Chapter 3

Refinements and ornaments

This chapter begins our exploration of the interconnection between internalism and externalism by looking at **the analytic direction**, i.e., the decomposition of sophisticated types into basic types and predicates on them. More specifically, we assume that the more sophisticated type and the basic type are known and there is straightforward evidence that they are related, and derive from the evidence an externalist predicate and a conversion isomorphism. As discussed in **??**, one purpose of such decomposition is for internalist datatypes and operations to take a round trip to the externalist world so as to harvest composability there. The abstractions and constructions facilitating externalist composition of internalist datatypes and operations are developed in this chapter as follows:

- Conversion isomorphisms between internalist and externalist datatypes are axiomatised as **refinements** (Section 3.1).
- Refinements are coordinated by **upgrades** (Section 3.1.2) to enable switching from externalist function types to internalist ones, so simply typed operations satisfying suitable properties can be upgraded to have more sophisticated types.
- A class of refinements can be mechanically derived (Section 3.3) by marking differences between datatype descriptions with an adapted version of McBride's **ornaments** [2011] (Section 3.2), which relate datatype descrip-

tions that are vertically the same but horizontally different. (For example, the datatypes of vectors and lists can be related by an ornament.)

- Ornaments can be composed in parallel (Section 3.2.3), producing new composite internalist datatypes which correspond to pointwise conjunction of externalist predicates (Section 3.3.2). (For example, ordered vectors can be produced by composing the ornament from lists to ordered lists and the ornament from lists to vectors in parallel.)

3.1 Refinements

3.1.1 Refinements between individual types

A **refinement** from a basic type X to a more informative type Y is a **promotion predicate** $P : X \to \mathsf{Set}$ and a **conversion isomorphism** $i : Y \cong \Sigma\, X\, P$.

```
record Refinement (X Y : Set) : Set₁ where
  field
    P : X → Set
    i : Y ≅ Σ X P
  forget : Y → X
  forget = outl ∘ Iso.to i
```

Refinements are not guaranteed to be interesting in general. For example, Y can be chosen to be $\Sigma\ X\ P$ and the conversion isomorphism simply the identity. Most of the time, however, we will only be interested in refinements from basic types to their more informative — often internalist — variants. The conversion isomorphism tells us that the inhabitants of Y exactly correspond to the inhabitants of X bundled with more information, i.e., proofs that the promotion predicate P is satisfied. Computationally, any inhabitant of Y can be decomposed (by $\mathsf{Iso}.to\ i$) into an underlying value $x : X$ and a proof that x satisfies the promotion predicate P (which we will call a **promotion proof** for x), and conversely, if $x : X$ satisfies P, then it can be promoted (by $\mathsf{Iso}.from\ i$) to an inhabitant of Y.

Example (*refinement from lists to ordered lists*)**.** Suppose A : Set is equipped with an ordering $\_\leqslant_A\_$. Fixing b : A, there is a refinement from List A to OrdList A $\_\leqslant_A\_$ b whose promotion predicate is Ordered A $\_\leqslant_A\_$ b, since we have an isomorphism of type

$$\mathsf{OrdList}\ A\ \_\leqslant_A\_\ b\ \cong\ \Sigma\ (\mathsf{List}\ A)\ (\mathsf{Ordered}\ A\ \_\leqslant_A\_\ b)$$

An ordered list of type OrdList A $\_\leqslant_A\_$ b can be decomposed into a list *as* : List A and a proof of type Ordered A $\_\leqslant_A\_$ b *as* that the list *as* is ordered and bounded below by b; conversely, a list satisfying Ordered A $\_\leqslant_A\_$ b can be promoted to an ordered list of type OrdList A $\_\leqslant_A\_$ b. □

XD

Example (*refinement from natural numbers to lists*)**.** Let A : Set. We have a refinement from Nat to List A

$$\mathit{Nat\text{-}List}\ A\ :\ \mathsf{Refinement}\ \mathsf{Nat}\ (\mathsf{List}\ A)$$

for which Vec A serves as the promotion predicate — there is a conversion isomorphism of type

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

whose decomposing direction computes from a list its length and a vector containing the same elements. We might say that a natural number n : Nat is an incomplete list — the list elements are missing from the successor nodes of n. To promote n to a List A, we need to supply a vector of type Vec A n, i.e., n elements of type A. This example helps to emphasise that the notion of refinements is **proof-relevant**: An underlying value can have more than one promotion proofs, and consequently the more informative type in a refinement can have more elements than the basic type does. Thus it is more helpful to think that a type is more refined in the sense of being more informative rather than being a subset. □

In a refinement r, we denote the forgetful computation of underlying values — i.e., outl ∘ Iso.*to* (Refinement.*i* r) — as Refinement.*forget* r. The forgetful function is actually the core of a refinement, which is justified by the following facts:

- The forgetful function determines a refinement extensionally — if the forget-

ful functions of two refinements are extensionally equal, then their promotion predicates are pointwise isomorphic:

$$\begin{aligned}&\mathit{forget\text{-}iso} : \{X\ Y : \mathsf{Set}\}\ (r\ s : \mathsf{Refinement}\ X\ Y) \to \\ &\qquad (\mathsf{Refinement}.\mathit{forget}\ r \doteq \mathsf{Refinement}.\mathit{forget}\ s) \to \\ &\qquad (x : X) \to \mathsf{Refinement}.P\ r\ x \cong \mathsf{Refinement}.P\ s\ x\end{aligned}$$

- From any function f, we can construct a **canonical refinement** which uses a simplistic promotion predicate and has f as its forgetful function:

$$\begin{aligned}&\mathit{canonRef} : \{X\ Y : \mathsf{Set}\} \to (Y \to X) \to \mathsf{Refinement}\ X\ Y \\ &\mathit{canonRef}\ \{X\}\ \{Y\}\ f = \mathbf{record} \\ &\quad \{\ P = \lambda\, x \mapsto \Sigma[\, y : Y\,]\ f\ y \equiv x \\ &\quad ;\ i = \mathbf{record}\ \{\ \mathit{to} \quad = f \vartriangle (\mathit{id} \vartriangle (\lambda\, y \mapsto \mathsf{refl})) \\ &\qquad\qquad\ ;\ \mathit{from} = \mathsf{outl} \circ \mathsf{outr} \\ &\qquad\qquad\ ;\ \text{proofs of laws}\ \}\ \}\end{aligned}$$

  We call $\lambda\, x \mapsto \Sigma[\, y : Y\,]\ f\ y \equiv x$ the **canonical promotion predicate**, which says that, to promote $x : X$ to type Y, we are required to supply a complete $y : Y$ and prove that its underlying value is x.

- For any refinement $r : \mathsf{Refinement}\ X\ Y$, its forgetful function is exactly that of *canonRef* $(\mathsf{Refinement}.\mathit{forget}\ r)$, so from *forget-iso* we can prove that a promotion predicate is always pointwise isomorphic to the canonical promotion predicate:

$$\begin{aligned}&\mathit{coherence} : \{X\ Y : \mathsf{Set}\}\ (r : \mathsf{Refinement}\ X\ Y) \to \\ &\qquad (x : X) \to \mathsf{Refinement}.P\ r\ x \\ &\qquad\qquad\qquad \cong \Sigma[\, y : Y\,]\ \mathsf{Refinement}.\mathit{forget}\ r\ y \equiv x \\ &\mathit{coherence}\ r\ x = \mathit{forget\text{-}iso}\ r\ (\mathit{canonRef}\ (\mathsf{Refinement}.\mathit{forget}\ r))\ (\lambda\, y \mapsto \mathsf{refl})\end{aligned}$$

  This is closely related to an alternative "coherence-based" definition of refinements, which will shortly be discussed.

The refinement mechanism's purpose of being is thus to express intensional (representational) optimisations of the canonical promotion predicate, such that it is possible work on just the residual information of the more refined type that is not present in the basic type.

Example (*promoting lists to ordered lists*)**.** Consider the refinement from lists to ordered lists using Ordered as its promotion predicate. A promotion proof of type Ordered $A\ \_\leqslant_A\_\ b\ as$ for the list as consists of only the inequality proofs necessary for ensuring that as is ordered and bounded below by b. Thus, to promote a list to an ordered list, we only need to supply the inequality proofs without providing the list elements again. □

Coherence-based definition of refinements

There is an alternative definition of refinements which, instead of the conversion isomorphism, postulates the forgetful computation and characterises the promotion predicate in term of it:

$$
\begin{array}{l}
\textbf{record}\ \mathsf{Refinement}'\ (X\ Y\ :\ \mathsf{Set})\ :\ \mathsf{Set}_1\ \textbf{where} \\
\quad \textbf{field} \\
\qquad P\ :\ X \to \mathsf{Set} \\
\qquad \mathit{forget}\ :\ Y \to X \\
\qquad p\ :\ (x\ :\ X) \to P\ x\ \cong\ \Sigma[\, y : Y \,]\ \mathit{forget}\ y\ \equiv\ x
\end{array}
$$

We say that $x\ :\ X$ and $y\ :\ Y$ are **in coherence** when $\mathit{forget}\ y\ \equiv\ x$, i.e., when x underlies y. The two definitions of refinements are equivalent. Of particular importance is the direction from Refinement to Refinement′:

$$
\begin{array}{l}
\mathit{toRefinement}'\ :\ \{X\ Y\ :\ \mathsf{Set}\} \to \mathsf{Refinement}\ X\ Y \to \mathsf{Refinement}'\ X\ Y \\
\mathit{toRefinement}'\ r\ =\ \textbf{record}\ \{\ P\ \ \ \ \ \ =\ \mathsf{Refinement}.P\ r \\
\qquad\qquad\qquad\qquad\qquad\quad ;\ \mathit{forget}\ =\ \mathsf{Refinement}.\mathit{forget}\ r \\
\qquad\qquad\qquad\qquad\qquad\quad ;\ p\ \ \ \ \ \ =\ \mathit{coherence}\ r\ \}
\end{array}
$$

We prefer the definition of refinements in terms of conversion isomorphisms because it is more concise and directly applicable to function upgrading. The coherence-based definition, however, is easier to generalise for function types, as we will see below.

3.1.2 Upgrades

Refinements are less useful when we move on to function types: the requirement that a conversion isomorphism exists between related function types is too strong, even when we have extensional equality for functions so isomorphisms between function types make more sense. For example, it is not — and should not be — possible to have a refinement from the function type $\mathsf{Nat} \to \mathsf{Nat}$ to the function type $\mathsf{List}\ \mathsf{Nat} \to \mathsf{List}\ \mathsf{Nat}$, despite that the component types Nat and List Nat are related by a refinement: If such a refinement existed, we would be able to extract from any function $f : \mathsf{List}\ \mathsf{Nat} \to \mathsf{List}\ \mathsf{Nat}$ an "underlying" function of type $\mathsf{Nat} \to \mathsf{Nat}$ which has roughly the same behaviour as f. However, the behaviour of a function taking a list may depend essentially on the list elements, which is not available to a function taking only a natural number. For example, a function of type $\mathsf{List}\ \mathsf{Nat} \to \mathsf{List}\ \mathsf{Nat}$ might compute the sum s of the input list and emit a list of length s whose elements are all zero. We cannot hope to write a function of type $\mathsf{Nat} \to \mathsf{Nat}$ that reproduces the corresponding behaviour on natural numbers.

It is only the decomposing direction of refinements that causes problem in the case of function types, however; the promoting direction is perfectly valid for function types. For example, to promote the function

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

to a function of type $\mathsf{List}\ A \to \mathsf{List}\ A$ for some fixed $A : \mathsf{Set}$, we can use

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

as the promotion predicate: Consider the refinement from Nat to List A. Given a promotion proof of type $Q\ \mathit{double}$, say

Explain the meaning of this (scoping).

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

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

definition of $*$

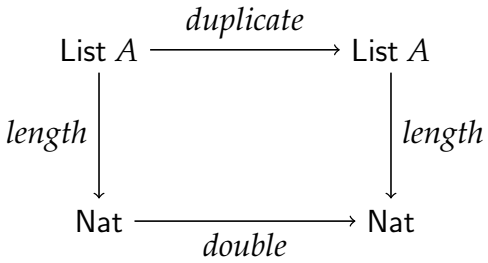
$$duplicate \;=\; \mathsf{Iso}.from\ i \circ (double * duplicate'\ \_) \circ \mathsf{Iso}.to\ i$$

i.e., 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 relationship between the promoted function *duplicate* and the underlying function *double* is characterised by the coherence property [Dagand and McBride, 2012]

definition of pointwise equality

$$double \circ length \;\doteq\; length \circ duplicate$$

or as a commutative diagram:



which states that *duplicate* preserves length as computed by *double*, or in more generic terms, processes the recursive structure (i.e., nil and cons nodes) of its input in the same way as *double* does.

We thus define **upgrades** to capture the promoting direction and the coherence property abstractly. An upgrade from $X : \mathsf{Set}$ to $Y : \mathsf{Set}$ is a promotion predicate $P : X \to \mathsf{Set}$, a coherence property $C : X \to Y \to \mathsf{Set}$ relating basic elements of type X and promoted elements of type Y, an upgrading (promoting) operation $u : (x : X) \to P\ x \to Y$, and a coherence proof $c : (x : X)\ (p : P\ x) \to C\ x\ (u\ x\ p)$ saying that the result of promoting a basic element $x : X$ must be in coherence with x.

record Upgrade $(X\ Y : \mathsf{Set}) : \mathsf{Set}_1$ **where**
  field
    $P : X \to \mathsf{Set}$
    $C : X \to Y \to \mathsf{Set}$
    $u : (x : X) \to P\ x \to Y$
    $c : (x : X)\ (p : P\ x) \to C\ x\ (u\ x\ p)$

Like refinements, arbitrary upgrades are not guaranteed to be interesting, but we will only use the upgrades synthesised by the combinators we define below specifically for deriving coherence properties and upgrading operations for function types from refinements between component types.

Upgrades from refinements

As we said, upgrades amount to only the promoting direction of refinements. This is most obvious when we look at the coherence-based refinements, of which upgrades are a direct generalisation: we get from $\mathsf{Refinement}'$ to $\mathsf{Upgrade}$ by abstracting the notion of coherence and weakening the isomorphism to only the left-to-right computation. Any coherence-based refinement can thus be weakened to an upgrade,

$$
\begin{aligned}
&\mathit{toUpgrade}' \,:\, \{X\ Y \,:\, \mathsf{Set}\} \to \mathsf{Refinement}'\ X\ Y \to \mathsf{Upgrade}\ X\ Y \\
&\mathit{toUpgrade}'\ r \;=\; \mathbf{record}\ \{\ P \;=\; \mathsf{Refinement}'.P\ r \\
&\qquad\qquad\qquad\qquad ;\ C \;=\; \lambda\, x\ y \mapsto \mathsf{Refinement}'.\mathit{forget}\ r\ y \equiv x \\
&\qquad\qquad\qquad\qquad ;\ u \;=\; \lambda\, x \mapsto \mathsf{outl} \circ \mathsf{Iso}.\mathit{to}\ (\mathsf{Refinement}'.p\ r\ x) \\
&\qquad\qquad\qquad\qquad ;\ c \;=\; \lambda\, x \mapsto \mathsf{outr} \circ \mathsf{Iso}.\mathit{to}\ (\mathsf{Refinement}'.p\ r\ x)\ \}
\end{aligned}
$$

and consequently any refinement gives rise to an upgrade.

$$
\begin{aligned}
&\mathit{toUpgrade} \,:\, \{X\ Y \,:\, \mathsf{Set}\} \to \mathsf{Refinement}\ X\ Y \to \mathsf{Upgrade}\ X\ Y \\
&\mathit{toUpgrade} \;=\; \mathit{toUpgrade}' \circ \mathit{toRefinement}'
\end{aligned}
$$

Composition of upgrades

The most representative combinator for upgrades is the following one for synthesising upgrades between function types:

$$
\begin{aligned}
&\_\rightharpoonup\_ \,:\, \{X\ Y\ Z\ W \,:\, \mathsf{Set}\} \to \\
&\qquad \mathsf{Refinement}\ X\ Y \to \mathsf{Upgrade}\ Z\ W \to \mathsf{Upgrade}\ (X \to Z)\ (Y \to W)
\end{aligned}
$$

Note that there should be a refinement between the source types X and Y, rather than just an upgrade. (As a consequence, we can produce upgrades

between curried multi-argument function types but not between higher-order function types.) This is because, as we see in the *double–duplicate* example, we need the ability to decompose the source type Y.

Let r : Refinement X Y and s : Upgrade Z W. The upgrading operation takes a function $f : X \to Z$ and combines it with a promotion proof to get a function $g : Y \to W$, which should transform underlying values in coherence with f. That is, as g takes $y : Y$ to $g\ y : W$ at the more informative level, correspondingly at the underlying level the value Refinement.*forget* $r\ y : X$ underlying y should be taken by f to a value in coherence with $g\ y$. We thus define the statement "g is in coherence with f" as

$$(x : X)\ (y : Y) \to \mathsf{Refinement}.\mathit{forget}\ r\ y \equiv x \to \mathsf{Upgrade}.C\ s\ (f\ x)\ (g\ y)$$

As for the type of promotion proofs, since we already know that the underlying values are transformed by f, the missing information is only how the residual parts are transformed — that is, we need to know for any $x : X$ how a promotion proof for x is transformed to a promotion proof for $f\ x$. The type of promotion proofs for f is thus

$$(x : X) \to \mathsf{Refinement}.P\ r\ x \to \mathsf{Upgrade}.P\ s\ (f\ x)$$

Having determined the coherence property and the promotion predicate, it is then easy to construct the upgrading operation and the coherence proof. In particular, following the *double–duplicate* example, the upgrading operation breaks an input $y : Y$ into its underlying value $x = \mathsf{Refinement}.\mathit{forget}\ r\ y : X$ and a promotion proof for x, computes a promotion proof q for $f\ x : Z$ using the given promotion proof for f, and upgrades $f\ x$ to an inhabitant of type W using q. To sum up, the complete definition of $\_\rightharpoonup\_$ is

$$\begin{aligned}
&_\rightharpoonup_ : \{X\ Y\ Z\ W : \mathsf{Set}\} \to \\
&\qquad \mathsf{Refinement}\ X\ Y \to \mathsf{Upgrade}\ Z\ W \to \mathsf{Upgrade}\ (X \to Z)\ (Y \to W) \\
&r \rightharpoonup s = \mathbf{record} \\
&\quad \{ P = \lambda f \mapsto (x : X) \to \mathsf{Refinement}.P\ r\ x \to \mathsf{Upgrade}.P\ s\ (f\ x) \\
&\quad ; C = \lambda f\ g \mapsto (x : X)\ (y : Y) \to \\
&\qquad\qquad \mathsf{Refinement}.\mathit{forget}\ r\ y \equiv x \to \mathsf{Upgrade}.C\ s\ (f\ x)\ (g\ y) \\
&\quad ; u = \lambda f\ h \mapsto \mathsf{Upgrade}.u\ s\ _ \circ \mathit{uncurry}\ h \circ \mathsf{Iso}.\mathit{to}\ (\mathsf{Refinement}.i\ r)
\end{aligned}$$

$$; c \;=\; \lambda\,\{\, f\ h\ \mathit{inferred}\ y\ \mathsf{refl} \;\mapsto\; \mathbf{let}\ (x\ ,\ p) \;=\; \mathsf{Iso}.\mathit{to}\ (\mathsf{Refinement}.i\ r)\ y \\ \mathbf{in}\ \ \mathsf{Upgrade}.c\ s\ (f\ x)\ (h\ x\ p)\ \}\ \}$$

Example (*upgrade from* Nat → Nat *to* List A → List A)**.** Using the _⇀_ combinator on the refinement

$$r \;=\; \mathit{Nat\text{-}List}\ A \;:\; \mathsf{Refinement}\ \mathsf{Nat}\ (\mathsf{List}\ A)$$

and the upgrade derived from r, we get an upgrade

$$u \;=\; r \rightharpoonup \mathit{toUpgrade}\ r \;:\; \mathsf{Upgrade}\ (\mathsf{Nat} \to \mathsf{Nat})\ (\mathsf{List}\ A \to \mathsf{List}\ A)$$

The type $\mathsf{Upgrade}.P\ u\ \mathit{double}$ is exactly the type of $\mathit{duplicate}'$, and the type $\mathsf{Upgrade}.C\ u\ \mathit{double}\ \mathit{duplicate}$ is exactly the coherence property satisfied by *double* and *duplicate*. □

Comparison (*functional ornaments*)**.**

Dagand and McBride [2012], origin of coherence property, no need to construct a universe (open for easy extension)

We can define more combinators for upgrades, like the ones in Figure 3.1. □

3.1.3 Refinement families

When we move on to consider refinements between indexed families of types, refinement relationship exists not only between the member types but also between the index sets: a type family $X : I \to \mathsf{Set}$ is refined by another type family $Y : J \to \mathsf{Set}$ when

- at the index level, there is a refinement r from I to J, and
- at the member type level, there is a refinement from $X\ i$ to $Y\ j$ whenever $i : I$ underlies $j : J$, i.e., $\mathsf{Refinement}.\mathit{forget}\ r\ j \equiv i$.

In short, each type $X\ i$ is refined by a collection of types in Y, the underlying values of their indices all being i. We will not exploit the full refinement structure on indices, though, so in the actual definition of **refinement families**

```
  -- the upgraded function type has an extra argument
new : {X : Set} (I : Set) {Y : I → Set} →
      (∀ i → Upgrade X (Y i)) → Upgrade X ((i : I) → Y i)
new I u = record { P = λ x ↦ ∀ i → Upgrade.P (u i) x
                 ; C = λ x y ↦ ∀ i → Upgrade.C (u i) x (y i)
                 ; u = λ x p i ↦ Upgrade.u (u i) x (p i)
                 ; c = λ x p i ↦ Upgrade.c (u i) x (p i) }

syntax new I (λ i ↦ u) = ∀⁺[ i : I ] u

  -- implicit version of new
new′ : {X : Set} (I : Set) {Y : I → Set} →
       (∀ i → Upgrade X (Y i)) → Upgrade X ({i : I} → Y i)
new′ I u = record { P = λ x ↦ ∀ {i} → Upgrade.P (u i) x
                  ; C = λ x y ↦ ∀ {i} → Upgrade.C (u i) x (y {i})
                  ; u = λ x p {i} ↦ Upgrade.u (u i) x (p {i})
                  ; c = λ x p {i} ↦ Upgrade.c (u i) x (p {i}) }

syntax new′ I (λ i ↦ u) = ∀⁺[[ i : I ]] u

  -- the underlying and the upgraded function types have a common argument
fixed : (I : Set) {X : I → Set} {Y : I → Set} →
        (∀ i → Upgrade (X i) (Y i)) → Upgrade ((i : I) → X i) ((i : I) → Y i)
fixed I u = record { P = λ f ↦ ∀ i → Upgrade.P (u i) (f i)
                   ; C = λ f g ↦ ∀ i → Upgrade.C (u i) (f i) (g i)
                   ; u = λ f h i ↦ Upgrade.u (u i) (f i) (h i)
                   ; c = λ f h i ↦ Upgrade.c (u i) (f i) (h i) }

syntax fixed I (λ i ↦ u) = ∀[ i : I ] u

  -- implicit version of fixed
fixed′ : (I : Set) {X : I → Set} {Y : I → Set} →
         (∀ i → Upgrade (X i) (Y i)) → Upgrade ({i : I} → X i) ({i : I} → Y i)
fixed′ I u = record { P = λ f ↦ ∀ {i} → Upgrade.P (u i) (f {i})
                    ; C = λ f g ↦ ∀ {i} → Upgrade.C (u i) (f {i}) (g {i})
                    ; u = λ f h {i} ↦ Upgrade.u (u i) (f {i}) (h {i})
                    ; c = λ f h {i} ↦ Upgrade.c (u i) (f {i}) (h {i}) }

syntax fixed′ I (λ i ↦ u) = ∀[[ i : I ]] u
```

Figure 3.1 More combinators for upgrades.

below, the index-level refinement degenerates into just the forgetful function.

$$\mathsf{FRefinement} : \{I\ J : \mathsf{Set}\}\ (e : J \to I)\ (X : I \to \mathsf{Set})\ (Y : J \to \mathsf{Set}) \to \mathsf{Set}_1$$
$$\mathsf{FRefinement}\ \{I\}\ e\ X\ Y = \{i : I\}\ (j : e^{-1}\ i) \to \mathsf{Refinement}\ (X\ i)\ (Y\ (\mathit{und}\ j))$$

Example (*refinement family from ordered lists to ordered vectors*)**.** The datatype $\mathsf{OrdList}\ A\ \_\leqslant_A\_ : A \to \mathsf{Set}$ is a family of types into which ordered lists are classified according to their lower bound. For each type of ordered lists having a particular lower bound, we can further classify them by their length, yielding $\mathsf{OrdVec}\ A\ \_\leqslant_A\_ : A \to \mathsf{Nat} \to \mathsf{Set}$. This further classification is captured as a refinement family of type

$$\mathsf{FRefinement}\ \mathsf{outl}\ (\mathsf{OrdList}\ A\ \_\leqslant_A\_)\ (\mathit{uncurry}\ (\mathsf{OrdVec}\ A\ \_\leqslant_A\_))$$

which consists of refinements from $\mathsf{OrdList}\ A\ \_\leqslant_A\_\ b$ to $\mathsf{OrdVec}\ A\ \_\leqslant_A\_\ b\ n$ for all $b : A$ and $n : \mathsf{Nat}$. □

relationship with ornaments

3.2 Ornaments

One possible way to establish relationships between datatypes is to write conversion functions. Conversions that involve only modifications of horizontal structures like copying, projecting away, or assigning default values to fields, however, may instead be stated at the level of datatype declarations, i.e., in terms of natural transformations between base functors. For example, a list is a natural number whose successor nodes are decorated with elements, and to convert a list to its length, we simply discard those elements. The essential information in this conversion is just that the elements associated with cons nodes should be discarded, which is described by the following natural transformation between the two base functors $\mathbb{F}\ (\mathit{ListD}\ A)$ and $\mathbb{F}\ \mathit{NatD}$:

$$\mathit{erase} : \{A : \mathsf{Set}\}\ \{X : \top \to \mathsf{Set}\} \to \mathbb{F}\ (\mathit{ListD}\ A)\ X \rightrightarrows \mathbb{F}\ \mathit{NatD}\ X$$
$$\mathit{erase}\ (\text{`nil}\ ,\ \blacksquare) = \text{`nil}\ ,\ \blacksquare \quad \text{-- `nil copied}$$
$$\mathit{erase}\ (\text{`cons}\ ,\ a\ ,\ x\ ,\ \blacksquare) = \text{`cons}\ ,\ x\ ,\ \blacksquare \quad \text{-- `cons copied, } a \text{ discarded,}$$

```
                                    -- and x retained
```

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

$$\mathit{length} : \{A : \mathsf{Set}\} \to \mu\ (\mathit{ListD}\ A) \rightrightarrows \mu\ \mathit{NatD}$$
$$\mathit{length}\ \{A\} = \mathit{fold}\ (\mathsf{con} \circ \mathit{erase}\ \{A\}\ \{\mu\ \mathit{NatD}\})$$

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

3.2.1 Universe construction

The definition of ornaments has the same two-level structure as that of datatype descriptions. We have an upper-level datatype Orn of ornaments

$$\mathsf{Orn} : \{I\ J : \mathsf{Set}\}\ (e : J \to I)\ (D : \mathsf{Desc}\ I)\ (E : \mathsf{Desc}\ J) \to \mathsf{Set}_1$$
$$\mathsf{Orn}\ e\ D\ E = \{i : I\}\ (j : e^{-1}\ i) \to \mathsf{ROrn}\ e\ (D\ i)\ (E\ (\mathit{und}\ j))$$

which is defined in terms of a lower-level datatype ROrn of **response ornaments**, while ROrn contains the actual encoding of horizontal transformations and is decoded by the function *erase*:

$$\textbf{data}\ \mathsf{ROrn}\ \{I\ J : \mathsf{Set}\}\ (e : J \to I) : \mathsf{RDesc}\ I \to \mathsf{RDesc}\ J \to \mathsf{Set}_1$$
$$\mathit{erase} : \{I\ J : \mathsf{Set}\}\ \{e : J \to I\}\ \{D : \mathsf{RDesc}\ I\}\ \{E : \mathsf{RDesc}\ J\} \to$$
$$\quad \mathsf{ROrn}\ e\ D\ E \to \{X : I \to \mathsf{Set}\} \to [\![\ E\ ]\!]\ (X \circ e) \to [\![\ D\ ]\!]\ X$$

The datatype Orn is parametrised by an erasure function $e : J \to I$ on the index sets and relates two datatype descriptions $D : \mathsf{Desc}\ I$ and $E : \mathsf{Desc}\ J$ such that from any ornament $O : \mathsf{Orn}\ e\ D\ E$ we can derive a forgetful map:

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

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

$$
\begin{aligned}
&\mathit{ornAlg} : \{I\ J : \mathsf{Set}\}\ \{e : J \to I\}\ \{D : \mathsf{Desc}\ I\}\ \{E : \mathsf{Desc}\ J\} \\
&\qquad (O : \mathsf{Orn}\ e\ D\ E) \to \mathbb{F}\ E\ (\mu\ D \circ e) \rightrightarrows \mu\ D \circ e \\
&\mathit{ornAlg}\ O\ \{j\} \ = \ \mathsf{con} \circ \mathit{erase}\ (O\ (\mathsf{ok}\ j))
\end{aligned}
$$

where the horizontal modification — a transformation from $[\![\ E\ j\ ]\!]\ (X \circ e)$ to $[\![\ D\ (e\ j)\ ]\!]\ X$, natural in X — is decoded by *erase* from a response ornament relating $D\ (e\ j)$ and $E\ j$. The forgetful function is then defined by

$$\mathit{forget}\ O \ = \ \mathit{fold}\ (\mathit{ornAlg}\ O)$$

Hence an ornament of type $\mathsf{Orn}\ e\ D\ E$ contains, for each index request j, a response ornament of type $\mathsf{ROrn}\ e\ (D\ (e\ j))\ (E\ j)$ to cope with all possible horizontal structures that can occur in a $(\mu\ E)$–inhabitant. The definition of Orn given above is a restatement of this in an intensionally more flexible form.

connection to refinement families

Now we look at the definitions of ROrn and *erase*, followed by explanations of the four cases.

$$
\begin{aligned}
&\textbf{data}\ \mathsf{ROrn}\ \{I\ J : \mathsf{Set}\}\ (e : J \to I) : \mathsf{RDesc}\ I \to \mathsf{RDesc}\ J \to \mathsf{Set}_1\ \textbf{where} \\
&\quad \mathsf{v} \ : \ \{js : \mathsf{List}\ J\}\ \{is : \mathsf{List}\ I\}\ (eqs : \mathbb{E}\ e\ js\ is) \to \mathsf{ROrn}\ e\ (\mathsf{v}\ is)\ (\mathsf{v}\ js) \\
&\quad \sigma \ : \ (S : \mathsf{Set})\ \{D : S \to \mathsf{RDesc}\ I\}\ \{E : S \to \mathsf{RDesc}\ J\} \\
&\qquad (O : (s : S) \to \mathsf{ROrn}\ e\ (D\ s)\ (E\ s)) \to \mathsf{ROrn}\ e\ (\sigma\ S\ D)\ (\sigma\ S\ E)
\end{aligned}
$$

```
  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

erase : {I J : Set} {e : J → I} {D : RDesc I} {E : RDesc J} →
        ROrn e D E → {X : I → Set} → ⟦ E ⟧ (X ∘ e) → ⟦ D ⟧ X
erase (v []       ) ▪       = ▪
erase (v (refl :: eqs)) (x , xs) = x , erase (v eqs) xs   -- x retained
erase (σ S O)       (s , xs) = s , erase (O s)   xs   -- s copied
erase (Δ T O)       (t , xs) =     erase (O t)   xs   -- t discarded
erase (∇ s O)          xs    = s , erase O      xs   -- s inserted
```

The first two cases v and σ of ROrn relate response descriptions that have the same top-level constructor, and the transformations decoded from them preserve horizontal structure.

- The v case of ROrn states that a response description v *js* refines another response description v *is*, i.e., when ⟦ v *js* ⟧ (*X* ∘ *e*) can be transformed into ⟦ v *is* ⟧ *X*. The source type ⟦ v *js* ⟧ (*X* ∘ *e*) expands to a product of types of the form *X* (*e j*) for some *j* : *J* and the target type ⟦ v *is* ⟧ *X* to a product of types of the form *X i* for some *i* : *I*. There are no horizontal contents and thus no horizontal modifications to make, and the input values should be preserved. We thus demand that *js* and *is* have the same number of elements and the corresponding pairs of indices *e j* and *i* are equal; that is, we demand a proof of *map e js* ≡ *is* (where *map* is the usual functorial mapping on lists). To make it easier to analyse a proof of *map e js* ≡ *is* in the v case of *erase*, we instead define the proposition inductively as 𝔼 *e js is*, where the datatype 𝔼 is defined by

  ```
  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)
  ```

- The σ case of ROrn states that σ *S E* refines σ *S D*, i.e., that both response

descriptions start with the same field of type S. The intended semantics — the σ case of *erase* — is to preserve (copy) the value of this field. To be able to transform the rest of the input structure, we should demand that, for any value $s : S$ of the field, the remaining response description $E\ s$ refines the other remaining response description $D\ s$.

The other two cases Δ and ∇ of ROrn deal with mismatching fields in the two response descriptions being related and prompt *erase* to perform nontrivial horizontal transformations.

- The Δ case of ROrn states that $\sigma\ T\ E$ refines D, the former having an additional field of type T whose value is not retained — the Δ case of *erase* discards the value of this field. We still need to transform the rest of the input structure, so the Δ constructor demands that, for every possible value $t : T$ of the field, the response description D is refined by the remaining response description $E\ t$.
- Conversely, the ∇ case of ROrn states that E refines $\sigma\ S\ D$, the latter having an additional field of type S. The value of this field needs to be restored by the ∇ case of *erase*, so the ∇ constructor demands a default value $s : S$ for the field. To be able to continue with the transformation, the ∇ constructor also demands that the response description E refines the remaining response description $D\ s$.

Convention. Again we regard Δ as a binder and write $\Delta[t : T]\ O\ t$ for $\Delta\ T\ (\lambda t \mapsto O\ t)$. Also, even though ∇ is not a binder, we write $\nabla[s]\ O$ for $\nabla\ s\ O$ to save the parentheses around O when O is a complex expression. □

Example (*ornament from natural numbers to lists*)**.** For any A : Set, there is an ornament from the description *NatD* of natural numbers to the description *ListD A* of lists:

```
NatD-ListD A : Orn ! NatD (ListD A)
NatD-ListD A (ok ▪) = σ ListTag λ { 'nil  ↦ v []
                                  ; 'cons ↦ Δ[ _ : A ] v (refl :: []) }
```

There is only one response ornament in *NatD-ListD A* since the datatype of lists is trivially indexed. The constructor tag is preserved (σ ListTag), and, in

the cons case, the list element field is marked as additional by Δ. Consequently, the forgetful function

$$\mathit{forget}\ (\mathit{NatD\text{-}ListD}\ A)\ \{\blacksquare\}\ :\ \mathsf{List}\ A \to \mathsf{Nat}$$

discards all list elements from a list and returns its underlying natural number, i.e., its length. □

Example (*ornament from lists to vectors*)**.** Again for any A : Set, there is an ornament from the description *ListD A* of lists to the description *VecD A* of vectors:

$$\begin{aligned}
&\mathit{ListD\text{-}VecD}\ A\ :\ \mathsf{Orn}\ !\ (\mathit{ListD}\ A)\ (\mathit{VecD}\ A)\\
&\mathit{ListD\text{-}VecD}\ A\ (\mathsf{ok\ zero}\quad)\ =\ \nabla[\,\text{`}\mathsf{nil}\,]\quad \mathsf{v}\ [\,]\\
&\mathit{ListD\text{-}VecD}\ A\ (\mathsf{ok}\ (\mathsf{suc}\ n))\ =\ \nabla[\,\text{`}\mathsf{cons}\,]\ \sigma[\,_ : A\,]\ \mathsf{v}\ (\mathsf{refl} :: [\,])\ \}
\end{aligned}$$

The response ornaments are indexed by Nat, since Nat is the index set of the datatype of vectors. We do pattern matching on the index request, resulting in two cases. In both cases, the constructor tag field exists for lists but not for vectors (since the constructor choice for vectors is determined from the index), so ∇ is used to insert the appropriate tag; in the suc case, the list element field is preserved by σ. Consequently, the forgetful function

$$\mathit{forget}\ (\mathit{ListD\text{-}VecD}\ A)\ :\ \{n\ :\ \mathsf{Nat}\} \to \mathsf{Vec}\ A\ n \to \mathsf{List}\ A$$

computes the underlying list of a vector. □

Remark (*vertical invariance of ornamental relationship*)**.** It is worth emphasising again that ornaments encode only horizontal transformations, so datatypes related by ornaments necessarily have the same recursion patterns (as enforced by the v constructor) — ornamental relationship exists between list-like datatypes but not between lists and binary trees, for example. □

3.2.2 Ornamental descriptions

There is apparent similarity between, e.g., the description *ListD A* and the ornament *NatD-ListD A*, which is typical: frequently we define a new description (e.g. *ListD A*), intending it to be a more refined version of an existing one (e.g.,

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

und-ℙ : {I J : Set} {e : J → I} (is : List I) → ℙ is (InvImage e) → List J
und-ℙ []       ▪       = []
und-ℙ (i :: is) (j , js) = und j :: und-ℙ is js

toRDesc : {I J : Set} {e : J → I} {D : RDesc I} → ROrnDesc J e D → RDesc J
toRDesc (∨ {is} js) = ∨ (und-ℙ is js)
toRDesc (σ S OD)    = σ[ s : S ] toRDesc (OD s)
toRDesc (Δ T OD)    = σ[ t : T ] toRDesc (OD t)
toRDesc (∇ s OD)    = toRDesc OD

toEq-ℙ : {I J : Set} {e : J → I} (is : List I) (js : ℙ is (InvImage e)) → 𝔼 e (und-ℙ is js) is
toEq-ℙ []       ▪       = []
toEq-ℙ (i :: is) (j , js) = toEq j :: toEq-ℙ is js

toROrn : {I J : Set} {e : J → I} {D : RDesc I} →
         (OD : ROrnDesc J e D) → ROrn e D (toRDesc OD)
toROrn (∨ js)    = ∨ (toEq-ℙ _ js)
toROrn (σ S OD)  = σ[ s : S ] toROrn (OD s)
toROrn (Δ T OD)  = Δ[ t : T ] toROrn (OD t)
toROrn (∇ s OD)  = ∇[ s ] (toROrn OD)

OrnDesc : {I : Set} (J : Set) (e : J → I) (D : Desc I) → Set₁
OrnDesc J e D = {i : I} (j : e ⁻¹ i) → ROrnDesc J e (D i)

⌊_⌋ : {I J : Set} {e : J → I} {D : Desc I} → OrnDesc J e D → Desc J
⌊ OD ⌋ j = toRDesc (OD (ok j))

⌈_⌉ : {I J : Set} {e : J → I} {D : Desc I} (OD : OrnDesc J e D) → Orn e D ⌊ OD ⌋
⌈ OD ⌉ (ok j) = toROrn (OD (ok j))
```

Figure 3.2 Definitions for ornamental descriptions.

NatD), and then immediately write an ornament from the latter to the former (e.g., *NatD-ListD A*). The syntactic structures of the new description and of the ornament are essentially the same, however, so the effort is duplicated. It would be more efficient if we could use the existing description as a template and just write a "relative description" specifying how to "patch" the template, and afterwards from this "relative description" extract a new description and an ornament from the template to the new description.

Ornamental descriptions are designed for this purpose; the definitions are shown in Figure 3.2 and closely follow the definitions for ornaments, having a upper-level type OrnDesc of ornamental descriptions which refers to a lower-level datatype ROrnDesc of response ornamental descriptions. An ornamental description looks like an annotated description, on which we can use a greater variety of constructors to mark differences from the template description. We think of an ornamental description

$$OD \;:\; \mathsf{OrnDesc}\ J\ e\ D$$

as simultaneously denoting a new description of type $\mathsf{Desc}\ J$ and an ornament from the template description D to the new description, and use floor and ceiling brackets $\lfloor \_ \rfloor$ and $\lceil \_ \rceil$ to resolve ambiguity: the new description is

$$\lfloor OD \rfloor \;:\; \mathsf{Desc}\ J$$

and the ornament is

$$\lceil OD \rceil \;:\; \mathsf{Orn}\ e\ D\ \lfloor OD \rfloor$$

Example (*ordered lists as an ornamentation of lists*)**.** Given $A \;:\; \mathsf{Set}$ with an ordering relation $\_\leqslant_A\_ \;:\; A \to A \to \mathsf{Set}$, we can define ordered lists on A by an ornamental description, using the description of lists as the template:

$$\begin{aligned}
&\mathit{OrdListOD}\ A\ _\leqslant_A_ \;:\; \mathsf{OrnDesc}\ A\ !\ (\mathit{ListD}\ A)\\
&\mathit{OrdListOD}\ A\ _\leqslant_A_\ (\mathsf{ok}\ b)\ =\\
&\quad \sigma\ \mathsf{ListTag}\ \lambda\ \{\ \text{`nil} \quad\ \mapsto\ \mathsf{v}\ \blacksquare\\
&\qquad\qquad ;\ \text{`cons} \quad\ \mapsto\ \sigma[\,a : A\,]\ \Delta[\,\mathit{leq} : b \leqslant_A a\,]\ \mathsf{v}\ (a\ ,\ \blacksquare)\ \}\\
&\textbf{indexfirst data}\ \mathsf{OrdList}\ A\ _\leqslant_A_ \;:\; A \to \mathsf{Set}\ \textbf{where}
\end{aligned}$$

$$\begin{aligned}&\mathsf{OrdList}\ A\ \_{\leqslant_A}\_\ b\ \ni\ \mathsf{nil}\\ &\qquad\qquad\qquad\qquad\mid\ \mathsf{cons}\ (a\ :\ A)\ (\mathit{leq}\ :\ b \leqslant_A a)\ (\mathit{as}\ :\ \mathsf{OrdList}\ A\ \_{\leqslant_A}\_\ a)\end{aligned}$$

If we read $\mathit{OrdListOD}\ A\ \_{\leqslant_A}\_$ as an annotated description, we can think of the *leq* field as being marked as additional (relative to the description of lists) by using Δ rather than σ. To decode $\mathit{OrdListOD}\ A\ \_{\leqslant_A}\_$ to an ordinary description of ordered lists, we write

$$\lfloor \mathit{OrdListOD}\ A\ \_{\leqslant_A}\_ \rfloor\ :\ \mathsf{Desc}\ A$$

and

$$\lceil \mathit{OrdListOD}\ A\ \_{\leqslant_A}\_ \rceil\ :\ \mathsf{Orn}\ !\ (\mathit{ListD}\ A)\ \lfloor \mathit{OrdListOD}\ A\ \_{\leqslant_A}\_ \rfloor$$

is an ornament from lists to ordered lists. □

Example (*singleton ornamentation*)**.** Consider the following **singleton datatype** for lists:

$$\begin{aligned}&\textbf{indexfirst data}\ \mathsf{ListS}\ A\ :\ \mathsf{List}\ A \to \mathsf{Set}\ \textbf{where}\\ &\quad \mathsf{ListS}\ A\ []\ \ni\ \mathsf{nil}\\ &\quad \mathsf{ListS}\ A\ (x :: xs)\ \ni\ \mathsf{cons}\ (s\ :\ \mathsf{ListS}\ A\ xs)\end{aligned}$$

For each type $\mathsf{ListS}\ A\ xs$, there is exactly one (canonical) inhabitant (hence the name "singleton datatype" [Monnier and Haguenauer, 2010]), which has the same vertical structure as xs and is devoid of any horizontal contents. We can encode the datatype as an ornamental description relative to $\mathit{ListD}\ A$:

$$\begin{aligned}&\mathit{ListSOD}\ :\ (A\ :\ \mathsf{Set}) \to \mathsf{OrnDesc}\ (\mathsf{List}\ A)\ !\ (\mathit{ListD}\ A)\\ &\mathit{ListSOD}\ A\ (\mathsf{ok}\ [])\ =\ \nabla[\,\text{`}\mathsf{nil}\,]\ \mathsf{v}\ \blacksquare\\ &\mathit{ListSOD}\ A\ (\mathsf{ok}\ (x :: xs))\ =\ \nabla[\,\text{`}\mathsf{cons}\,]\ \nabla[\,x\,]\ \mathsf{v}\ (\mathsf{ok}\ xs\ ,\ \blacksquare)\end{aligned}$$

which does pattern matching on the index request, in each case restricts the constructor choice to the one matched against, and in the cons case deletes the element field and sets the index of the recursive position to be the value of the tail in the pattern. In general, we can define a parametrised ornamental description

$$\mathit{singletonOD}\ :\ \{I\ :\ \mathsf{Set}\}\ (D\ :\ \mathsf{Desc}\ I) \to \mathsf{OrnDesc}\ (\Sigma\ I\ (\mu\ D))\ \mathsf{outl}\ D$$

called the **singleton ornamental description**, which delivers a singleton datatype as an ornamentation of any datatype. The complete definition is

$$
\begin{aligned}
&\mathit{erode} : \{I : \mathsf{Set}\}\ (D : \mathsf{RDesc}\ I)\ \{J : I \to \mathsf{Set}\} \to \\
&\qquad [\![\ D\ ]\!]\ J \to \mathsf{ROrnDesc}\ (\Sigma\ I\ J)\ \mathsf{outl}\ D \\
&\mathit{erode}\ (\mathsf{v}\ \mathit{is})\quad \mathit{js} \quad = \mathsf{v}\ (\mathbb{P}\text{-}\mathit{map}\ (\lambda\ \{i\}\ j \mapsto \mathsf{ok}\ (i\ ,\ j))\ \mathit{is}\ \mathit{js}) \\
&\mathit{erode}\ (\sigma\ S\ D)\ (s\ ,\ \mathit{js}) = \nabla[\,s\,]\ \mathit{erode}\ (D\ s)\ \mathit{js} \\
&\mathit{singletonOD} : \{I : \mathsf{Set}\}\ (D : \mathsf{Desc}\ I) \to \mathsf{OrnDesc}\ (\Sigma\ I\ (\mu\ D))\ \mathsf{outl}\ D \\
&\mathit{singletonOD}\ D\ (\mathsf{ok}\ (i\ ,\ \mathsf{con}\ \mathit{ds})) = \mathit{erode}\ (D\ i)\ \mathit{ds}
\end{aligned}
$$

where

$$
\begin{aligned}
&\mathbb{P}\text{-}\mathit{map} : \{I : \mathsf{Set}\}\ \{X\ Y : I \to \mathsf{Set}\} \to (X \rightrightarrows Y) \to \\
&\qquad (\mathit{is} : \mathsf{List}\ I) \to \mathbb{P}\ \mathit{is}\ X \to \mathbb{P}\ \mathit{is}\ Y \\
&\mathbb{P}\text{-}\mathit{map}\ f\ [\,] \qquad \blacksquare \qquad = \blacksquare \\
&\mathbb{P}\text{-}\mathit{map}\ f\ (i :: \mathit{is})\ (x\ ,\ \mathit{xs}) = f\ x\ ,\ \mathbb{P}\text{-}\mathit{map}\ f\ \mathit{is}\ \mathit{xs}
\end{aligned}
$$

Note that *erode* deletes all fields (i.e., horizontal contents), drawing default values from the index request, and retains only the vertical structure. We will see in Section 3.3 that singleton ornamentation plays a key role in the ornament–refinement framework. □

Remark (*ornaments as relations*)**.** We define ornaments as relations between descriptions (indexed with an erasure function), whereas the original ornaments [McBride, 2011; Dagand and McBride, 2012] are rebranded as ornamental descriptions. One obvious advantage of relational ornaments is that they can arise between existing descriptions, whereas ornamental descriptions always produce (definitionally) new descriptions at the more informative end. A consequence is that there can be multiple ornaments between a pair of descriptions. For example, consider the following description of a datatype consisting of two fields of the same type:

$$
\begin{aligned}
&\mathit{SquareD} : (A : \mathsf{Set}) \to \mathsf{Desc}\ \top \\
&\mathit{SquareD}\ A\ \blacksquare = \sigma[\,\_ : A\,]\ \sigma[\,\_ : A\,]\ \mathsf{v}\ [\,]
\end{aligned}
$$

Between *SquareD A* and itself, we have the identity ornament

$$\lambda\ \{\ \blacksquare \mapsto\ \sigma[\,\_ : A\,]\ \sigma[\,\_ : A\,]\ \mathsf{v}\ [\,]\ \}$$

and the "swapping" ornament

$$\lambda\ \{\ \blacksquare \mapsto\ \Delta[\,x : A\,]\ \Delta[\,y : A\,]\ \nabla[\,y\,]\ \nabla[\,x\,]\ \mathsf{v}\ [\,]\ \}$$

whose forgetful function swaps the two fields.

The other advantage of relational ornaments is that they allow new datatypes to arise at the less informative end. For example, **coproduct of signatures** as used in, e.g., data types à la carte [Swierstra, 2008], can be implemented naturally with relational ornaments but not with ornamental descriptions. In more detail: Consider (a simplistic version of) **tagged descriptions** [Chapman et al., 2010], which are descriptions that, for any index request, always respond with a constructor field first. A tagged description with index set $I : \mathsf{Set}$ thus consists of a family of types $C : I \to \mathsf{Set}$, where each $C\ i$ is the set of constructor tags for the index request $i : I$, and a family of subsequent response descriptions for each constructor tag.

$$\begin{aligned}&\mathsf{TDesc} : \mathsf{Set} \to \mathsf{Set}_1\\&\mathsf{TDesc}\ I\ =\ \Sigma[\,C : I \to \mathsf{Set}\,]\ ((i\ :\ I) \to C\ i \to \mathsf{RDesc}\ I)\end{aligned}$$

Tagged descriptions are decoded to ordinary descriptions by

$$\begin{aligned}&\lfloor\_\rfloor_T\ :\ \{I\ :\ \mathsf{Set}\} \to \mathsf{TDesc}\ I \to \mathsf{Desc}\ I\\&\lfloor C\ ,\ D\rfloor_T\ i\ =\ \sigma\ (C\ i)\ (D\ i)\end{aligned}$$

We can then define binary coproduct of tagged descriptions, which sums the corresponding constructor fields, as follows:

coproduct-related definitions

$$\begin{aligned}&\_\oplus\_\ :\ \{I\ :\ \mathsf{Set}\} \to \mathsf{TDesc}\ I \to \mathsf{TDesc}\ I \to \mathsf{TDesc}\ I\\&(C\ ,\ D) \oplus (C'\ ,\ D')\ =\ (\lambda\, i \mapsto C\ i + C'\ i)\ ,\ (\lambda\, i \mapsto D\ i \triangledown D'\ i)\end{aligned}$$

Now given two tagged descriptions $tD\ =\ (C\ ,\ D)$ and $tD'\ =\ (C'\ ,\ D')$ of type $\mathsf{TDesc}\ I$, there are two ornaments from $\lfloor tD \oplus tD'\rfloor_T$ to $\lfloor tD\rfloor_T$ and $\lfloor tD'\rfloor_T$

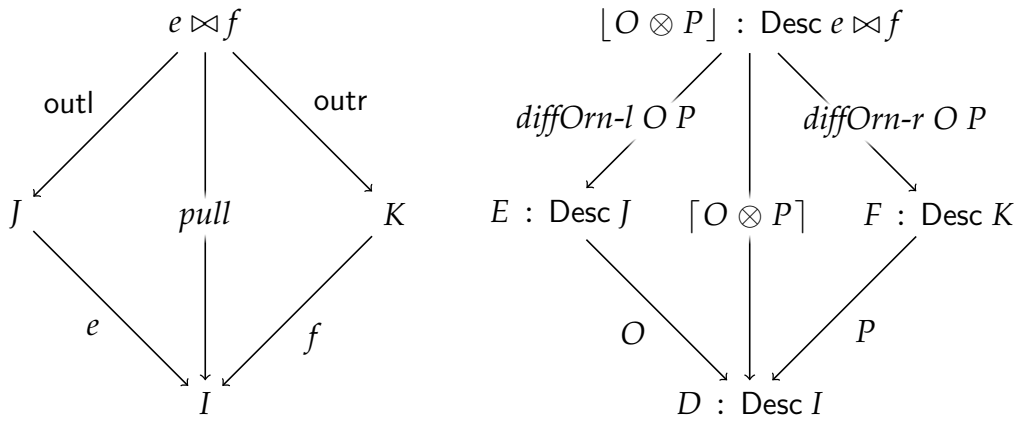
$$\begin{aligned}&\mathit{inlOrn}\ :\ \mathsf{Orn}\ \mathit{id}\ \lfloor tD \oplus tD'\rfloor_T\ \lfloor tD\rfloor_T\\&\mathit{inlOrn}\ (\mathsf{ok}\ i)\ =\ \Delta[\,c\ :C\ i\,]\ \ \nabla[\,\mathsf{inl}\ c\,]\ \ \mathit{idOrn}\ (D\ i\ c)\\&\mathit{inrOrn}\ :\ \mathsf{Orn}\ \mathit{id}\ \lfloor tD \oplus tD'\rfloor_T\ \lfloor tD'\rfloor_T\\&\mathit{inrOrn}\ (\mathsf{ok}\ i)\ =\ \Delta[\,c' : C'\ i\,]\ \nabla[\,\mathsf{inr}\ c'\,]\ \mathit{idOrn}\ (D'\ i\ c')\end{aligned}$$

whose forgetful functions perform suitable injection of constructor tags. Note that the synthesised new description $\lfloor\ tD \oplus tD'\ \rfloor_T$ is at the less informative end of *inlOrn* and *inrOrn*. (This, of course, is not a complete implementation of data types à la carte and requires more engineering for practical use.) □

3.2.3 Parallel composition of ornaments

intro — analysis for composability

The generic scenario is illustrated below: ??



Given three descriptions D : Desc I, E : Desc J, and F : Desc K and two ornaments O : Orn e D E and P : Orn e D F independently specifying how D is refined to E and F, we can compute an ornamental description

$$O \otimes P \ : \ \mathsf{OrnDesc}\ (e \bowtie f)\ \mathit{pull}\ D$$

Intuitively, since both O and P encode modifications to the same base description D, we can commit all modifications encoded by O and P to D to get a new description $\lfloor O \otimes P \rfloor$, and encode all these modifications in one ornament $\lceil O \otimes P \rceil$. (This merging of two sets of modifications is best characterised by a category-theoretic pullback, which we defer until **??**.) The forgetful function of the ornament $\lceil O \otimes P \rceil$ removes all modifications, taking $\mu\ \lfloor O \otimes P \rfloor$ all the way back to the base datatype $\mu\ D$; there are also two **difference ornaments**

$$\begin{array}{ll} \mathit{diffOrn\text{-}l}\ O\ P \ : \ \mathsf{Orn}\ \mathsf{outl}\ E\ \lfloor O \otimes P \rfloor & \text{-- left difference ornament} \\ \mathit{diffOrn\text{-}r}\ O\ P \ : \ \mathsf{Orn}\ \mathsf{outr}\ F\ \lfloor O \otimes P \rfloor & \text{-- right difference ornament} \end{array}$$

which give rise to "less forgetful" functions taking $\mu\ \lfloor O \otimes P \rfloor$ to $\mu\ E$ and $\mu\ F$, such that both

$$\mathit{forget}\ O \circ \mathit{forget}\ (\mathit{diffOrn\text{-}l}\ O\ P)$$

and

$$\mathit{forget}\ P \circ \mathit{forget}\ (\mathit{diffOrn\text{-}r}\ O\ P)$$

are extensionally equal to $\mathit{forget}\ \lceil O \otimes P \rceil$.

Example (*ordered vectors*)**.** Consider the two ornaments $\lceil \mathit{OrdListOD}\ A\ \_\leqslant_A\_ \rceil$ from lists to ordered lists and $\mathit{ListD\text{-}VecD}\ A$ from lists to vectors. Composing them in parallel gives us an ornamental description from which we can decode (i) a new datatype of ordered vectors

$$\begin{aligned}
&\mathsf{OrdVec}\ A\ _\leqslant_A_ : A \to \mathsf{Nat} \to \mathsf{Set} \\
&\mathsf{OrdVec}\ A\ _\leqslant_A_\ b\ n\ = \\
&\quad \mu\ \lfloor \lceil \mathit{OrdListOD}\ A\ _\leqslant_A_ \rceil \otimes \mathit{ListD\text{-}VecD}\ A \rfloor\ (\mathsf{ok}\ (\mathsf{ok}\ b\ ,\ \mathsf{ok}\ n)) \\
&\textbf{indexfirst data}\ \mathsf{OrdVec}\ A\ _\leqslant_A_ : A \to \mathsf{Nat} \to \mathsf{Set}\ \textbf{where} \\
&\quad \mathsf{OrdVec}\ A\ _\leqslant_A_\ b\ \mathsf{zero} \quad \ni\ \mathsf{nil} \\
&\quad \mathsf{OrdVec}\ A\ _\leqslant_A_\ b\ (\mathsf{suc}\ n) \ \ni\ \mathsf{cons}\ (a\ :\ A)\ (\mathit{leq}\ :\ b \leqslant_A a) \\
&\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{as}\ :\ \mathsf{OrdVec}\ A\ _\leqslant_A_\ a\ n)
\end{aligned}$$

and (ii) an ornament whose forgetful function converts ordered vectors to plain lists, retaining the list elements. The forgetful functions of the difference ornaments convert ordered vectors to ordered lists and vectors, removing only length and ordering information respectively. □

The complete definitions for parallel composition are shown in Figure 3.3. The core definition is *pcROD*, which analyses and merges the modifications encoded by two response ornaments into a response ornamental description at the level of individual fields. Below are some representative cases of *pcROD*.

- When both response ornaments use σ, both of them preserve the same field in the base description — no modification is made. Consequently, the field is preserved in the resulting response ornamental description as well.

  $$\mathit{pcROD}\ (\sigma\ S\ O)\ (\sigma\ .S\ P)\ =\ \sigma[\,s : S\,]\ \mathit{pcROD}\ (O\ s)\ (P\ s)$$

- When one of the response ornaments uses Δ to mark the addition of a new field, that field would be added into the resulting response ornamental description, like in

  $$\mathit{pcROD}\ (\Delta\ T\ O)\ P\ =\ \Delta[\,t : T\,]\ \mathit{pcROD}\ (O\ t)\ P$$

```
pc-𝔼 : {I J K : Set} {e : J → I} {f : K → I} →
       {is : List I} {js : List J} {ks : List K} →
       𝔼 e js is → 𝔼 f ks is → ℙ is (InvImage pull)
pc-𝔼          []          []          = ▪
pc-𝔼 {e := e} {f} (eeq :: eeqs) (feq :: feqs) = ok (fromEq e eeq , fromEq f feq) ,
                                                pc-𝔼 eeqs feqs

mutual
  pcROD : {I J K : Set} {e : J → I} {f : K → I}
          {D : RDesc I} {E : RDesc J} {F : RDesc K} →
          ROrn e D E → ROrn f D F → ROrnDesc (e ⋈ f) pull D
  pcROD (v eeqs) (v feqs)   = v (pc-𝔼 eeqs feqs)
  pcROD (v eeqs) (Δ T P)    = Δ[ t : T ] pcROD (v eeqs) (P t)
  pcROD (σ S O) (σ .S P)    = σ[ s : S ] pcROD (O s)    (P s)
  pcROD (σ f O) (Δ T P)     = Δ[ t : T ] pcROD (σ f O) (P t)
  pcROD (σ S O) (∇ s P)     = ∇[ s ]     pcROD (O s)    P
  pcROD (Δ T O) P           = Δ[ t : T ] pcROD (O t)    P
  pcROD (∇ s O) (σ S P)     = ∇[ s ]     pcROD O        (P s)
  pcROD (∇ s O) (Δ T P)     = Δ[ t : T ] pcROD (∇ s O) (P t)
  pcROD (∇ s O) (∇ s′ P)    = Δ (s ≡ s′) (pcROD-double∇ O P)

  pcROD-double∇ :
    {I J K S : Set} {e : J → I} {f : K → I}
    {D : S → RDesc I} {E : RDesc J} {F : RDesc K} {s s′ : S} →
    ROrn e (D s) E → ROrn f (D s′) F →
    s ≡ s′ → ROrnDesc (e ⋈ f) pull (σ S D)
  pcROD-double∇ {s := s} O P refl = ∇[ s ] pcROD O P

_⊗_ : {I J K : Set} {e : J → I} {f : K → I}
      {D : Desc I} {E : Desc J} {F : Desc K} →
      Orn e D E → Orn f D F → OrnDesc (e ⋈ f) pull D
(O ⊗ P) (ok (j , k)) = pcROD (O j) (P k)
```

Figure 3.3 Definitions for parallel composition of ornaments.

- If one of the response ornaments retains a field by σ and the other deletes it by ∇, the only modification to the field is deletion, and thus the field is deleted in the resulting response ornamental description, like in

  $$\mathit{pcROD}\ (\sigma\ S\ O)\ (\nabla\ s\ P)\ =\ \nabla[\,s\,]\ \mathit{pcROD}\ (O\ s)\ P$$

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

  $$\mathit{pcROD}\ (\nabla\ s\ O)\ (\nabla\ s'\ P)\ =\ \Delta\ (s\ \equiv\ s')\ (\mathit{pcROD\text{-}double}\nabla\ O\ P)$$

  and then *pcROD-double*∇ puts the deletion into the resulting response ornamental description after matching the proof of the equality field with refl.

  $$\mathit{pcROD\text{-}double}\nabla\ \{s := s\}\ O\ P\ \mathsf{refl}\ =\ \nabla[\,s\,]\ \mathit{pcROD}\ O\ P$$

  It might seem bizarre that two deletions results in a new field (and a deletion), but consider this informally described scenario: A field $\sigma\ S$ in the base response description is refined by two independent response ornaments

  $$\Delta[\,t : T\,]\quad \nabla[\,g\ t\,]$$

  and

  $$\Delta[\,u : U\,]\ \nabla[\,h\ u\,]$$

  That is, instead of S-values, the response descriptions at the more informative end of the two response ornaments use T- and U-values at this position, which are erased to their underlying S-value by $g\ :\ T \to S$ and $h\ :\ U \to S$ respectively. Composing the two response ornaments in parallel, we get

  $$\Delta[\,t : T\,]\ \Delta[\,u : U\,]\ \Delta[\,\_ : g\ t\ \equiv\ h\ u\,]\ \nabla[\,g\ t\,]$$

  where the added equality field completes the construction of a set-theoretic pullback of g and h. Here indeed we need a pullback: When we have an actual value for the field $\sigma\ S$, which gets refined to values of types T and U, the generic way to mix the two refining values is to store them both, as a product. If we wish to retrieve the underlying value of type S, we can either extract the value of type T and apply g to it or extract the value of type U and

apply h to it, and through either path we should get the same underlying value. So the product should really be a pullback to ensure this. ??

Example (*ornamental description of ordered vectors*)**.** Composing the ornaments $\lceil$ *OrdListOD* A $\_\leqslant_A\_$ $\rceil$ and *ListD-VecD* A in parallel yields the following ornamental description relative to *ListD* A:

$$\begin{aligned}\lambda\,\{\ &(\mathsf{ok}\ (\mathsf{ok}\ b\ ,\ \mathsf{ok}\ \mathsf{zero}\quad))\ \mapsto\ \nabla[\,\text{`}\mathsf{nil}\,]\ \mathsf{v}\ \blacksquare\\ ;\ &(\mathsf{ok}\ (\mathsf{ok}\ b\ ,\ \mathsf{ok}\ (\mathsf{suc}\ n)))\ \mapsto\ \nabla[\,\text{`}\mathsf{cons}\,]\ \sigma[\,a : A\,]\\ &\qquad\qquad\qquad\qquad\Delta[\,\_ : b \leqslant_A a\,]\ \mathsf{v}\ (\mathsf{ok}\ (\mathsf{ok}\ a\ ,\ \mathsf{ok}\ n)\ ,\ \blacksquare)\ \}\end{aligned}$$

where lighter box indicates modifications from $\lceil$ *OrdListOD* A $\_\leqslant_A\_$ $\rceil$ and darker box from *ListD-VecD* A. $\square$

Finally, the definitions for left difference ornament are shown in Figure 3.4. Left difference ornament has the same structure as parallel composition, but records only modifications from the right-hand side ornament. For example, the case

$$\mathit{diffROrn\text{-}l}\ (\sigma\ S\ O)\ (\nabla\ s\ P)\ =\ \nabla[\,s\,]\ \mathit{diffROrn\text{-}l}\ (O\ s)\ P$$

is the same as the corresponding case of *pcROD*, since the deletion comes from the right-hand side response ornament, whereas the case

$$\mathit{diffROrn\text{-}l}\ (\Delta\ T\ O)\ P\ =\ \sigma[\,t : T\,]\ \mathit{diffROrn\text{-}l}\ (O\ t)\ P$$

produces σ (a preservation) rather than Δ (a modification) as in the corresponding case of *pcROD*, since the addition comes from the left-hand side response ornament. We can then see that the composition of the forgetful functions

$$\mathit{forget}\ O \circ \mathit{forget}\ (\mathit{diffOrn\text{-}l}\ O\ P)$$

is indeed extensionally equal to $\mathit{forget}\ \lceil\ O \otimes P\ \rceil$, since $\mathit{forget}\ (\mathit{diffOrn\text{-}l}\ O\ P)$ removes modifications encoded in the right-hand side ornament and then $\mathit{forget}\ O$ removes modifications encoded in the left-hand side ornament. Right difference ornament is defined analogously and is omitted from the presentation.

```
diff-𝔼-l : {I J K : Set} {e : J → I} {f : K → I} →
           {is : List I} {js : List J} {ks : List K} →
           (eeqs : 𝔼 e js is) (feqs : 𝔼 f ks is) → 𝔼 outl (und-ℙ is (pc-𝔼 eeqs feqs)) js
diff-𝔼-l          []          []          = []
diff-𝔼-l {e := e} (eeq :: eeqs) (feq :: feqs) = und-fromEq e eeq :: diff-𝔼-l eeqs feqs

mutual
  diffROrn-l :
    {I J K : Set} {e : J → I} {f : K → I} →
    {D : RDesc I} {E : RDesc J} {F : RDesc K} →
    (O : ROrn e D E) (P : ROrn f D F) → ROrn outl E (toRDesc (pcROD O P))
  diffROrn-l (v eeqs) (v feqs)   = v (diff-𝔼-l eeqs feqs)
  diffROrn-l (v eeqs) (Δ T P)    = Δ[ t : T ] diffROrn-l (v eeqs) (P t)
  diffROrn-l (σ S O) (σ .S P)    = σ[ s : S ] diffROrn-l (O s)    (P s)
  diffROrn-l (σ S O) (Δ T P)     = Δ[ t : T ] diffROrn-l (σ S O) (P t)
  diffROrn-l (σ S O) (∇ s P)     = ∇[ s ]     diffROrn-l (O s)    P
  diffROrn-l (Δ T O) P           = σ[ t : T ] diffROrn-l (O t)    P
  diffROrn-l (∇ s O) (σ S P)     =            diffROrn-l O        (P s)
  diffROrn-l (∇ s O) (Δ T P)     = Δ[ t : T ] diffROrn-l (∇ s O) (P t)
  diffROrn-l (∇ s O) (∇ s′ P)    = Δ (s ≡ s′) (diffROrn-l-double∇ O P)
  diffROrn-l-double∇ :
    {I J K S : Set} {e : J → I} {f : K → I} →
    {D : S → RDesc I} {E : RDesc J} {F : RDesc K} {s s′ : S} →
    (O : ROrn e (D s) E) (P : ROrn f (D s′) F) (eq : s ≡ s′) →
    ROrn outl E (toRDesc (pcROD-double∇ O P eq))
  diffROrn-l-double∇ O P refl = diffROrn-l O P

diffOrn-l : {I J K : Set} {e : J → I} {f : K → I} →
            {D : Desc I} {E : Desc J} {F : Desc K} →
            (O : Orn e D E) (P : Orn f D F) → Orn outl E ⌊ O ⊗ P ⌋
diffOrn-l O P (ok (j , k)) = diffROrn-l (O j) (P k)
```

Figure 3.4 Definitions for left difference ornament.

3.3 Refinement semantics of ornaments

Every ornament O : Orn e D E induces a refinement family from μ D to μ E. That is, we can construct a function

$$\begin{aligned} RSem\ :\ &\{I\ J\ :\ \mathsf{Set}\}\ \{e\ :\ J \to I\}\ \{D\ :\ \mathsf{Desc}\ I\}\ \{E\ :\ \mathsf{Desc}\ J\} \to \\ &\mathsf{Orn}\ e\ D\ E \to \mathsf{FRefinement}\ e\ (\mu\ D)\ (\mu\ E) \end{aligned}$$

which is called the **refinement semantics** of ornaments.

intro

3.3.1 Optimised predicates

Our most important task for now is to construct a promotion predicate

$$\begin{aligned} \mathsf{OptP}\ :\ &\{I\ J\ :\ \mathsf{Set}\}\ \{e\ :\ J \to I\}\ \{D\ :\ \mathsf{Desc}\ I\}\ \{E\ :\ \mathsf{Desc}\ J\} \to \\ &(O\ :\ \mathsf{Orn}\ e\ D\ E)\ \{i\ :\ I\}\ (j\ :\ e^{-1}\ i)\ (x\ :\ \mu\ D\ i) \to \mathsf{Set} \end{aligned}$$

which is called the **optimised predicate** for the ornament O. Given x : μ D i, a proof of type OptP O j x contains the necessary information for complementing x and forming an inhabitant y of type μ E $(und\ j)$ with the same recursive structure — the proof is the "horizontal" difference between y and x, speaking in terms of the two-dimensional metaphor. Such a proof should have the same vertical structure as x, and, at each recursive node, store horizontally only those data marked as modified by the ornament. For example, if we are promoting the natural number

Optimised in what sense?

$$\begin{aligned} two\ =\ &\mathsf{con}\ (\text{`}\mathsf{cons}\ , \\ &\mathsf{con}\ (\text{`}\mathsf{cons}\ , \\ &\mathsf{con}\ (\text{`}\mathsf{nil}\quad , \\ &\quad \blacksquare)\ ,\ \blacksquare)\ ,\ \blacksquare)\ :\ \mu\ NatD\ \blacksquare \end{aligned}$$

to a list, an optimised promotion proof would look like

$$\begin{aligned} p\ =\ &\mathsf{con}\ (a\ , \\ &\mathsf{con}\ (a'\ , \end{aligned}$$

$$\begin{array}{l}\mathsf{con}\ (\\ \quad \blacksquare)\ ,\ \blacksquare)\ ,\ \blacksquare)\ :\ \mathsf{OptP}\ (\mathit{NatD\text{-}ListD}\ A)\ (\mathsf{ok}\ \blacksquare)\ \mathit{two}\end{array}$$

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

$$\begin{array}{l}\mathsf{con}\ (\text{`}\mathsf{cons}\ ,\ a\ \ ,\\ \mathsf{con}\ (\text{`}\mathsf{cons}\ ,\ a'\ ,\\ \mathsf{con}\ (\text{`}\mathsf{nil}\ \ \ ,\\ \quad \blacksquare)\ ,\ \blacksquare)\ ,\ \blacksquare)\ :\ \mu\ (\mathit{ListD}\ A)\ \blacksquare\end{array}$$

Note that p contains only values of the field marked as additional by Δ in the ornament *NatD-ListD A*. The constructor tags are essential for determining the recursive structure of p, but instead of being stored in p, they are derived from *two*, which is part of the index of the type of p. In general, here is how we compute an ornamental description for such proofs, using D as the template: we incorporate the modifications made by O, and delete the fields that already exist in D, whose default values are derived in the index-first fashion from the inhabitant being promoted, which appears in the index of the type of a proof. The deletion is independent of O and can be performed by the singleton ornament for D (Section 3.2.2), so the desired ornamental description is produced by the parallel composition of O and $\lceil \mathit{singletonOD}\ D \rceil$:

$$\begin{array}{l}\mathit{OptPOD}\ :\ \{I\ J\ :\ \mathsf{Set}\}\ \{e\ :\ J \to I\}\ \{D\ :\ \mathsf{Desc}\ I\}\ \{E\ :\ \mathsf{Desc}\ J\} \to \\ \qquad\qquad \mathsf{Orn}\ e\ D\ E \to \mathsf{OrnDesc}\ (e \bowtie \mathsf{outl})\ \mathit{pull}\ D \\ \mathit{OptPOD}\ \{D := D\}\ O\ =\ O \otimes \lceil \mathit{singletonOD}\ D \rceil\end{array}$$

where outl has type $\Sigma\ I\ (\mu\ D) \to I$. The optimised predicate, then, is the least fixed point of the description.

$$\begin{array}{l}\mathsf{OptP}\ :\ \{I\ J\ :\ \mathsf{Set}\}\ \{e\ :\ J \to I\}\ \{D\ :\ \mathsf{Desc}\ I\}\ \{E\ :\ \mathsf{Desc}\ J\} \to \\ \qquad (O\ :\ \mathsf{Orn}\ e\ D\ E)\ \{i\ :\ I\}\ (j\ :\ e^{-1}\ i)\ (x\ :\ \mu\ D\ i) \to \mathsf{Set} \\ \mathsf{OptP}\ O\ \{i\}\ j\ d\ =\ \mu\ \lfloor \mathit{OptPOD}\ O \rfloor\ (j\ ,\ (\mathsf{ok}\ (i\ ,\ d)))\end{array}$$

Example (*index-first vectors as an optimised predicate*)**.** The optimised predicate for the ornament *NatD-ListD A* from natural numbers to lists is the datatype of index-first vectors. Expanding the definition of the ornamental description

OptPOD (*NatD-ListD A*) relative to *NatD*:

$$\begin{aligned}\lambda\,\{\ &(\mathsf{ok}\ (\mathsf{ok}\ \blacksquare\ ,\ \mathsf{ok}\ (\blacksquare\ ,\ \mathsf{zero}\ )))\ \mapsto\ \nabla[\,\text{`nil}\,]\ \mathsf{v}\ \blacksquare\\ ;\ &(\mathsf{ok}\ (\mathsf{ok}\ \blacksquare\ ,\ \mathsf{ok}\ (\blacksquare\ ,\ \mathsf{suc}\ n)))\ \mapsto\ \nabla[\,\text{`cons}\,]\ \Delta[\,\_ : A\,]\\ &\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{v}\ (\mathsf{ok}\ (\mathsf{ok}\ \blacksquare\ ,\ \mathsf{ok}\ (\blacksquare\ ,\ n))\ ,\ \blacksquare)\ \}\end{aligned}$$

where lighter box indicates contributions from the ornament *NatD-ListD A* and darker box from the singleton ornament $\lceil$*singletonOD NatD*$\rceil$, we see that the ornamental description indeed yields the datatype of index-first vectors (albeit indexed by a more heavily packaged datatype of natural numbers). □

Example (*predicate characterising ordered lists*)**.** The optimised predicate for the ornament $\lceil$ *OrdListOD A* $\_\leqslant_A\_$ $\rceil$ from lists to ordered lists is given by the ornamental description *OptPOD* $\lceil$ *OrdListOD A* $\_\leqslant_A\_$ $\rceil$ relative to *ListD A*, which expands to

$$\begin{aligned}\lambda\,\{\ &(\mathsf{ok}\ (\mathsf{ok}\ b\ ,\ \mathsf{ok}\ (\blacksquare\ ,\ []\quad\ )))\ \mapsto\ \nabla[\,\text{`nil}\,]\ \mathsf{v}\ \blacksquare\\ ;\ &(\mathsf{ok}\ (\mathsf{ok}\ b\ ,\ \mathsf{ok}\ (\blacksquare\ ,\ a :: as)))\ \mapsto\ \nabla[\,\text{`cons}\,]\ \nabla[\,a\,]\ \Delta[\,leq : b \leqslant_A a\,]\\ &\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{v}\ (\mathsf{ok}\ (\mathsf{ok}\ a\ ,\ \mathsf{ok}\ (\blacksquare\ ,\ as))\ ,\ \blacksquare)\ \}\end{aligned}$$

where lighter box indicates contributions from $\lceil$ *OrdListOD A* $\_\leqslant_A\_$ $\rceil$ and darker box from $\lceil$*singletonOD* (*ListD A*)$\rceil$.

$$\begin{aligned}&\textbf{indexfirst data}\ \mathsf{Ordered}\ A\ \_\leqslant_A\_\ :\ A \to \mathsf{List}\ A \to \mathsf{Set}\ \textbf{where}\\ &\quad \mathsf{Ordered}\ A\ \_\leqslant_A\_\ b\ []\qquad\ \ni\ \mathsf{nil}\\ &\quad \mathsf{Ordered}\ A\ \_\leqslant_A\_\ b\ (a :: as)\ \ni\ \mathsf{cons}\ (leq\ :\ b \leqslant_A a)\ (o\ :\ \mathsf{Ordered}\ A\ \_\leqslant_A\_\ a\ as)\end{aligned}$$

Since a proof of $\mathsf{Ordered}\ A\ \_\leqslant_A\_\ b\ as$ consists of exactly the inequality proofs necessary for ensuring that *as* is ordered and bounded below by *b*, its representation is optimised, justifying the name "optimised predicate". □

Example (*inductive length predicate on lists*)**.** The optimised predicate for the ornament *ListD-VecD A* from lists to vectors is produced by the ornamental description *OptPOD* (*ListD-VecD A*) relative to *ListD A*:

$$\begin{aligned}\lambda\,\{\ &(\mathsf{ok}\ (\mathsf{ok}\ \mathsf{zero}\quad\ ,\ \mathsf{ok}\ (\blacksquare\ ,\ []\quad\ )))\ \mapsto\ \Delta[\,\_ : \text{`nil}\ \equiv\ \text{`nil}\,]\ \nabla[\,\text{`nil}\,]\ \mathsf{v}\ \blacksquare\\ ;\ &(\mathsf{ok}\ (\mathsf{ok}\ \mathsf{zero}\quad\ ,\ \mathsf{ok}\ (\blacksquare\ ,\ a :: as)))\ \mapsto\ \Delta\ (\text{`nil}\ \equiv\ \text{`cons})\ \lambda\ ()\\ ;\ &(\mathsf{ok}\ (\mathsf{ok}\ (\mathsf{suc}\ n)\ ,\ \mathsf{ok}\ (\blacksquare\ ,\ []\quad\ )))\ \mapsto\ \Delta\ (\text{`cons}\ \equiv\ \text{`nil})\ \lambda\ ()\end{aligned}$$

$$; \ (\mathsf{ok}\ (\mathsf{ok}\ (\mathsf{suc}\ n)\ ,\ \mathsf{ok}\ (\blacksquare\ ,\ a :: as))) \ \mapsto\ \Delta[\ \_ : \text{`}\mathsf{cons} \ \equiv\ \text{`}\mathsf{cons}\ ]\ \nabla[\ \text{`}\mathsf{cons}\ ]$$
$$\nabla[\ a\ ]\ \mathsf{v}\ (\mathsf{ok}\ (\ \mathsf{ok}\ n\ ,\ \mathsf{ok}\ (\blacksquare\ ,\ as)\ )\ ,\ \blacksquare)\ \}$$

where lighter box indicates contributions from *ListD-VecD A* and darker box from $\lceil\ singletonOD\ (ListD\ A)\ \rceil$. Both ornaments perform pattern matching and accordingly restrict constructor choices by ∇, so the resulting four cases all start with an equality field demanding that the constructor choices specified by the two ornaments are equal.

- In the first and last cases, where the specified constructor choices match, the equality proof obligation can be successfully discharged and the response ornamental description can continue after installing the constructor choice by ∇;
- in the middle two cases, where the specified constructor choices mismatch, the equality is obviously unprovable and the rest of the response ornamental description is (extensionally) the empty function $\lambda\ ()$.

Thus, in effect, the ornamental description produces the following inductive length predicate on lists:

$$\begin{array}{l}
\textbf{indexfirst data}\ \mathsf{Length}\ A\ :\ \mathsf{Nat} \to \mathsf{List}\ A \to \mathsf{Set}\ \textbf{where} \\
\quad \mathsf{Length}\ A\ \mathsf{zero}\ \quad [\,]\ \quad \ni\ \mathsf{nil} \\
\quad \mathsf{Length}\ A\ \mathsf{zero}\ \quad (a :: as)\ \ \not\ni \\
\quad \mathsf{Length}\ A\ (\mathsf{suc}\ n)\ [\,]\ \quad \not\ni \\
\quad \mathsf{Length}\ A\ (\mathsf{suc}\ n)\ (a :: as)\ \ni\ \mathsf{cons}\ (l\ :\ \mathsf{Length}\ A\ n\ as)
\end{array}$$

where $\not\ni$ indicates that a case is uninhabited. $\square$

We have thus determined the promotion predicate used by the refinement semantics of ornaments to be the optimised predicate:

$$\begin{array}{l}
RSem\ :\ \{I\ J\ :\ \mathsf{Set}\}\ \{e\ :\ J \to I\}\ \{D\ :\ \mathsf{Desc}\ I\}\ \{E\ :\ \mathsf{Desc}\ J\} \to \\
\qquad \mathsf{Orn}\ e\ D\ E \to \mathsf{FRefinement}\ e\ (\mu\ D)\ (\mu\ E) \\
RSem\ O\ j\ =\ \textbf{record}\ \{\ P\ =\ \mathsf{OptP}\ O\ j \\
\qquad\qquad\qquad\qquad ;\ i\ \ =\ ornConvIso\ O\ j\ \}
\end{array}$$

We call *ornConvIso* the **ornamental conversion isomorphisms**, whose type is

$$ornConvIso\ :$$

$\{I\ J\ :\ \mathsf{Set}\}\ \{e\ :\ J \to I\}\ \{D\ :\ \mathsf{Desc}\ I\}\ \{E\ :\ \mathsf{Desc}\ J\}\ (O\ :\ \mathsf{Orn}\ e\ D\ E) \to$
$\{i\ :\ I\}\ (j\ :\ e^{-1}\ i) \to \mu\ E\ (und\ j)\ \cong\ \Sigma[\, x\ :\ \mu\ D\ i\,]\ \mathsf{OptP}\ O\ j\ x$

The construction of *ornConvIso* will be deferred until **??**.

3.3.2 Predicate swapping for parallel composition

An ornament describes differences between two datatypes, and the optimised predicate for the ornament is the datatype of differences between inhabitants of the two datatypes. To promote an inhabitant from the less informative end to the more informative end of the ornament using its refinement semantics, we give a proof that the object satisfies the optimised predicate for the ornament. If, however, the ornament is a parallel composition, say $\lceil\, O \otimes P\,\rceil$, then the differences recorded in the ornament are simply collected from the component ornaments O and P. Consequently, it should suffice to give separate proofs that the inhabitant satisfies the optimised predicates for O and P, instead of a proof that it satisfies the monolithic optimised predicate induced by $\lceil O \otimes P \rceil$. We are thus led to prove that the optimised predicate for $\lceil\, O \otimes P\,\rceil$ amounts to the pointwise conjunction of the optimised predicates for O and P. More precisely: if $O\ :\ \mathsf{Orn}\ e\ D\ E$ and $P\ :\ \mathsf{Orn}\ f\ D\ F$ where $D\ :\ \mathsf{Desc}\ I$, $E\ :\ \mathsf{Desc}\ J$, and $F\ :\ \mathsf{Desc}\ K$, then we expect the existence of the **modularity isomorphisms**

$$\mathsf{OptP}\ \lceil O \otimes P \rceil\ (\mathsf{ok}\ (j\ ,\ k))\ x\ \cong\ \mathsf{OptP}\ O\ j\ x \times \mathsf{OptP}\ P\ k\ x$$

for all $i\ :\ I, j\ :\ e^{-1}\ i, k\ :\ f^{-1}\ i$, and $x\ :\ \mu\ D\ i$.

Example (*promotion predicate from lists to ordered vectors*)**.** The optimised predicate for the ornament $\lceil\, \lceil\, \mathit{OrdListOD}\ A\ \_\leqslant_A\_\,\rceil \otimes \mathit{ListD\text{-}VecD}\ A\,\rceil$ from lists to ordered vectors is

indexfirst data $\mathsf{OrderedLength}\ A\ \_\leqslant_A\_\ :\ A \to \mathsf{Nat} \to \mathsf{List}\ A \to \mathsf{Set}$ **where**
$\mathsf{OrderedLength}\ A\ \_\leqslant_A\_\ b\ \mathsf{zero}\quad [\,]\quad \ni\ \mathsf{nil}$
$\mathsf{OrderedLength}\ A\ \_\leqslant_A\_\ b\ \mathsf{zero}\quad (a :: as)\quad \not\ni$
$\mathsf{OrderedLength}\ A\ \_\leqslant_A\_\ b\ (\mathsf{suc}\ n)\ [\,]\quad \not\ni$
$\mathsf{OrderedLength}\ A\ \_\leqslant_A\_\ b\ (\mathsf{suc}\ n)\ (a :: as)$
$\quad \ni\ \mathsf{cons}\ (leq\ :\ b \leqslant_A a)\ (ol\ :\ \mathsf{OrderedLength}\ A\ \_\leqslant_A\_\ a\ n\ as)$

which is monolithic and inflexible. We can avoid using this predicate directly by exploiting the modularity isomorphisms

$$\mathsf{OrderedLength}\ A\ \_{\leqslant_A}\_\ b\ n\ as\ \cong\ \mathsf{Ordered}\ A\ \_{\leqslant_A}\_\ b\ as\ \times\ \mathsf{Length}\ A\ n\ as$$

for all $b : A$, $n : \mathsf{Nat}$, and $as : \mathsf{List}\ A$ — to promote a list to an ordered vector, we can prove that it satisfies Ordered and Length instead of OrderedLength. Promotion proofs from lists to ordered vectors can thus be divided into ordering and length aspects and carried out separately. □

Along with the ornamental conversion isomorphisms, the construction of the modularity isomorphisms will be deferred until **??**. Here we deal with a practical issue regarding composition of modularity isomorphisms: for example, to get pointwise isomorphisms between the optimised predicate for $\lceil\, O \otimes \lceil\, P \otimes Q \,\rceil\,\rceil$ and the pointwise conjunction of the optimised predicates for O, P, and Q, we need to instantiate the modularity isomorphisms twice and compose the results appropriately, a procedure which quickly becomes tedious. What we need is an auxiliary mechanism that helps with organising computation of composite predicates and isomorphisms following the parallel compositional structure of ornaments, in the same spirit as the upgrade mechanism (Section 3.1.2) helping with organising computation of coherence properties and proofs following the syntactic structure of function types.

We thus define the following auxiliary datatype Swap, parametrised with a refinement whose promotion predicate is to be swapped for a new one:

$$\begin{array}{l}\textbf{record}\ \mathsf{Swap}\ \{X\ Y : \mathsf{Set}\}\ (r : \mathsf{Refinement}\ X\ Y) : \mathsf{Set}_1\ \textbf{where}\\ \quad \textbf{field}\\ \qquad Q : X \to \mathsf{Set}\\ \qquad i\ \ : (x : X) \to \mathsf{Refinement}.P\ r\ x\ \cong\ Q\ x\end{array}$$

An inhabitant of Swap r consists of a new promotion predicate for r and a proof that the new predicate is pointwise isomorphic to the original one in r. The actual swapping is done by the function

$$\begin{array}{l}toRefinement : \{X\ Y : \mathsf{Set}\}\ \{r : \mathsf{Refinement}\ X\ Y\} \to \mathsf{Swap}\ r \to \mathsf{Refinement}\ X\ Y\\ toRefinement\ s\ =\ \textbf{record}\ \{\ P\ =\ \mathsf{Swap}.Q\ s\\ \qquad\qquad\qquad\qquad ;\ i\ \ =\ \{\ \}_0\ \}\end{array}$$

where Goal 0 is the new conversion isomorphism

$$Y \cong \Sigma\, X\, (\mathsf{Refinement}.P\; r) \cong \Sigma\, X\, (\mathsf{Swap}.Q\; s)$$

constructed by using transitivity and product of isomorphisms to compose $\mathsf{Refinement}.i\; r$ and $\mathsf{Swap}.i\; s$. We can then define the datatype FSwap of "swap families" in the usual way:

$$\begin{aligned}
&\mathsf{FSwap} : \{I\, J : \mathsf{Set}\}\, \{e : J \to I\}\, \{X : I \to \mathsf{Set}\}\, \{Y : J \to \mathsf{Set}\} \to \\
&\qquad (rs : \mathsf{FRefinement}\; e\; X\; Y) \to \mathsf{Set}_1 \\
&\mathsf{FSwap}\; rs = \{i : I\}\, (j : e^{-1}\; i) \to \mathsf{Swap}\; (rs\; j)
\end{aligned}$$

and provide the following combinator on swap families, which says that if there are alternative promotion predicates for the refinement semantics of O and P, then the pointwise conjunction of the two predicates is an alternative promotion predicate for the refinement semantics of $\lceil O \otimes P \rceil$:

$$\begin{aligned}
&\otimes\text{-}FSwap : \{I\, J\, K : \mathsf{Set}\}\, \{e : J \to I\}\, \{f : K \to I\} \to \\
&\qquad \{D : \mathsf{Desc}\; I\}\, \{E : \mathsf{Desc}\; J\}\, \{F : \mathsf{Desc}\; K\} \to \\
&\qquad (O : \mathsf{Orn}\; e\; D\; E)\, (P : \mathsf{Orn}\; f\; D\; F) \to \\
&\qquad \mathsf{FSwap}\; (RSem\; O) \to \mathsf{FSwap}\; (RSem\; P) \to \mathsf{FSwap}\; (RSem\; \lceil O \otimes P \rceil) \\
&\otimes\text{-}FSwap\; O\; P\; ss\; ts\; (\mathsf{ok}\; (j\, , k)) = \\
&\quad \mathbf{record}\; \{\, Q = \lambda\, x \mapsto \mathsf{Swap}.Q\; (ss\; j)\; x \times \mathsf{Swap}.Q\; (ts\; k)\; x \\
&\qquad\quad ;\, i = \lambda\, x \mapsto \{\,\}_1\, \}
\end{aligned}$$

Goal 1 is straightforwardly discharged by composing the modularity isomorphisms and the isomorphisms in *ss* and *ts*:

$$\begin{aligned}
\mathsf{OptP}\; \lceil O \otimes P \rceil\; (\mathsf{ok}\; (j\, , k))\; x &\cong \mathsf{OptP}\; O\; j\; x \quad\quad \times \mathsf{OptP}\; P\; k\; x \\
&\cong \mathsf{Swap}.Q\; (ss\; j)\; x \times \mathsf{Swap}.Q\; (ts\; k)\; x
\end{aligned}$$

Example (*modular promotion predicate for the parallel composition of three ornaments*)**.** To use the pointwise conjunction of the optimised predicates for ornaments O, P, and Q as an alternative promotion predicate for $\lceil O \otimes \lceil P \otimes Q \rceil \rceil$, we use the swap family

$$\otimes\text{-}FSwap\; O\; \lceil P \otimes Q \rceil\; id\text{-}FSwap\; (\otimes\text{-}FSwap\; P\; Q\; id\text{-}FSwap\; id\text{-}FSwap)$$

where

$$\mathit{id\text{-}FSwap} : \{I : \mathsf{Set}\}\ \{X\ Y : I \to \mathsf{Set}\}\ \{rs : \mathsf{FRefinement}\ X\ Y\} \to \mathsf{FSwap}\ rs$$

simply retains the original promotion predicate in rs. □

Example (*swapping the promotion predicate from lists to ordered vectors*)**.** The swap family

$$\otimes\text{-}\mathit{FSwap}\ \lceil \mathit{OrdListOD}\ A\ \_\leqslant_A\_ \rceil\ (\mathit{ListD\text{-}VecD}\ A)\ \mathit{id\text{-}FSwap}\ (\mathit{Length\text{-}FSwap}\ A)$$

yields a refinement family from lists to ordered vectors using

$$\lambda\, b\ n\ as \mapsto \mathsf{Ordered}\ A\ \_\leqslant_A\_\ b\ as \times \mathit{length}\ as \equiv n$$

as the promotion predicate, where

$$\mathit{Length\text{-}FSwap}\ A : \mathsf{FSwap}\ (\mathit{RSem}\ (\mathit{ListD\text{-}VecD}\ A))$$

swaps $\mathsf{Length}\ A$ for $\lambda\, n\ as \mapsto \mathit{length}\ as \equiv n$. □

3.4 Examples

To demonstrate the use of the ornament–refinement framework, we first resolve the library structuring problem about insertion into a list and then look at two dependently typed heap data structures adapted from Okasaki's work on purely functional data structures [1999]. Of the latter two examples,

- the first one about **binomial heaps** shows that Okasaki's idea of **numerical representations** can be elegantly captured by ornaments and the coherence properties computed with upgrades, and
- the second one about **leftist heaps** demonstrates the power of parallel composition of ornaments by treating heap ordering and leftist balancing properties modularly.

Postulate operations on *Val* like $\_\leqslant_?\_$, $\leqslant$*-refl*, $\leqslant$*-trans*, and $\not\leq$*-invert* in Chapter 2.

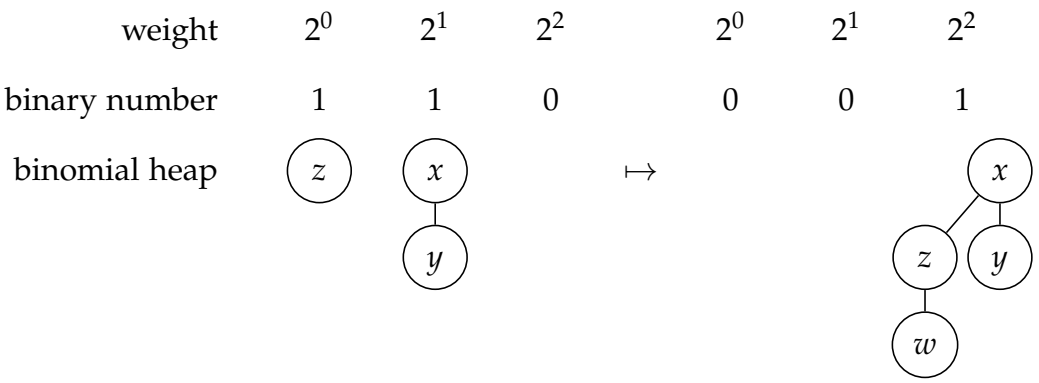


Figure 3.5 **Left:** a binomial heap of size 3 consisting of two binomial trees storing elements x, y, and z. **Right:** the result of inserting an element w into the heap. (Note that the digits of the underlying binary numbers are ordered with the least significant digit first.)

3.4.1 Insertion into a list

3.4.2 Binomial heaps

We are all familiar with the idea of **positional number systems**, in which we represent numbers as a list of digits. Each position in a list of digits 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 3.5) can be thought of as a binary number in which every 1-digit stores a **binomial tree** — the actual place for storing elements — whose size is exactly the weight of the 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, with carrying corresponding to combining smaller binomial trees into larger ones. Okasaki thus proposed to design container data structures by analogy with positional representations of natural numbers, and called such data structures **numerical representations**.

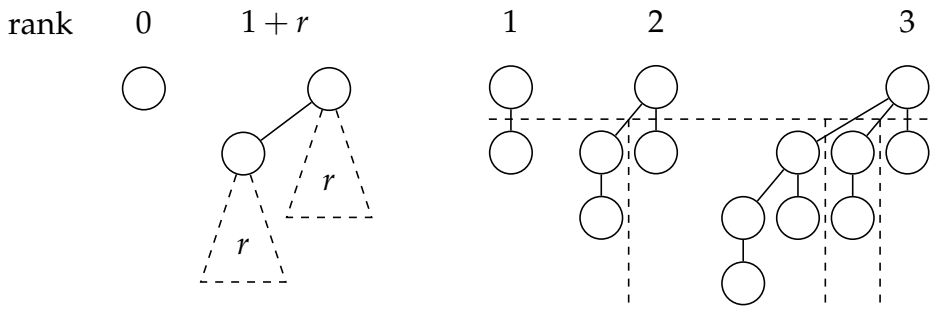


Figure 3.6 **Left:** inductive definition of binomial trees. **Right:** decomposition of binomial trees of ranks 1 to 3.

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.

We use insertion into a binomial heap as an example, which is presented in detail below.

Binomial trees

The basic building blocks of binomial heaps are **binomial trees**, in which elements are stored. Binomial trees are defined inductively on their **rank**, which is a natural number (see Figure 3.6):

- a binomial tree of rank 0 is a single node storing an element of type *Val*, and
- a binomial tree of rank $1 + r$ 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 useful: a binomial tree of rank r is constructed by attaching binomial trees of ranks 0 to $r-1$ under a root node. (Figure 3.6 shows how binomial trees of ranks 1 to 3 can be decomposed according to this view.) We thus define the datatype $\mathsf{BTree} : \mathsf{Nat} \to \mathsf{Set}$ — which is indexed with the rank of binomial trees — as follows: for any rank $r : \mathsf{Nat}$, the type $\mathsf{BTree}\ r$ has a field of type *Val* — which is the root node — and r recursive positions indexed from $r-1$ down to 0. This is directly encoded as a description:

$$
\begin{aligned}
&\mathit{BTreeD} : \mathsf{Desc\ Nat} \\
&\mathit{BTreeD}\ r \;=\; \sigma[\,\_ : \mathit{Val}\,]\ \mathsf{v}\ (\mathit{descend}\ r)
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{BTree} : \mathsf{Nat} \to \mathsf{Set} \\
&\mathsf{BTree} \;=\; \mu\ \mathit{BTreeD}
\end{aligned}
$$

where *descend* r is a list from $r-1$ down to 0:

$$
\begin{aligned}
&\mathit{descend} : \mathsf{Nat} \to \mathsf{List\ Nat} \\
&\mathit{descend}\ \mathsf{zero} \quad\;\; = [\,] \\
&\mathit{descend}\ (\mathsf{suc}\ n) \;=\; n :: \mathit{descend}\ n
\end{aligned}
$$

Note that, in *BTreeD*, we are exploiting the full computational power of Desc, computing the list of recursive indices from the index request. Due to this, it is tricky to wrap up *BTreeD* as an index-first datatype declaration, so we will skip this step and work directly with the raw representation, which looks reasonably intuitive anyway: a binomial tree of type $\mathsf{BTree}\ r$ is of the form $\mathsf{con}\ (x\ ,\ ts)$ where $x : \mathit{Val}$ is the root element and $ts : \mathbb{P}\ (\mathit{descend}\ r)\ \mathsf{BTree}$ is a series of sub-trees.

The most important operation on binomial trees is combining two smaller binomial trees of the same rank into a larger one, which corresponds to carrying 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 $1 + r$ — this is exactly the inductive definition of binomial trees.

```
attach : {r : Nat} → BTree r → BTree r → BTree (suc r)
attach t (con (y , us)) = con (y , t , us)
```

For use in binomial heaps, though, we should ensure that elements in binomial trees are in **heap order**, i.e., the root of any binomial tree (including sub-trees) is the minimum element in the tree. This is achieved by comparing the roots of two binomial trees before deciding which one is to be attached to the other:

```
link : {r : Nat} → BTree r → BTree r → BTree (suc r)
link t u with root t ⩽? root u
link t u | yes _ = attach u t
link t u | no  _ = attach t u
```

where *root* extracts the root element of a binomial tree:

```
root : {r : Nat} → BTree r → Val
root (con (x , ts)) = x
```

If we always build binomial trees of positive rank by *link*, then the elements in any binomial tree we build would be in heap order. This is a crucial assumption in binomial heaps (which is not essential to our development, though).

From binary numbers to binomial heaps

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

```
data BinTag : Set where 'nil 'zero 'one : BinTag

BinD : Desc ⊤
BinD ▪ = σ BinTag λ { 'nil  ↦ v []
                    ; 'zero ↦ v (▪ :: [])
                    ; 'one  ↦ v (▪ :: []) }

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

The intended interpretation of binary numbers is given by

```
toNat : Bin → Nat
toNat nil        = 0
toNat (zero b)   = 0 + 2 * toNat b
toNat (one  b)   = 1 + 2 * toNat b
```

That is, the list of digits of a binary number of type Bin starts from the least significant digit, and the i-th digit (counting from 0) 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 beginning, binomial heaps are binary numbers whose 1-digits are decorated with binomial trees of matching rank, which can be expressed straightforwardly as an ornamentation of binary numbers. To ensure that the binomial trees in binomial heaps have the right rank, the datatype BHeap : Nat → Set is indexed with a "starting rank": if a binomial heap of type BHeap r is nonempty (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 $1 + r$.

```
BHeapOD : OrnDesc Nat ! BinD
BHeapOD (ok r) = σ BinTag λ { 'nil  ↦ v ▪
                            ; 'zero ↦ v (ok (suc r) , ▪)
                            ; 'one  ↦ Δ[ t : BTree r ] v (ok (suc r) , ▪) }

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

In applications, we would use binomial heaps of type BHeap 0, which encompasses binomial heaps of all sizes.

Increment and insertion, in coherence

Increment of binary numbers is defined by

$$
\begin{array}{l}
\mathit{incr} \;:\; \mathsf{Bin} \to \mathsf{Bin} \\
\mathit{incr}\ \mathsf{nil} \quad\;\; = \; \mathsf{one}\ \mathsf{nil} \\
\mathit{incr}\ (\mathsf{zero}\ b) \; = \; \mathsf{one}\ b \\
\mathit{incr}\ (\mathsf{one}\ \ b) \; = \; \mathsf{zero}\ (\mathit{incr}\ b)
\end{array}
$$

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

$$
\begin{array}{l}
\mathit{insT} \;:\; \{r \;:\; \mathsf{Nat}\} \to \mathsf{BTree}\ r \to \mathsf{BHeap}\ r \to \mathsf{BHeap}\ r \\
\mathit{insT}\ t\ \mathsf{nil} \quad\;\; = \; \mathsf{one}\ t\ \mathsf{nil} \\
\mathit{insT}\ t\ (\mathsf{zero}\ \ h) \; = \; \mathsf{one}\ t\ h \\
\mathit{insT}\ t\ (\mathsf{one}\ u\ h) \; = \; \mathsf{zero}\ (\mathit{insT}\ (\mathit{link}\ t\ u)\ h)
\end{array}
$$

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 carrying happens. Having defined *insT*, inserting a single element into a binomial heap of type $\mathsf{BHeap}\ 0$ is then inserting, by *insT*, a rank-0 binomial tree (i.e., a single node) storing the element into the heap.

$$
\begin{array}{l}
\mathit{insert} \;:\; \mathit{Val} \to \mathsf{BHeap}\ 0 \to \mathsf{BHeap}\ 0 \\
\mathit{insert}\ x \;=\; \mathit{insT}\ (\mathsf{con}\ (x\ ,\ \blacksquare))
\end{array}
$$

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:

$$
\begin{array}{l}
\mathit{upg} \;:\; \mathsf{Upgrade}\ (\mathsf{Bin} \to \mathsf{Bin})\ (\{r \;:\; \mathsf{Nat}\} \to \mathsf{BTree}\ r \to \mathsf{BHeap}\ r \to \mathsf{BHeap}\ r) \\
\mathit{upg} \;=\; \forall^{+}[[\, r : \mathsf{Nat}\,]]\ \forall^{+}[\, \_ : \mathsf{BTree}\ r\,] \\
\qquad\quad \mathbf{let}\ \mathit{ref} \;:\; \mathsf{Refinement}\ \mathsf{Bin}\ (\mathsf{BHeap}\ r) \\
\qquad\qquad\quad \mathit{ref} \;=\; \mathit{RSem}\ \lceil \mathit{BHeapOD} \rceil\ (\mathsf{ok}\ r) \\
\qquad\quad \mathbf{in}\ \ \mathit{ref} \rightharpoonup \mathit{toUpgrade}\ \mathit{ref}
\end{array}
$$

The upgrade *upg* says that, compared to the type of *incr*, the type of *insT* has two new arguments — the implicit argument r : Nat 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 refinement semantics of the ornament $\lceil BHeapOD \rceil$ (ok r) from Bin to BHeap r. The type Upgrade.*C upg incr insT* (which states that *incr* and *insT* are in coherence with respect to *upg*) expands to

$$\{r : \mathsf{Nat}\}\ (t : \mathsf{BTree}\ r)\ (b : \mathsf{Bin})\ (h : \mathsf{BHeap}\ r) \rightarrow$$
$$toBin\ h \equiv b \rightarrow toBin\ (insT\ t\ h) \equiv incr\ b$$

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

$$toBin : \{r : \mathsf{Nat}\} \rightarrow \mathsf{BHeap}\ r \rightarrow \mathsf{Bin}$$
$$toBin = forget\ \lceil BHeapOD \rceil$$

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* does.

We have seen that the coherence property of *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 *insT* that satisfies the coherence property automatically. The above 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 in coherence with *incr*.

Let BHeap′ be the optimised predicate for the ornament from Bin to BHeap r:

$$\mathsf{BHeap}' : \mathsf{Nat} \rightarrow \mathsf{Bin} \rightarrow \mathsf{Set}$$
$$\mathsf{BHeap}'\ r\ b = \mathsf{OptP}\ \lceil BHeapOD \rceil\ (\mathsf{ok}\ r)\ b$$

$$
\begin{array}{l}
\textbf{indexfirst data}\ \mathsf{BHeap'}\ :\ \mathsf{Nat} \to \mathsf{Bin} \to \mathsf{Set}\ \textbf{where} \\
\quad \mathsf{BHeap'}\ r\ \mathsf{nil} \ni \mathsf{nil} \\
\quad \mathsf{BHeap'}\ r\ (\mathsf{zero}\ b) \ni \mathsf{zero}\ (h : \mathsf{BHeap'}\ (\mathsf{suc}\ r)\ b) \\
\quad \mathsf{BHeap'}\ r\ (\mathsf{one}\ b) \ni \mathsf{one}\ (t : \mathsf{BTree}\ r)\ (h : \mathsf{BHeap'}\ (\mathsf{suc}\ r)\ b)
\end{array}
$$

Here a more helpful interpretation is that $\mathsf{BHeap'}$ is a datatype of binomial heaps additionally indexed with the underlying binary number. The type $\mathsf{Upgrade}.P\ upg\ incr$ of promotion proofs for *incr* then expands to

$$\{r : \mathsf{Nat}\} \to \mathsf{BTree}\ r \to (b : \mathsf{Bin}) \to \mathsf{BHeap'}\ r\ b \to \mathsf{BHeap'}\ r\ (incr\ b)$$

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

$$
\begin{array}{l}
insT' : \{r : \mathsf{Nat}\} \to \mathsf{BTree}\ r \to (b : \mathsf{Bin}) \to \mathsf{BHeap'}\ r\ b \to \mathsf{BHeap'}\ r\ (incr\ b) \\
insT'\ t\ \mathsf{nil}\ \ \ \mathsf{nil}\ \ \ = \mathsf{one}\ t\ \mathsf{nil} \\
insT'\ t\ (\mathsf{zero}\ b)\ (\mathsf{zero}\ \ h) = \mathsf{one}\ t\ h \\
insT'\ t\ (\mathsf{one}\ \ b)\ (\mathsf{one}\ u\ h) = \mathsf{zero}\ (insT'\ (link\ t\ u)\ h)
\end{array}
$$

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

$$
\begin{array}{l}
insT' : \{r : \mathsf{Nat}\} \to \mathsf{BTree}\ r \to (b : \mathsf{Bin}) \to \mathsf{BHeap'}\ r\ b \to \mathsf{BHeap'}\ r\ (incr\ b) \\
insT'\ t\ (\mathsf{con}\ (\text{`}\mathsf{nil}\ \ \ ,\ \ \ \blacksquare))\ (\mathsf{con}\ \ \ \ \ \ \ \blacksquare\ ) = \mathsf{con}\ (t\ ,\ \mathsf{con}\ \blacksquare\ \ \ \ \ \ \ \ \ ,\ \blacksquare) \\
insT'\ t\ (\mathsf{con}\ (\text{`}\mathsf{zero}\ ,\ b\ ,\ \blacksquare))\ (\mathsf{con}\ (\ \ \ h\ ,\ \blacksquare)) = \mathsf{con}\ (t\ ,\ h\ \ \ \ \ \ \ \ \ \ \ \ \ \ ,\ \blacksquare) \\
insT'\ t\ (\mathsf{con}\ (\text{`}\mathsf{one}\ \ ,\ b\ ,\ \blacksquare))\ (\mathsf{con}\ (u\ ,\ h\ ,\ \blacksquare)) = \mathsf{con}\ (\ \ \ insT'\ (link\ t\ u)\ b\ h\ ,\ \blacksquare)
\end{array}
$$

in which no constructor tags for binomial heaps are present. This means that the types would instruct which constructors to use when programming $insT'$, establishing the coherence property by construction. Finally, since $insT'$ is a promotion proof for *incr*, we can invoke the upgrading operation of *upg* and get *insT*:

$$
\begin{array}{l}
insT : \{r : \mathsf{Nat}\} \to \mathsf{BTree}\ r \to \mathsf{BHeap}\ r \to \mathsf{BHeap}\ r \\
insT = \mathsf{Upgrade}.u\ upg\ incr\ insT'
\end{array}
$$

which is automatically in coherence with *incr*:

```
incr-insT-coherence : {r : Nat} (t : BTree r) (b : Bin) (h : BHeap r) →
                      toBin h ≡ b → toBin (insT t h) ≡ incr b
incr-insT-coherence = Upgrade.c upg incr insT′
```

Summary

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 *insT′*, 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 by *upg*. Compared to Okasaki's implementation, besides rank-indexing, which elegantly transfers the management of rank-related invariants to the type system, the extra work is only the straightforward markings of the differences between Bin and BHeap (in *BHeapOD*) and between the type of *incr* and that of *insT* (in *upg*). The reward is huge 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. From straightforward markings to nontrivial types and programs — this clearly demonstrates the power of the ornament–refinement framework.

3.4.3 Leftist heaps

Our second example is about treating the ordering and balancing properties of **leftist heaps** modularly. In Okasaki's words:

> Leftist heaps [...] are heap-ordered binary trees that satisfy the **leftist property**: the rank of any left child is at least as large as the rank of its right sibling. The rank of a node is defined to be the length of its **right spine** (i.e., the rightmost path from the node in question to an empty node).

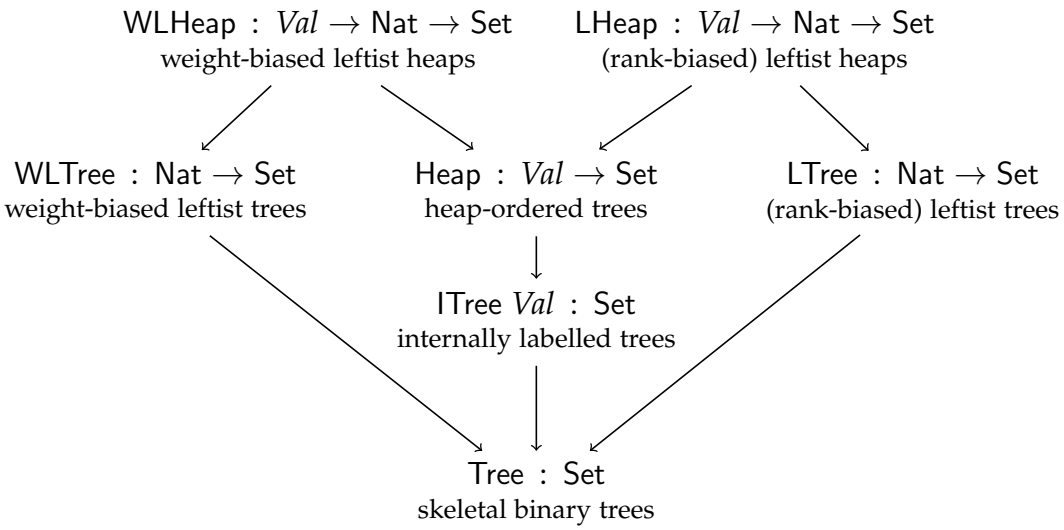


Figure 3.7 Datatypes about leftist heaps and their ornamental relationships.

From this passage we can immediately analyse the concept of leftist heaps into three: leftist heaps (i) are binary trees that (ii) are heap-ordered and (iii) satisfy the leftist property. This suggests that there is a basic datatype of binary trees together with two ornamentations, one expressing heap ordering and the other the leftist property. The datatype of leftist heaps is then synthesised by composing the two ornamentations in parallel. All the datatypes about leftist heaps and their ornamental relationships are shown in Figure 3.7.

Datatypes leading to leftist heaps

The basic datatype Tree : Set of "skeletal" binary trees, which consist of empty nodes and internal nodes not storing any elements, is defined by

```
data TreeTag : Set where ‘nil ‘node : TreeTag

TreeD : Desc ⊤
TreeD ▪ = σ TreeTag λ { ‘nil  ↦ v []
                      ; ‘node ↦ v (▪ :: ▪ :: []) }
```

```
indexfirst data Tree : Set where
  Tree ∋ nil
       | node (t : Tree) (u : Tree)
```

Leftist trees — skeletal binary trees satisfying the leftist property — are then an ornamented version of Tree. The datatype LTree : Nat → Set of leftist trees is indexed with the rank of the root of the trees. The constructor choices can be determined from the rank: the only node that can have rank zero is the empty node nil; otherwise, when the rank of a node is non-zero, it must be an internal node constructed by the node constructor, which enforces the leftist property.

```
LTreeOD : OrnDesc Nat ! TreeD
LTreeOD (ok zero   ) = ∇['nil] v ▪
LTreeOD (ok (suc r)) = ∇['node] Δ[l : Nat] Δ[r⩽l : r ⩽ l] v (ok l , ok r , ▪)
indexfirst data LTree : Nat → Set where
  Tree zero    ∋ nil
  Tree (suc r) ∋ node {l : Nat} (r⩽l : r ⩽ l) (t : Tree l) (u : Tree r)
```

Independently, **heap-ordered trees** are also an ornamented version of Tree. The datatype Heap : *Val* → Set of heap-ordered trees can be regarded as a generalisation of ordered lists: in a heap-ordered tree, every path from the root to an empty node is an ordered list.

```
HeapOD : OrnDesc Val ! TreeD
HeapOD (ok b) =
  σ TreeTag λ { 'nil  ↦ v ▪
              ; 'node ↦ Δ[x : Val] Δ[b⩽x : b ⩽ x] v (ok x , ok x , ▪) }
indexfirst data Heap : Val → Set where
  Heap b ∋ nil
         | node (x : Val) (b⩽x : b ⩽ x) (t : Heap x) (u : Heap x)
```

Composing the two ornaments in parallel gives us exactly the datatype of leftist heaps.

```
LHeapOD : OrnDesc (! ⋈ !) pull TreeD
LHeapOD = ⌈HeapOD⌉ ⊗ ⌈LTreeOD⌉
```

```
indexfirst data LHeap : Val → Nat → Set where
  LHeap b zero     ∋ nil
  LHeap b (suc r)  ∋ node (x : Val) (b⩽x : b ⩽ x)
                          {l : Nat} (r⩽l : r ⩽ l) (t : Heap x l) (u : Heap x r)
```

Operations on leftist heaps

The analysis of leftist heaps as the parallel composition of the two ornamentations allows us to talk about heap ordering and the leftist property independently. For example, a useful operation on heap-ordered trees is relaxing the lower bound. It can be regarded as an upgraded version of the identity function on Tree, since it leaves the tree structure intact, changing only the ordering information. With the help of the optimised predicate for $\lceil HeapOD \rceil$,

```
Heap′ : Val → Set
Heap′ b = OptP ⌈HeapOD⌉ (ok b)

indexfirst data Heap′ : Val → Tree → Set where
  Heap′ b nil        ∋ nil
  Heap′ b (node t u) ∋ node (x : Val) (b⩽x : b ⩽ x)
                            (t′ : Heap x t) (u′ : Heap x u)
```

we can give the type of bound-relaxing in predicate form, stating explicitly in the type that the underlying tree structure is unchanged:

```
relax : {b b′ : Val} → b′ ⩽ b → {t : Tree} → Heap′ b t → Heap′ b′ t
relax b′⩽b {nil}      nil                = nil
relax b′⩽b {node _ _} (node x b⩽x t u) = node x (⩽-trans b′⩽b b⩽x) t u
```

Since the identity function on LTree can also be seen as an upgraded version of the identity function on Tree, we can combine *relax* and the predicate form of the identity function on LTree to get bound-relaxing on leftist heaps, which modifies only the heap-ordering portion of a leftist heap:

```
lhrelax : {b b′ : Val} → b′ ⩽ b → {r : Nat} → LHeap b r → LHeap b′ r
lhrelax {b} {b′} b′⩽b {r} = Upgrade.u upg id (λ _ ↦ relax b′⩽b ∗ id)
```

```
  where
    ref : (b'' : Val) → Refinement Tree (LHeap b'' r)
    ref b'' = toRefinement
                (⊗-FSwap ⌈HeapOD⌉ ⌈LTreeOD⌉ id-FSwap id-FSwap
                  (ok (ok b'' , ok r)))
    upg : Upgrade (Tree → Tree) (LHeap b r → LHeap b' r)
    upg = ref b ⇀ toUpgrade (ref b')
```

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

```
ITreeOD : Set → OrnDesc ⊤ ! TreeD
ITreeOD A ▪ = σ TreeTag λ { 'nil  ↦ v ▪
                          ; 'node ↦ Δ[ _ : A ] v (ok tt, ok tt, ▪) }

indexfirst data ITree (A : Set) : Set where
  ITree A ∋ nil
          | node (x : A) (t : ITree A) (u : ITree A)
```

on which we can do preorder traversal:

```
preorder : {A : Set} → ITree A → List A
preorder nil        = []
preorder (node x t u) = x :: preorder t ++ preorder u
```

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

```
ITreeD-HeapD : Orn ! ⌊ITreeOD Val⌋ ⌊HeapOD⌋
ITreeD-HeapD (ok b) =
  σ TreeTag λ { 'nil  ↦ v []
              ; 'node ↦ σ[ x : Val ] Δ[ _ : b ⩽ x ] v (refl :: refl :: []) }
```

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

use an ornament-parametrised upgrade

$$\begin{aligned}&\mathit{toList} : \{b : \mathit{Val}\}\ \{r : \mathsf{Nat}\} \to \mathsf{LHeap}\ b\ r \to \mathsf{List}\ \mathit{Val}\\&\mathit{toList} = \mathit{preorder} \circ \mathit{forget}\ (\mathit{ITreeD\text{-}HeapD} \odot \mathit{diffOrn\text{-}l}\ \lceil \mathit{HeapOD} \rceil\ \lceil \mathit{LTreeOD} \rceil)\end{aligned}$$

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

$$\begin{aligned}\mathit{merge} : {} & \{b_0 : \mathit{Val}\}\ \{r_0 : \mathsf{Nat}\} \to \mathsf{LHeap}\ b_0\ r_0 \to\\ & \{b_1 : \mathit{Val}\}\ \{r_1 : \mathsf{Nat}\} \to \mathsf{LHeap}\ b_1\ r_1 \to\\ & \{b : \mathit{Val}\} \to b \leqslant b_0 \to b \leqslant b_1 \to \Sigma[\, r : \mathsf{Nat}\,]\ \mathsf{LHeap}\ b\ r\end{aligned}$$

with which we can easily implement insertion of a new element (by merging with a singleton heap) and deletion of the minimum element (by deleting the root and merging the two sub-heaps). The definition of *merge* is shown in Figure 3.8. It is a more precisely typed version of Okasaki's implementation, split into two mutually recursive functions to make it clear to Agda's termination checker that we are doing two-level induction on the ranks of the two input heaps. When one of the ranks is zero, meaning that the corresponding heap is nil, we simply return the other heap (whose bound is suitably relaxed) as the result. When both ranks are nonzero, meaning that both heaps are nonempty, we compare the roots of the two heaps and recursively merge the heap with the larger root into the right branch of the heap with the smaller root. The recursion is structural because the rank of the right branch of a nonempty heap is strictly smaller. There is a catch, however: the rank of the new right subheap resulting from the recursive merging might be larger than that of the left sub-heap, violating the leftist property, so there is a helper function *makeT* that swaps the sub-heaps when necessary.

```
makeT : (x : Nat) → {r₀ : Nat} (h₀ : LHeap x r₀) →
                    {r₁ : Nat} (h₁ : LHeap x r₁) → Σ[ r : Nat ] LHeap x r
makeT x {r₀} h₀ {r₁} h₁ with r₀ ⩽? r₁
makeT x {r₀} h₀ {r₁} h₁ | yes r₀⩽r₁ = suc r₀ , node x ⩽-refl r₀⩽r₁          h₁ h₀
makeT x {r₀} h₀ {r₁} h₁ | no  r₀⩽̸r₁ = suc r₁ , node x ⩽-refl (⩽̸-invert r₀⩽̸r₁) h₀ h₁

mutual

  merge : {b₀ : Val} {r₀ : Nat} → LHeap b₀ r₀ →
          {b₁ : Val} {r₁ : Nat} → LHeap b₁ r₁ →
          {b  : Val} → b ⩽ b₀ → b ⩽ b₁ → Σ[ r : Nat ] LHeap b r
  merge {b₀} {zero  } nil h₁ b⩽b₀ b⩽b₁ = _ , lhrelax b⩽b₁ h₁
  merge {b₀} {suc r₀} h₀ h₁ b⩽b₀ b⩽b₁ = merge′ h₀ h₁ b⩽b₀ b⩽b₁

  merge′ : {b₀ : Val} {r₀ : Nat} → LHeap b₀ (suc r₀) →
           {b₁ : Val} {r₁ : Nat} → LHeap b₁ r₁       →
           {b  : Val} → b ⩽ b₀ → b ⩽ b₁ → Σ[ r : Nat ] LHeap b r
  merge′ h₀                  {b₁} {zero  } nil                  b⩽b₀ b⩽b₁ = _ , lhrelax b⩽b₀ h₀
  merge′ (node x₀ b₀⩽x₀ r₀⩽l₀ t₀ u₀) {b₁} {suc r₁} (node x₁ b₁⩽x₁ r₁⩽l₁ t₁ u₁) b⩽b₀ b⩽b₁ with x₀ ⩽? x₁
  merge′ (node x₀ b₀⩽x₀ r₀⩽l₀ t₀ u₀) {b₁} {suc r₁} (node x₁ b₁⩽x₁ r₁⩽l₁ t₁ u₁) b⩽b₀ b⩽b₁ | yes x₀⩽x₁ =
    _ , lhrelax (⩽-trans b⩽b₀ b₀⩽x₀) (outr (makeT x₀ t₀ (outr (merge u₀ (node x₁ x₀⩽x₁ r₁⩽l₁ t₁ u₁) ⩽-refl ⩽-refl))))
  merge′ (node x₀ b₀⩽x₀ r₀⩽l₀ t₀ u₀) {b₁} {suc r₁} (node x₁ b₁⩽x₁ r₁⩽l₁ t₁ u₁) b⩽b₀ b⩽b₁ | no  x₀⩽̸x₁ =
    _ , lhrelax (⩽-trans b⩽b₁ b₁⩽x₁) (outr (makeT x₁ t₁ (outr (merge′ (node x₀ (⩽̸-invert x₀⩽̸x₁) r₀⩽l₀ t₀ u₀) u₁ ⩽-refl ⩽-refl))))
```

Figure 3.8 Merging two leftist heaps. Proof terms about ordering are coloured grey to aid comprehension (taking inspiration from — but not really employing — Bernardy and Guilhem's type theory in colour [2013]).

Weight-biased leftist heaps

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

$$
\begin{aligned}
&\mathit{WLTreeOD} \;:\; \mathsf{OrnDesc}\ \mathsf{Nat}\ !\ \mathit{TreeD}\\
&\mathit{WLTreeOD}\ (\mathsf{ok}\ \mathsf{zero}\quad) \;=\; \nabla[\,\text{`}\mathsf{nil}\,]\ \mathsf{v}\ \blacksquare\\
&\mathit{WLTreeOD}\ (\mathsf{ok}\ (\mathsf{suc}\ n)) \;=\; \nabla[\,\text{`}\mathsf{node}\,]\ \Delta[\,l : \mathsf{Nat}\,]\ \Delta[\,r : \mathsf{Nat}\,]\\
&\qquad\qquad\qquad\qquad \Delta[\,\_ : r \leqslant l\,]\ \Delta[\,\_ : n \equiv l + r\,]\ \mathsf{v}\ (\mathsf{ok}\ l\ ,\ \mathsf{ok}\ r\ ,\ \blacksquare)
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{indexfirst data}\ \mathsf{WLTree} \;:\; \mathsf{Nat} \to \mathsf{Set}\ \textbf{where}\\
&\quad \mathsf{WLTree}\ \mathsf{zero}\quad \ni\ \mathsf{nil}\\
&\quad \mathsf{WLTree}\ (\mathsf{suc}\ n)\ \ni\ \mathsf{node}\ \{l \;:\; \mathsf{Nat}\}\ \{r \;:\; \mathsf{Nat}\}\\
&\qquad\qquad\qquad\qquad (r{\leqslant}l \;:\; r \leqslant l)\ (n{\equiv}l{+}r \;:\; n \equiv l + r)\\
&\qquad\qquad\qquad\qquad (t \;:\; \mathsf{WLTree}\ l)\ (u \;:\; \mathsf{WLTree}\ r)
\end{aligned}
$$

which can be composed in parallel with the heap-ordering ornament $\lceil \mathit{HeapOD} \rceil$ and gives us weight-biased leftist heaps.

$$
\begin{aligned}
&\mathit{WLHeapD} \;:\; \mathsf{Desc}\ (! \bowtie !)\\
&\mathit{WLHeapD} \;=\; \lfloor \lceil \mathit{HeapOD} \rceil \otimes \lceil \mathit{WLTreeOD} \rceil \rfloor
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{indexfirst data}\ \mathsf{WLHeap} \;:\; \mathit{Val} \to \mathsf{Nat} \to \mathsf{Set}\ \textbf{where}\\
&\quad \mathsf{WLHeap}\ b\ \mathsf{zero}\quad \ni\ \mathsf{nil}\\
&\quad \mathsf{WLHeap}\ b\ (\mathsf{suc}\ n)\ \ni\ \mathsf{node}\ (x \;:\; \mathit{Val})\ (b{\leqslant}x \;:\; b \leqslant x)\\
&\qquad\qquad\qquad\qquad \{l \;:\; \mathsf{Nat}\}\ \{r \;:\; \mathsf{Nat}\}\\
&\qquad\qquad\qquad\qquad (r{\leqslant}l \;:\; r \leqslant l)\ (n{\equiv}l{+}r \;:\; n \equiv l + r)\\
&\qquad\qquad\qquad\qquad (t \;:\; \mathsf{WLHeap}\ x\ l)\ (u \;:\; \mathsf{WLHeap}\ x\ r)
\end{aligned}
$$

The weight-biased leftist property makes it possible to reimplement merg-

ing in a single, top-down pass rather than two passes: With the original rank-biased leftist property, recursive calls to *merge* are determined top-down by comparing root elements, and the helper function *makeT* swaps a recursively computed sub-heap with the other sub-heap if the rank of the former is larger; the rank of a recursively computed sub-heap, however, is not known before a recursive call returns (which is reflected by the existential quantification of the rank index in the result type of *merge*), so during the whole merging process *makeT* does the swapping in a second bottom-up pass. On the other hand, with the weight-biased leftist property, the merging operation has type

$$\begin{aligned} \mathit{wmerge} \;:\; & \{b_0 \;:\; \mathit{Val}\}\; \{n_0 \;:\; \mathsf{Nat}\} \to \mathsf{WLHeap}\; b_0\; n_0 \to \\ & \{b_1 \;:\; \mathit{Val}\}\; \{n_1 \;:\; \mathsf{Nat}\} \to \mathsf{WLHeap}\; b_1\; n_1 \to \\ & \{b \;\;:\; \mathit{Val}\} \to b \leqslant b_0 \to b \leqslant b_1 \to \mathsf{WLHeap}\; b\; (n_0 + n_1) \end{aligned}$$

The implementation of *wmerge* is largely similar to *merge* and is omitted here. For *wmerge*, however, the weight of a recursively computed sub-heap is known before the recursive merging is actually performed (so the weight index can be given explicitly in the result type of *wmerge*). The counterpart of *makeT* can thus determine before a recursive call whether to do the swapping or not, and the whole merging process requires only one top-down pass.

Do we need a summary here?

3.5 Discussion

summary of the three-level architecture of ornaments, refinements, and upgrades; bundle; why ornaments?; functor-level computation and recursion schemes; compare with Bernardy and Guilhem [2013]

Bibliography

Jean-Philippe Bernardy and Moulin Guilhem [2013]. Type theory in color. In *International Conference on Functional Programming*, ICFP'13, pages 61–72. ACM. doi: 10.1145/2500365.2500577. ↰ pages 51, 53, and 56

James Chapman, Pierre-Évariste Dagand, Conor McBride, and Peter Morris [2010]. The gentle art of levitation. In *International Conference on Functional Programming*, ICFP'10, pages 3–14. ACM. doi: 10.1145/1863543.1863547. ↰ page 22

Pierre-Évariste Dagand and Conor McBride [2012]. Transporting functions across ornaments. In *International Conference on Functional Programming*, ICFP'12, pages 103–114. ACM. doi: 10.1145/2364527.2364544. ↰ pages 7, 10, 21, and 55

Conor McBride [2011]. Ornamental algebras, algebraic ornaments. To appear in *Journal of Functional Programming*. ↰ pages 1 and 21

Stefan Monnier and David Haguenauer [2010]. Singleton types here, singleton types there, singleton types everywhere. In *Programming Languages meets Program Verification*, PLPV'10, pages 1–8. ACM. doi: 10.1145/1707790.1707792. ↰ page 20

Chris Okasaki [1999]. *Purely functional data structures*. Cambridge University Press. ↰ pages 36, 37, 45, and 50

Wouter Swierstra [2008]. Data types à la carte. *Journal of Functional Programming*, 18(4):423–436. doi: 10.1017/S0956796808006758. ↰ page 22

Todo list