children back into the heap. This would have no connection with our *extract* function, though; since we have this operation at our disposal, it is appealing to be able to reuse it. With the standard algorithm, on the other hand, step (i) has no direct analogue in the numerical representation, so would need to be defined from first principles; and steps (ii) and (iv) amount to implementing numerical operations to set a specified 1-digit (say, the $k$th digit) to 0 and then to add back $2^k - 1$. Our algorithm is arguably conceptually simpler, as well as benefitting from the leverage provided by ornaments.

In the interests of brevity, we will provide less detail from now on — as foreshadowed on page 10, we present this section as a kind of thought experiment, and rather than defining the ornaments and deriving from them the refined datatypes and the forgetful functions, we will directly present the results that would have arisen.

We start by refining BHeap to be indexed by its minimum element. A heap of type $\mathsf{BHeap}^{\mathsf{M}}\ r\ mv$ is essentially a $\mathsf{BHeap}\ r$; but the additional index $mv$ is nothing iff the heap is empty, and otherwise $mv = \mathsf{just}\ v$ where $v$ is the minimum element of the heap. The nil and zero constructors are analogous to those for BHeap. But the one constructor of BHeap is refined here into three constructors: min indicates that the minimum element is the root of the first tree, and there are more trees to follow; sin indicates that the minimum element is again the root of the first tree, but that there are no more trees to follow; and one indicates that the minimum element is the root of a higher ranked tree, not the first tree. Note that in the min and sin cases, the element $v$ is not duplicated: Instead of storing the original tree, they store only its forest of children.

```
indexfirst data BHeap^M : Nat → Maybe Val → Set where
  BHeap^M r nothing  ∋ nil
  BHeap^M r (just v) ∋ min (BForest r v) {v′ : Val} ⦃v ≤ v′⦄
                            (BHeap^M (suc r) (just v′))
                     | sin (t : BForest r v) (BHeap^M (suc r) nothing)
                     | one (t : BTree r) ⦃v ≤ root t⦄ (BHeap^M (suc r) (just v))
  BHeap^M r mv       ∋ zero (BHeap^M (suc r) mv)
```

The function $toBHeap^M$ lifts an ordinary BHeap to a more informative $\mathsf{BHeap}^{\mathsf{M}}$:

```
toBHeap^M : {r : Nat} → BHeap r → Σ (Maybe Val) (BHeap^M r)
toBHeap^M nil       = ⟪ nil ⟫
```

```
toBHeapᴹ (zero  h) = let ⟪ h' ⟫ = toBHeapᴹ h in ⟪ zero h' ⟫
toBHeapᴹ (one t h) with toBHeapᴹ h
toBHeapᴹ (one t h) | nothing , h' = ⟪ sin (children t) h' ⟫
toBHeapᴹ (one t h) | just v   , h' = if root t ≤? v then ⟪ min (children t) h' ⟫
                                                      else ⟪ one t h' ⟫
```

We use the pattern synonym **pattern** ⟪_⟫ $b$ = _ , $b$ to hide the first components, which are indices in the types of the heaps and are inferable. Note that the dependent pair representation Σ (Maybe *Val*) (BHeap$^{\mathsf{M}}$ $r$) provides constant-time access to the minimum element, by first projection from the dependent pair. The function *toBHeap*$^{M}$ does some real work, refining the original one constructor into three, and cannot be expected to be automatically derivable; on the other hand, if we had expressed BHeap$^{\mathsf{M}}$ as an ornamentation of BHeap, we would have been able to derive for free the following forgetful function:

```
-- Derivable from an ornament from BHeap to BHeapᴹ
fromBHeapᴹ  : {r : Nat} {mv : Maybe Val} → BHeapᴹ r mv → BHeap r
fromBHeapᴹ nil          = nil
fromBHeapᴹ (min ts h) = one ⟪ ts ⟫ (fromBHeapᴹ h)
fromBHeapᴹ (sin  ts h) = one ⟪ ts ⟫ (fromBHeapᴹ h)
fromBHeapᴹ (one t  h) = one   t    (fromBHeapᴹ h)
fromBHeapᴹ (zero   h) = zero       (fromBHeapᴹ h)
```

The reader may have noticed that the sin constructor of BHeap$^{\mathsf{M}}$ takes a trailing heap which is necessarily empty because of the index of its type. Could we not just get rid of this seemly redundant argument? This, in fact, points out a limitation of the ornament framework (by design): Ornamentation does not change the recursive structure. The sin constructor is a refinement of the one constructor of BHeap, and has to take the same number of recursive arguments as the latter. In a sense, this problem is not one inherent to the ornament framework, but originates from our initial choice of allowing binomial heaps to be represented with redundant zeros; that is, if we started with a non-redundant representation, then sin would not have to take that argument. On the other hand, the ornament framework is indeed limited, and cannot help us to refine a redundant representation into a non-redundant one.

The next step is to introduce a further ornamentation so we can tell from the index in the type whether a binomial heap is normalised or not — in fact, it is normalised iff the first non-zero constructor is not one:

```
indexfirst data BHeapᴺ : Nat → Maybe Val → Bool → Set where
  BHeapᴺ r nothing true  ∋ nil
  BHeapᴺ r (just v) true  ∋ min (BForest r v) {v' : Val} ⦃v ≤ v'⦄
                                {norm : Bool} (BHeapᴺ (suc r) (just v') norm)
                          | sin (t : BForest r v) {norm : Bool}
                                (BHeapᴺ (suc r) nothing norm)
  BHeapᴺ r (just v) false ∋ one (t : BTree r) ⦃v ≤ root t⦄
                                {norm : Bool} (BHeapᴺ (suc r) (just v) norm)
  BHeapᴺ r mv       norm  ∋ zero (BHeapᴺ (suc r) mv norm)
```

As before, if we had defined $\mathsf{BHeap}^{\mathsf{N}}$ as an ornamentation of $\mathsf{BHeap}^{\mathsf{M}}$, then we would have got for free the forgetful function:

```
-- Derivable from an ornament from BHeap^M to BHeap^N
fromBHeap^N  : {r : Nat} {mv : Maybe Val} {norm : Bool} →
                   BHeap^N r mv norm → BHeap^M r mv
fromBHeap^N nil          = nil
fromBHeap^N (min ts h)   = min ts (fromBHeap^N h)
fromBHeap^N (sin  ts h)  = sin  ts (fromBHeap^N h)
fromBHeap^N (one t   h)  = one t   (fromBHeap^N h)
fromBHeap^N (zero    h)  = zero    (fromBHeap^N h)
```

In addition, the relationship between $\mathsf{BHeap}^{\mathsf{M}}$ and $\mathsf{BHeap}^{\mathsf{N}}$ (at least in a non-indexed-first variant on the boolean index) turns out to be an *algebraic ornamentation* (McBride, 2011; Ko & Gibbons, 2013b), so we could get not only the forgetful function from $\mathsf{BHeap}^{\mathsf{N}}$ to $\mathsf{BHeap}^{\mathsf{M}}$ but also its converse:

```
-- Derivable from an algebraic ornament from BHeap^M to BHeap^N
toBHeap^N  : {r : Nat} {mv : Maybe Val} → BHeap^M r mv → Σ Bool (BHeap^N r mv)
toBHeap^N nil         = ⟪ nil ⟫
toBHeap^N (min t h)   = let ⟪ h′ ⟫ = toBHeap^N h in ⟪ min t h′ ⟫
toBHeap^N (sin  t h)  = let ⟪ h′ ⟫ = toBHeap^N h in ⟪ sin  t h′ ⟫
toBHeap^N (one t h)   = let ⟪ h′ ⟫ = toBHeap^N h in ⟪ one t h′ ⟫
toBHeap^N (zero   h)  = let ⟪ h′ ⟫ = toBHeap^N h in ⟪ zero   h′ ⟫
```

*Gap 4* (*representation optimisation for algebraic ornamentation*)
None of the existing formulations of general algebraic ornamentation (McBride, 2011; Dagand & McBride, 2014; Ko & Gibbons, 2013b) takes representation optimisation into account — they all work by inserting some equality constraints as fields into descriptions, and cannot further classify constructors in an index-first manner, which happened for $\mathsf{BHeap}^{\mathsf{N}}$. (Dagand & McBride (2014)'s reornaments, whilst being algebraic and achieving representation optimisation, are specialised only to the ornamental algebras.) Ko & Gibbons (2013b) noted that, in order to fully exploit the optimising power of index-first inductive families, we might need to switch to *coalgebraic ornamentation* (for inductive datatypes), which remains to be investigated.

Now, normalisation takes a binomial heap with information about ordering amongst the roots of the binomial trees and turns it into a normalised binomial heap. Note that working on $\mathsf{BHeap}^{\mathsf{M}}$ instead of $\mathsf{BHeap}$ means that all ordering information is reified through different constructors, so we can simply use pattern matching to determine whether we are looking at a binomial tree whose root is the minimum, instead of having to perform comparisons and deal with ordering proofs on the fly.

```
normalise  : {r : Nat} {mv : Maybe Val} → BHeap^M r mv → BHeap^N r mv true
normalise nil        = nil
normalise (min t h)  = let ⟪ h′ ⟫ = toBHeap^N h in min t h′
normalise (sin  t h)  =  let ⟪ h′ ⟫  =  toBHeap^N h in sin  t h′
normalise (one t h) with normalise-aux h
normalise (one t h) | us , f with f t
normalise (one t h) | us , _ | ⟪ ⟨ h′ ⟩ ⟫  =  min us h′
normalise (zero  h)  =  zero (normalise h)
```

We introduce another pattern synonym **pattern** ⟨_⟩ *b* = *b* , _ to hide inferable second components. The only complicated case in *normalise* is for the one constructor, where the root of the first tree is not the minimum; this case is handled by an auxiliary function *normalise-aux*. This function takes the trailing heap, extracts the forest under the minimum root (which is determined by the index), and also returns the rest of the heap but with a hole into which the first tree can be inserted. The size and ordering constraints are carefully encoded in the type of the function and kept track of by AGDA; with some effort we can hide all this extra information to some extent, and present the program almost as if it were simply typed.

```
normalise-aux  :
  {r  : Nat} {v  : Val} → BHeap^M (suc r) (just v) →
  BForest r v ×
  ((u  : BTree r) {{_  : v ≤ root u}} →
     Σ (Val × Bool) λ {(v′ , norm) → Σ^I [ _ ∈ BHeap^N (suc r) (just v′) norm ]  v ≤ v′})
normalise-aux (min (t ◃ ts) {v′} h)  =
  forest ts , λ u  ↦ let ⟨ u′ ⟩  =  link′ t u
                         ⟪ h′ ⟫  =  toBHeap^N h
                     in if root u′ ≤? v′ then ⟪ ⟨ min (children u′) h′ ⟩ ⟫
                                         else ⟪ ⟨ one u′ h′              ⟩ ⟫
normalise-aux (sin (t ◃ ts) h)  =
  forest ts , λ u  ↦ let ⟨ u′ ⟩  =  link′ t u
                         ⟪ h′ ⟫  =  toBHeap^N h
                     in ⟪ ⟨ sin (children u′) h′ ⟩ ⟫
normalise-aux (one t h) with normalise-aux h
normalise-aux (one t h) | (t′ ◃ ts) , f  =
  forest ts , λ u  ↦ let ⟨ u′ ⟩  =  link′ t′ u
                         ((v′ , _) , ⟨ h′ ⟩)  = f u′
                     in if root t ≤? v′ then ⟪ ⟨ min (children t) h′ ⟩ ⟫
                                        else ⟪ ⟨ one t h′              ⟩ ⟫
normalise-aux (zero h) with normalise-aux h
normalise-aux (zero h) | (t ◃ ts) , f  =
  forest ts , λ u  ↦ let ⟨ u′ ⟩  =  link′ t u
                         ⟪ ⟨ h′ ⟩ ⟫  = f u′
                     in ⟪ ⟨ zero h′ ⟩ ⟫
```

Here, *link′* is a more informative version of the function *link* from Section 5.1, not only linking together two trees of some rank *r* to a single tree of rank suc *r*, but also combining lower bounds for the two input roots into a lower bound for the output root:

```
link′  : {r  : Nat} (t t′  : BTree r) {v  : Val} {{_  : v ≤ root t}}
           {{_  : v ≤ root t′}} → Σ^I [ u ∈ BTree (suc r) ]  v ≤ root u
```

When used in the context of *normalise* (one $t$ $h$), the type of *normalise-aux* declares that it takes $h$ and returns a pair, whose first component is the children of the minimum-rooted tree, and whose second component is a heap with a hole able to accommodate $t$. We will not go through the implementation, but the rank and ordering constraints implicit in the uses of the $\langle\_\rangle$ constructor and the $\langle\!\langle\_\rangle\!\rangle$ pattern synonym mean that it would be hard to make mistakes; the only constraints that are not currently expressed are that the output is a permutation of the input, for which some kind of linear typing seems appropriate, as noted by McBride (2014).

*Gap 5* (*hiding inferable terms*)
The most important result that function *normalise-aux* returns is a $\mathsf{BHeap}^{\mathsf{N}}$, but this is buried inside several levels of nested tuples, which provide type indices and ordering proofs. Similar things can be said for *link′* and $toBHeap^{N}$. To prevent the program from being cluttered by these indices and proofs, we have used the $\langle\_\rangle$ constructor and the $\langle\!\langle\_\rangle\!\rangle$ pattern synonym to make them less noticeable typographically. This points to the need for some mechanism for eliding such ancillary data, and a nice implementation of dependent intersection types (Kopylov, 2003) could be helpful here. Bernardy & Moulin's work on "type theory in colour" (2013) might also be helpful, but it does not solve our problem directly, because in our index-first setting the indices are used in a computationally relevant way.

Finally, we can implement *extract-min* by turning a BHeap into a $\mathsf{BHeap}^{\mathsf{M}}$ (i.e., establishing ordering amongst the roots), normalising it to a $\mathsf{BHeap}^{\mathsf{N}}$, and erasing it to a BHeap, whose first tree contains the minimum as its root, and then applying *extract*.

$$\begin{aligned} &extract\text{-}min \;:\; \{r \;:\; \mathsf{Nat}\} \to \mathsf{BHeap}\; r \to \mathsf{Maybe}\;(\mathsf{BTree}\; r \times \mathsf{BHeap}\; r) \\ &extract\text{-}min \;=\; extract \circ fromBHeap^{M} \circ fromBHeap^{N} \circ normalise \circ \mathsf{proj}_2 \circ toBHeap^{M} \end{aligned}$$

We remark that, up to now, we have not actually proved that the value extracted by *extract-min* is indeed the minimum. That would require us to start from basic definitions such as "the minimum element in a heap"; the whole proof does not have much to do with ornamentation, so we omit it here (but include it in the supplementary code). But it is worth noting that the sophisticated use of indexed datatypes does facilitate the proof — the indices carry enough information for most properties to be established by straightforward induction on them (i.e., simple recursive programs defined by pattern matching).

## 6 Discussion

This paper stems from a part of Chapter 3 of the first author's DPhil dissertation (Ko, 2014), which in turn is partly based on both authors' paper in *Progress in Informatics* (Ko & Gibbons, 2013a). The definition of the optimised promotion predicate in terms of parallel composition first appeared in Ko & Gibbons (2013a), and the mechanism of promotions and upgrades is inspired by Dagand & McBride (2014) and developed in Ko (2014) as a more straightforward and flexible alternative. In this paper, we make a slight simplification of the function promotion mechanism, and present one way to deal with higher order functions. Ko (2014) uses insertion into binomial heaps as an example; that example is fully developed in this paper, by also lifting decrement to extraction and then to minimum extraction via adding a normalising step.

Williams *et al.* (2014) have also conducted a case study on ornaments, lifting OCaml library functions semi-automatically to more informative datatypes. Their case study is larger, but mainly deals with only non-dependent types. The real power of ornaments lies in its ability to relate inductive families, and we believe that the case study about binomial heaps — which tries to push programming with inductive families to an extreme — has helped to point out how ornaments can help in a full dependently typed setting, and what conveniences are still lacking to support programming with ornaments.

We have seen that, in the case study on binomial heaps, the ornament framework is capable of generating all sorts of types and operations that help the programmer to move between more informative and less informative variants of datatypes, namely $\mathsf{Bin}$ (binary numbers), $\mathsf{BHeap}$ (binomial heaps indexed with starting rank), $\mathsf{BHeap}^{\mathsf{U}}$ ($\mathsf{BHeap}$ indexed additionally with the underlying binary number), $\mathsf{BHeap}^{\mathsf{M}}$ ($\mathsf{BHeap}$ indexed additionally with the minimum element), and $\mathsf{BHeap}^{\mathsf{N}}$ ($\mathsf{BHeap}^{\mathsf{M}}$ indexed additionally with a normalisation flag). The ornament framework also effectively guides the lifting of increment and decrement to insertion and extraction, and establishes coherence by construction. Finally, by carefully designing the datatypes $\mathsf{BHeap}^{\mathsf{M}}$ and $\mathsf{BHeap}^{\mathsf{N}}$ and employing McBride (2014)'s instance argument technique for managing ordering witnesses, we are able to express the heap normalisation algorithm (for minimum extraction) in a syntactically simple program which is nevertheless guaranteed by its type to maintain the rank and ordering constraints.

During our development, we have also encountered some inconveniences, which we identify as "gaps". Of the five gaps we identify (on pages 9, 10, 16, 38, and 40), Gaps 1, 2, and 3 are about higher level presentation of index-first inductive families and ornaments, and Gap 5 is about hiding inferable terms in programs — these all call for better language support, so that we could read and write dependently typed programs more easily. The remaining Gap 4 concerns coalgebraic ornamentation, which calls for further investigation into programming methodology. We look forward to advances that can fill these gaps.

## Acknowledgments

Most of this work was done whilst the first author was in Oxford, supported by a Clarendon Scholarship and by the UK EPSRC-funded project *Reusability and Dependent Types* (EP/G034516/1). The first author then relocated to Japan and is supported by the Japan Society for the Promotion of Science (JSPS) Grant-in-Aid for Scientific Research (A) No. 25240009. We are indebted to Conor McBride for inspiration, not least in introducing us to ornaments and index-first datatypes. We would also like to thank Ralf Hinze and Venanzio Capretta for their insightful comments on the binomial heaps development during the first author's DPhil viva.

Thanks are also due to the anonymous referees for their thorough reviewing and invaluable comments.

## Supplementary Material

To view supplementary material for this article, please visit https://doi.10.1017/S0956796816000307.

## References

Bernardy, J.-P. & Moulin, G. (2013) Type theory in color. In Proceedings of International Conference on Functional Programming, Tarmo Uustalu (ed), ICFP'13. New York, NY, USA: ACM, pp. 61–72.

Bove, A. & Dybjer, P. (2009) Dependent types at work. In *Language Engineering and Rigorous Software Development*, Ana Bove, Luís Soares Barbosa, Alberto Pardo, and Jorge Sousa Pinto (eds), Lecture Notes in Computer Science, vol. 5520. Berlin, Germany: Springer-Verlag, pp. 57–99.

Brady, E., McBride, C. & McKinna, J. (2004) Inductive families need not store their indices. In *Types for Proofs and Programs*, Stefano Berardi, Mario Coppo, and Ferruccio Damiani (eds), Lecture Notes in Computer Science, vol. 3085. Berlin, Germany: Springer-Verlag, pp. 115–129.

Chapman, J., Dagand, P.-É., McBride, C. & Morris, P. (2010) The gentle art of levitation. In Proceedings of International Conference on Functional Programming, Stephanie Weirich (ed), ICFP'10. New York, NY, USA: ACM, pp. 3–14.

Cockx, J., Devriese, D. & Piessens, F. (2014) Pattern matching without K. In International Conference on Functional Programming, Manuel M.T. Chakravarty (ed), ICFP'14. New York, NY, USA: ACM, pp. 257–268.

Czarnecki, K., Foster, J. N., Hu, Z., Lämmel, R., Schürr, A. & Terwilliger, J. F. (2009) Bidirectional transformations: A cross-discipline perspective. In Proceedings of International Conference on Model Transformation, Richard F. Paige (ed), Lecture Notes in Computer Science, vol. 5563. Berlin, Germany: Springer-Verlag, pp. 260–283.

Dagand, P.-É. & McBride, C. (2013) *Elaborating Inductive Definitions.* In Journées Francophones des Langages Applicatifs, JFLA'13. Rocquencourt, France: INRIA.

Dagand, P.-É. & McBride, C. (2014) Transporting functions across ornaments. *J. Funct. Program.* **24**(2–3), 316–383.

Devriese, D. & Piessens, F. (2011) On the bright side of type classes: Instance arguments in Agda. In Proceedings of International Conference on Functional Programming, Olivier Danvy (ed), ICFP'11. New York, NY, USA: ACM, pp. 143–155.

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

Goguen, H., McBride, C. & McKinna, J. (2006) Eliminating dependent pattern matching. In Algebra, Meaning, and Computation, Kokichi Futatsugi, Jean-Pierre Jouannaud, and José Meseguer, (eds), Lecture Notes in Computer Science, vol. 4060. Berlin, Germany: Springer-Verlag, pp. 521–540.

Ko, H.-S. (2014) *Analysis and Synthesis of Inductive Families.* DPhil Thesis, University of Oxford.

Ko, H.-S. & Gibbons, J. (2013a) Modularising inductive families. *Prog. Informat.* **10**, 65–88.

Ko, H.-S. & Gibbons, J. (2013b) Relational algebraic ornaments. In *Dependently Typed Programming*, Stephanie Weirich (ed), DTP'13. New York, NY, USA: ACM, pp. 37–48.

Kopylov, A. (2003) Dependent intersection: A new way of defining records in type theory. In *Logic in Computer Science*, Phokion G. Kolaitis (ed), LICS'03. Washington, DC, USA: IEEE, pp. 86–95.

Martin-Löf, P. (1984) *Intuitionistic Type Theory*. Bibliopolis, Napoli.

McBride, C. (2011) *Ornamental Algebras, Algebraic Ornaments*. Unpublished manuscript.

McBride, C. (2014) How to keep your neighbours in order. In Proceedings of International Conference on Functional Programming, Manuel M.T. Chakravarty (ed), ICFP'14. New York, NY, USA: ACM, pp. 297–309.

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

Norell, U. (2007) *Towards a Practical Programming Language Based on Dependent Type Theory*. PhD Thesis, Chalmers University of Technology.

Norell, U. (2009) Dependently typed programming in Agda. In *Advanced Functional Programming*, Pieter Koopman, Rinus Plasmeijer, and Doaitse Swierstra (eds), Lecture Notes in Computer Science, vol. 5832. Berlin, Germany: Springer-Verlag, pp. 230–266.

Okasaki, C. (1999) *Purely Functional Data Structures*. Cambridge University Press.

Sheard, T. & Linger, N. (2007) Programming in Ωmega. In *Central European Functional Programming School*, Zoltán Horváth, Rinus Plasmeijer, Anna Soós, and Viktória Zsók (eds), Lecture Notes in Computer Science, vol. 5161. Berlin, Germany: Springer-Verlag, pp. 158–227.

Williams, T., Dagand, P.-É. & Rémy, D. (2014) Ornaments in practice. In *Workshop on Generic Programming*, José Pedro Magalhäes and Tiark Rompf (eds), WGP'14. New York, NY, USA: ACM, pp. 15–24.